Mergesort and Quicksort

Sorting algorithms

- Insertion, selection and bubble sort have quadratic worst-case performance
- The faster comparison based algorithm ?

  O(nlogn)

- Mergesort and Quicksort

Merge Sort

- Apply divide-and-conquer to sorting problem
- Problem: Given *n* elements, sort elements into non-decreasing order
- Divide-and-Conquer:
  - If n=1 terminate (every one-element list is already sorted)
  - If n>1, partition elements into two or more sub-collections; sort each; combine into a single sorted list
- How do we partition?

Partitioning - Choice 1

- First n-1 elements into set A, last element set B
- Sort A using this partitioning scheme recursively
  - B already sorted
- Combine A and B using method Insert() (= insertion into sorted array)
- Leads to recursive version of InsertionSort()
  - Number of comparisons: $O(n^2)$
    - Best case = n-1
    - Worst case = [redacted]

Partitioning - Choice 2

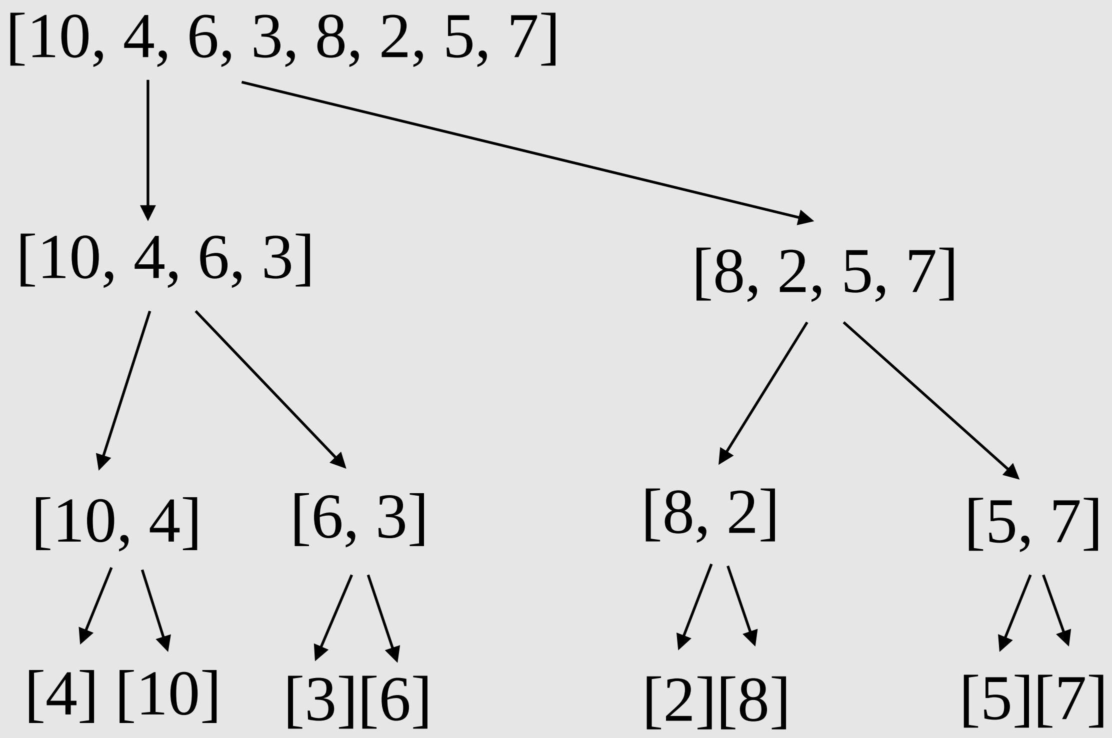
- Put element with largest key in B, remaining elements in A
- Sort A recursively
- To combine sorted A and B, append B to sorted A
  - Use Max() to find largest element $\rightarrow$ recursive SelectionSort()
  - Use bubbling process to find and move largest element to right-most position $\rightarrow$ recursive BubbleSort()
- All $O(n^2)$

Partitioning - Choice 3

- Let's try to achieve balanced partitioning
- A gets *n/2* elements, B gets rest half
- Sort A and B recursively
- Combine sorted A and B using a process called *merge*, which combines two sorted lists into one
    - How? We will see soon

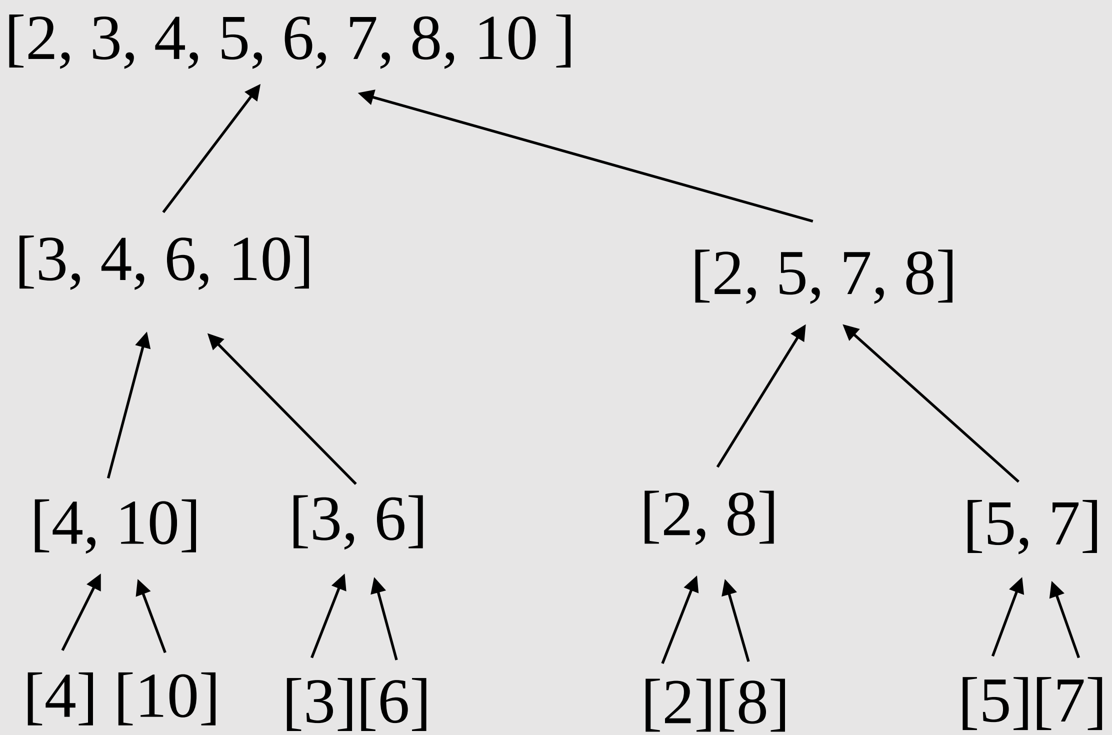
Example

- Partition into lists of size n/2



Example Cont’d

- Merge



Static Method mergeSort()

```
Public static void mergeSort(Comparable []a, int left, int
 right)
{
 // sort a[left:right]
 if (left < right)
 {// at least two elements
    int mid = (left+right)/2;   //midpoint
    mergeSort(a, left, mid);
    mergeSort(a, mid + 1, right);
    merge(a, b, left, mid, right); //merge from a to b
    copy(b, a, left, right);  //copy result back to a
 }
}
```

Merge Function

Evaluation

- Recurrence equation:
- Assume n is a power of 2

$$T(n) = \begin{cases} c_1 & \text{if } n=1 \\ 2T(n/2) + c_2 n & \text{if } n>1,\ n=2^k \end{cases}$$

Solution

By Substitution:

$T(n) = 2T(n/2) + c_2 n$

$T(n/2) = 2T(n/4) + c_2 n/2$

$T(n) = 4T(n/4) + 2\ c_2 n$

$T(n) = 8T(n/8) + 3\ c_2 n$

$T(n) = 2^i T(n/2^i) + ic_2 n$

Assuming $n = 2^k$, expansion halts when we get $T(1)$ on right side; this happens when $i=k$ $T(n) = 2^k T(1) + kc_2 n$

Since $2^k=n$, we know $k=\log n$; since $T(1) = c_1$, we get

$T(n) = c_1 n + c_2 n\log n$;

thus an upper bound for $T_{mergeSort}(n)$ is $O(n\log n)$

Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

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

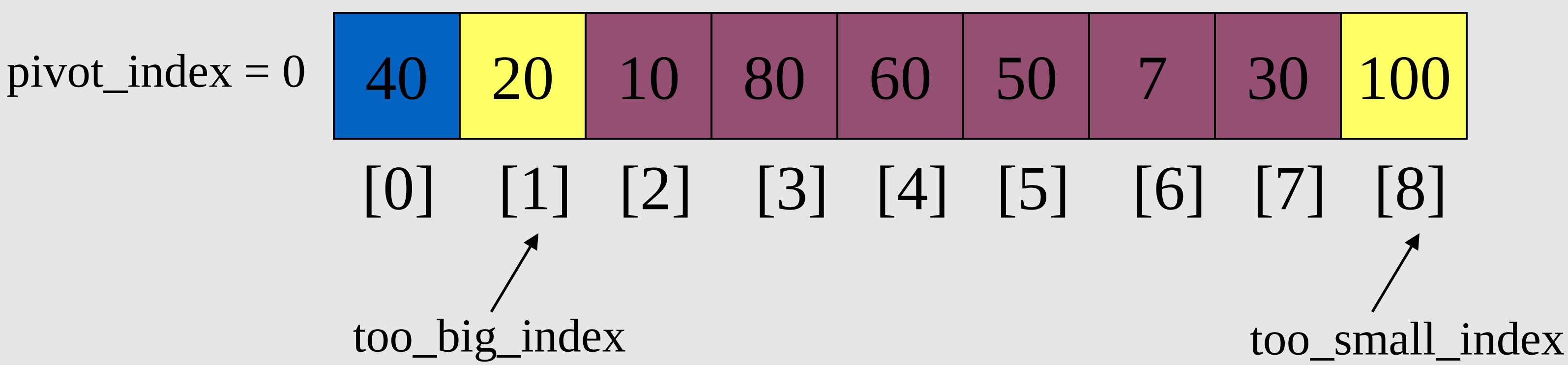
Partitioning Array

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

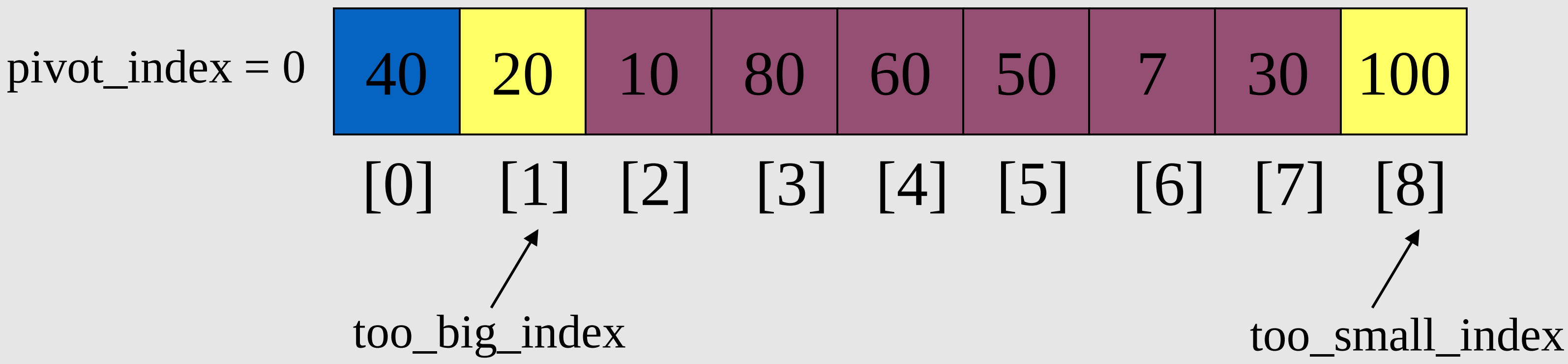
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

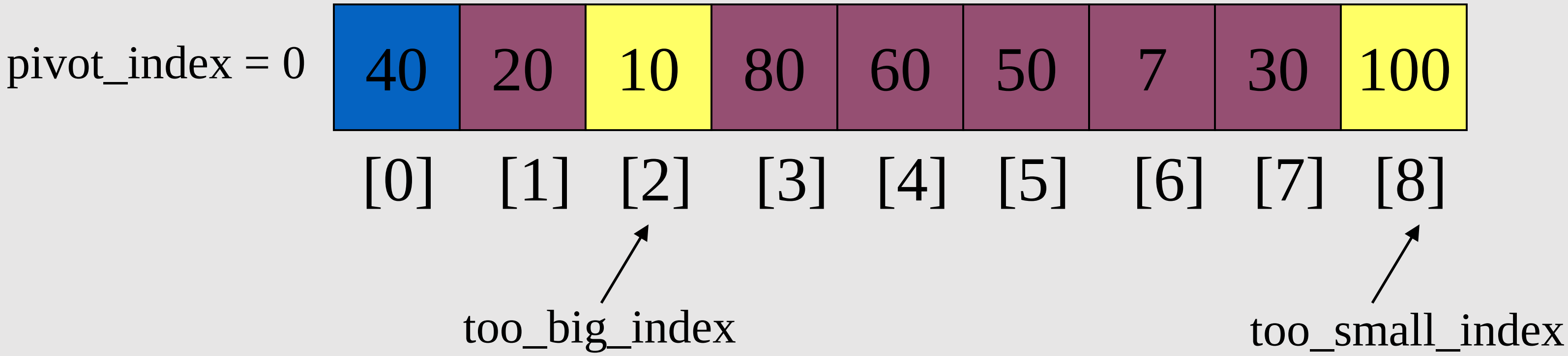
Partitioning loops through, swapping elements below/above pivot.


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 data[too_big_index] <= data[pivot]
   ++too_big_index



1. While data[too_big_index] <= data[pivot]
    ++too_big_index



1. While data[too_big_index] <= data[pivot]
   ++too_big_index

pivot_index = 0

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

too_big_index

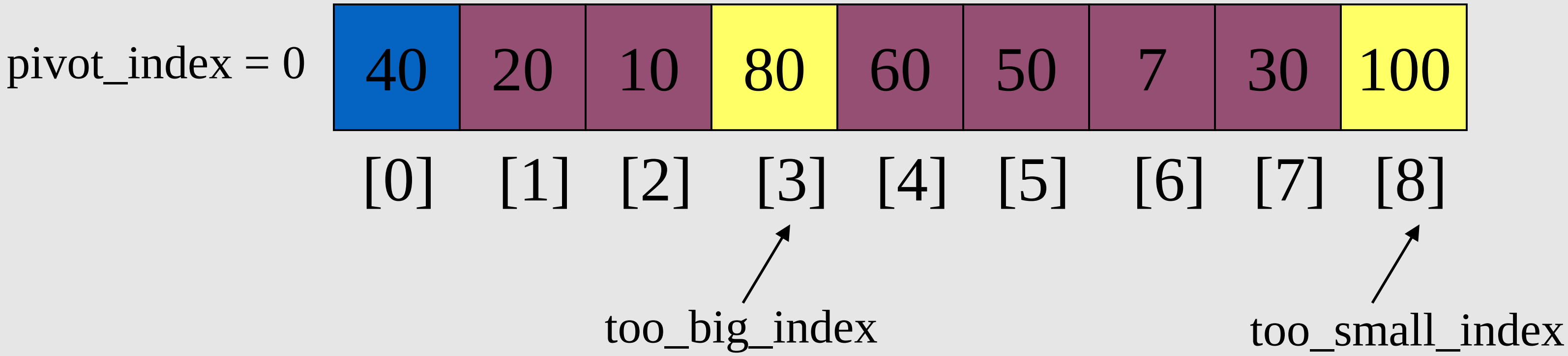
too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

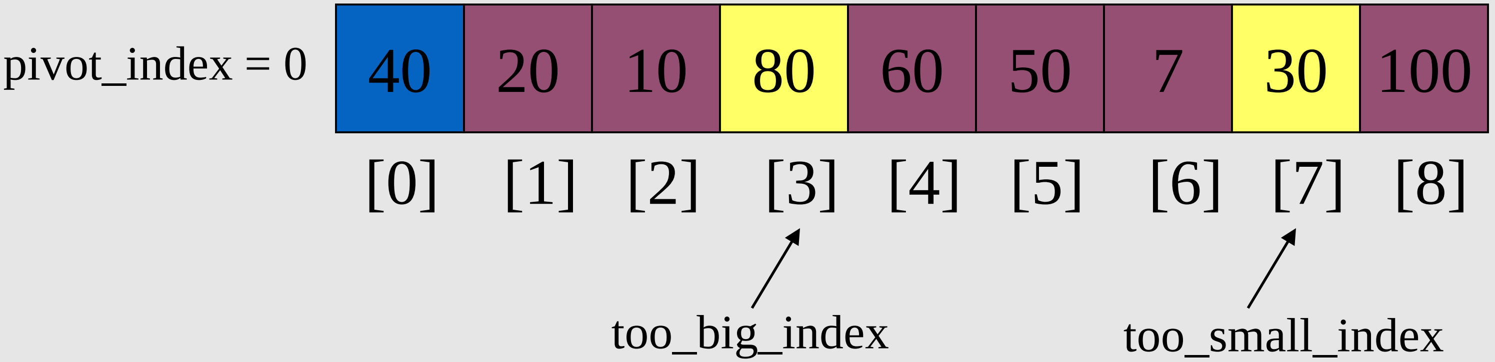


1. While data[too_big_index] <= data[pivot]

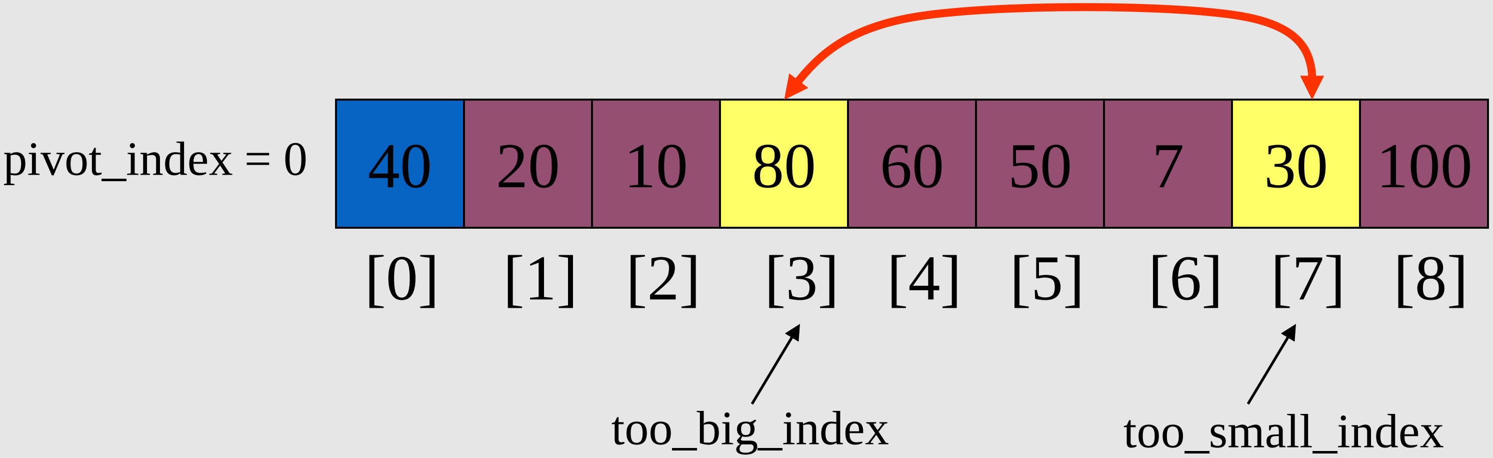
   ++too_big_index

2. While data[too_small_index] > data[pivot]

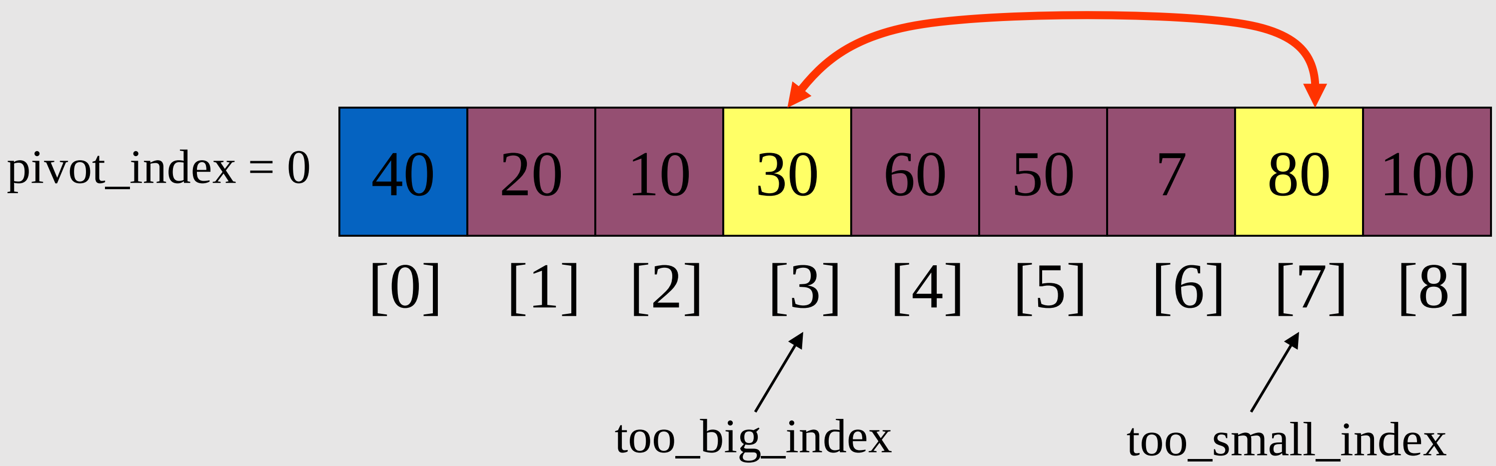
   --too_small_index



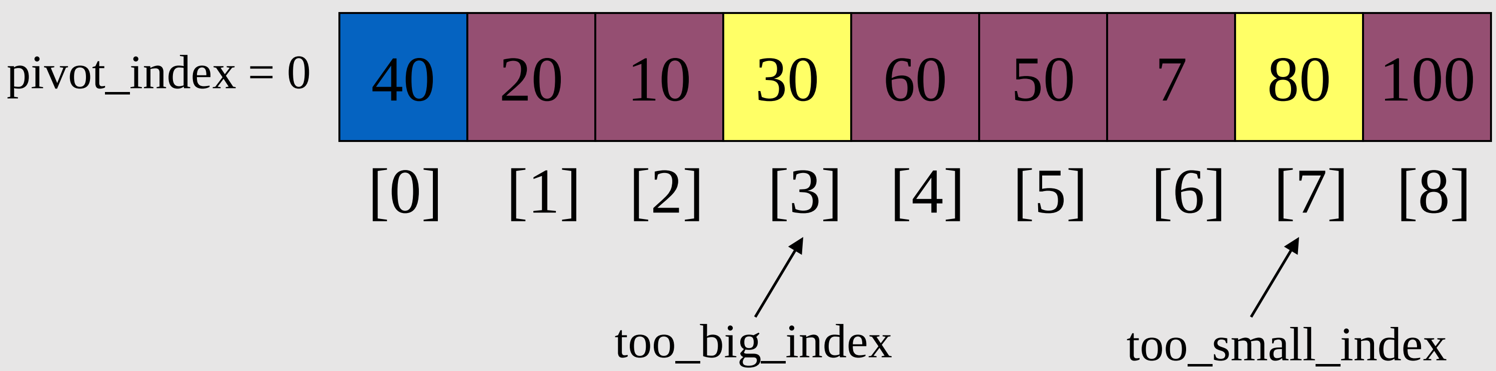
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



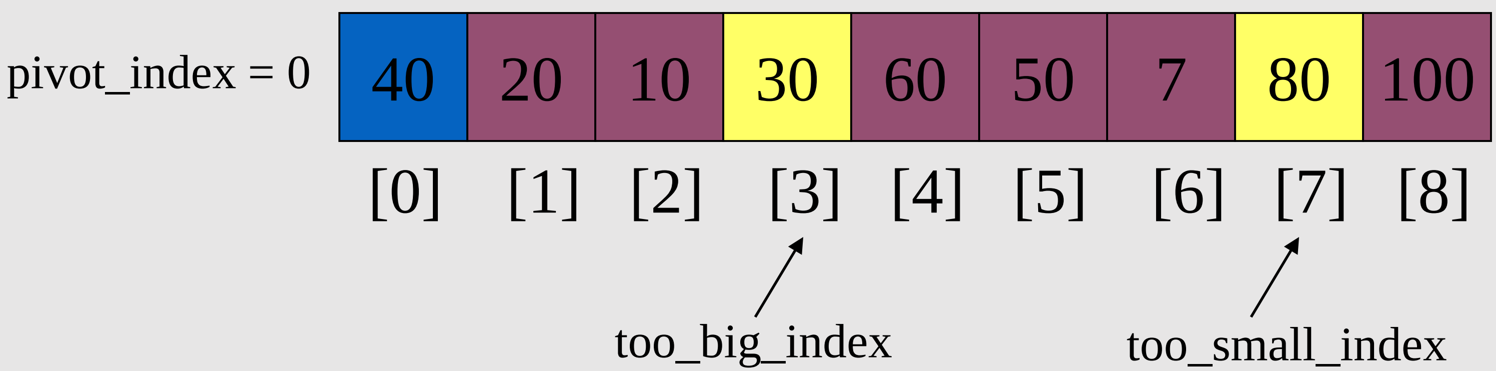
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



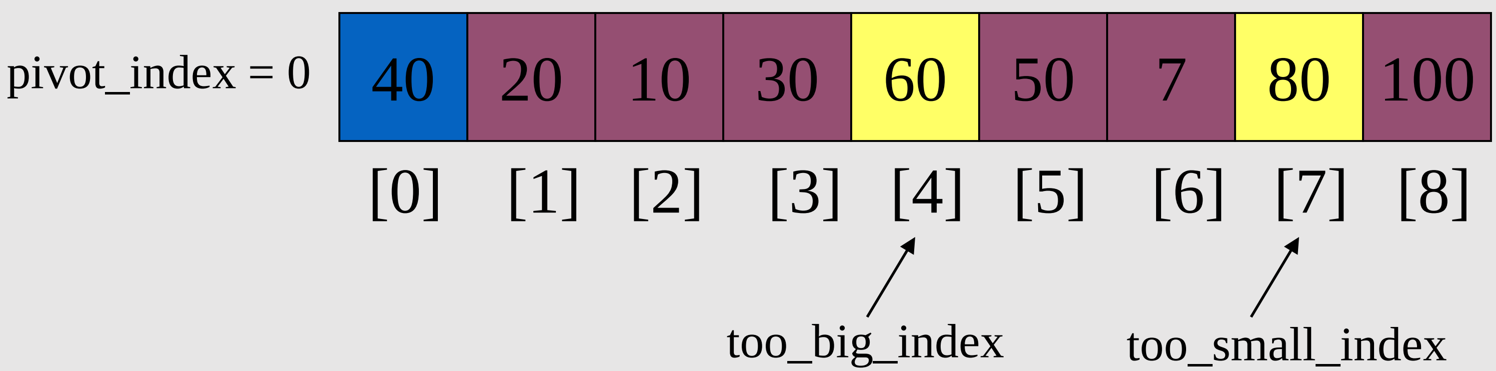
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



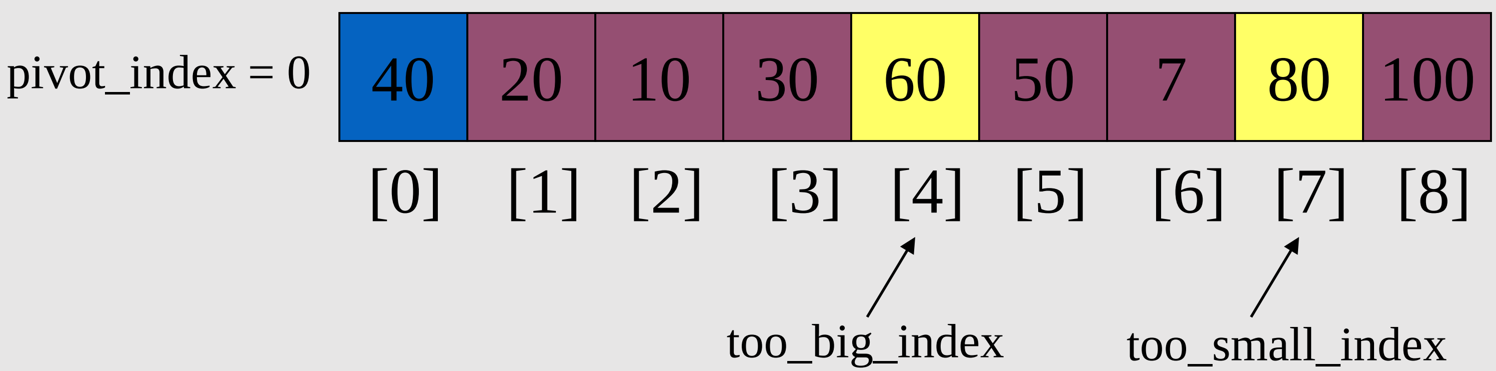
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



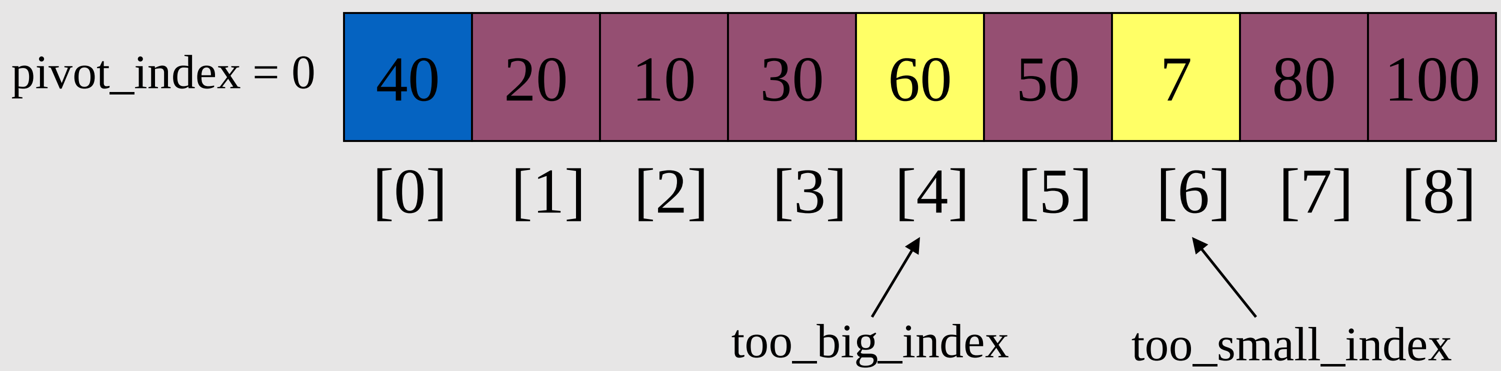
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



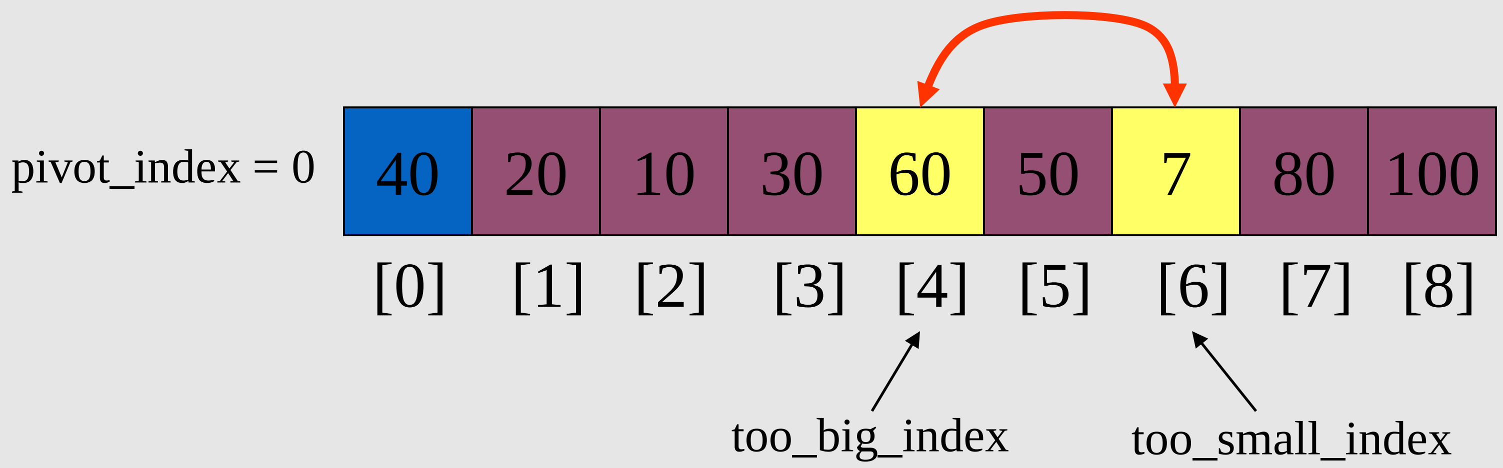
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



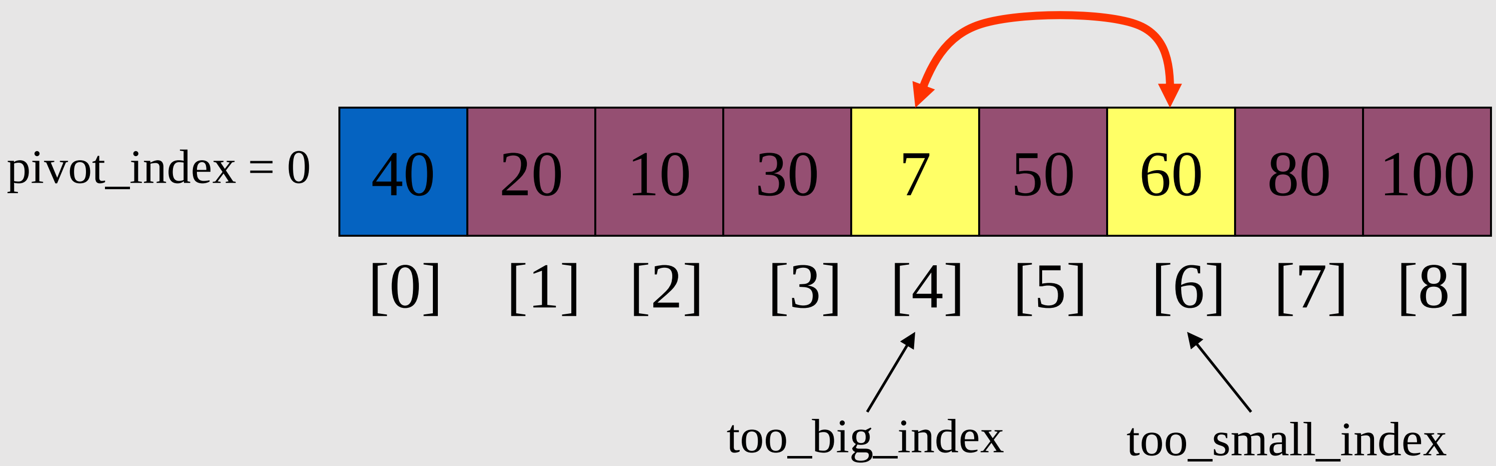
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



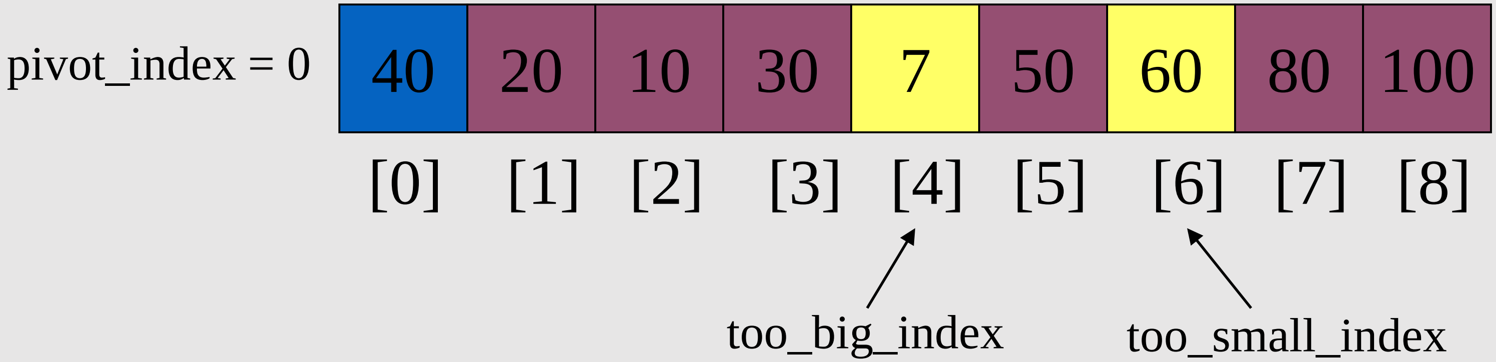
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



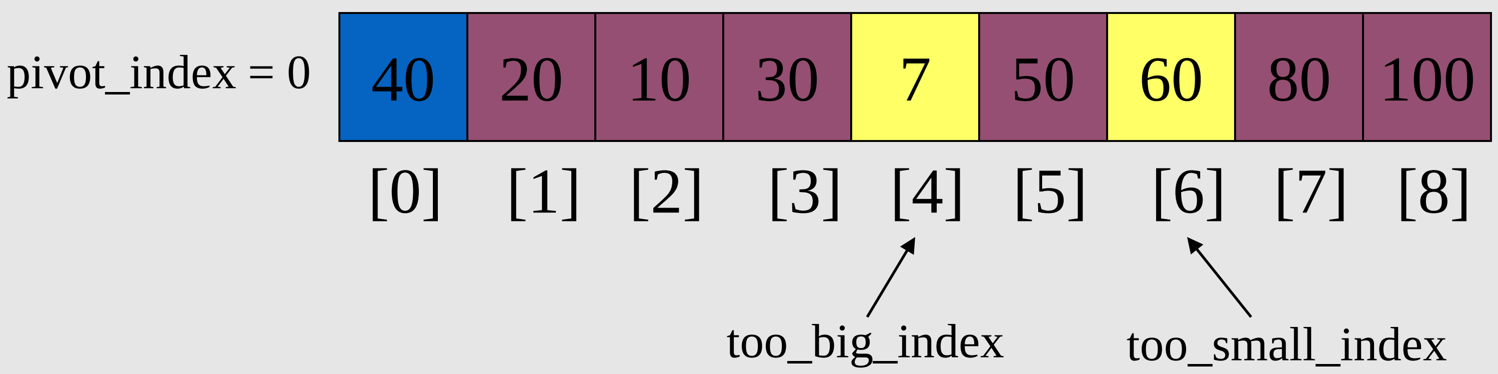
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



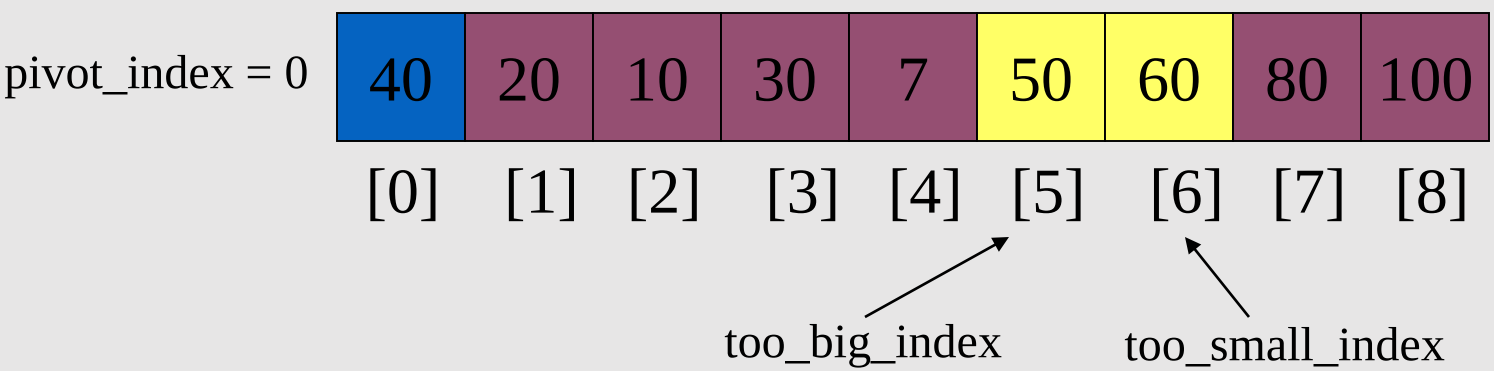
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



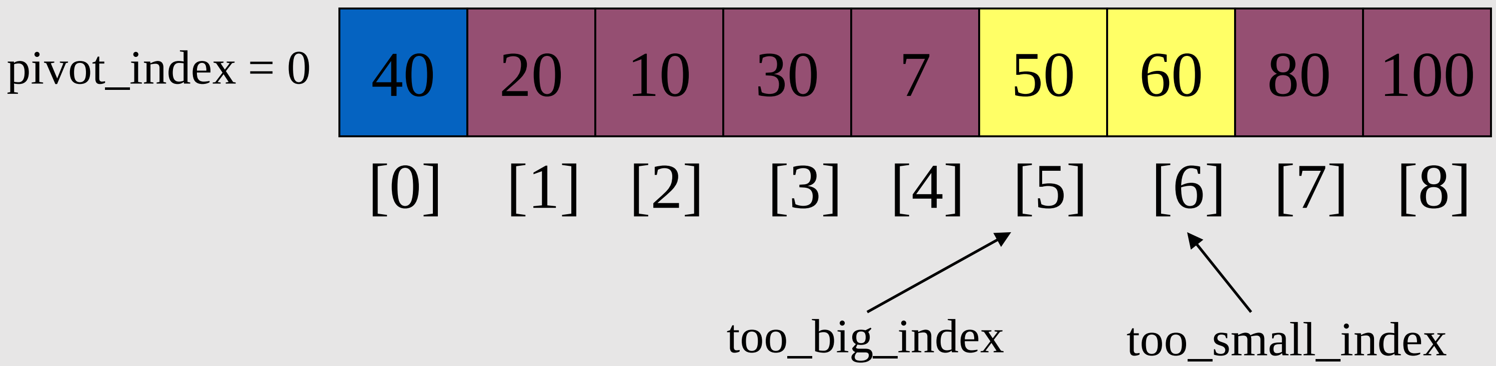
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



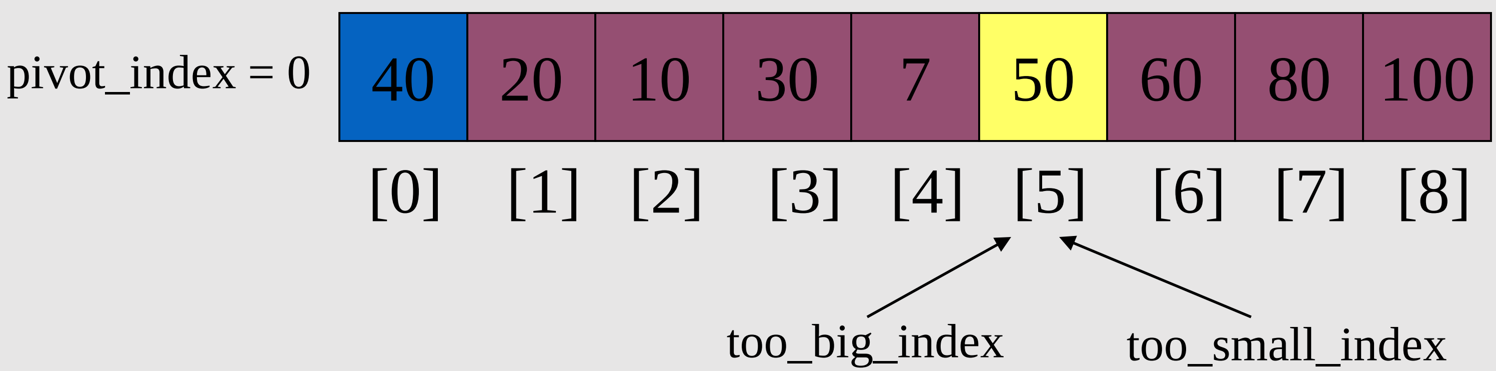
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



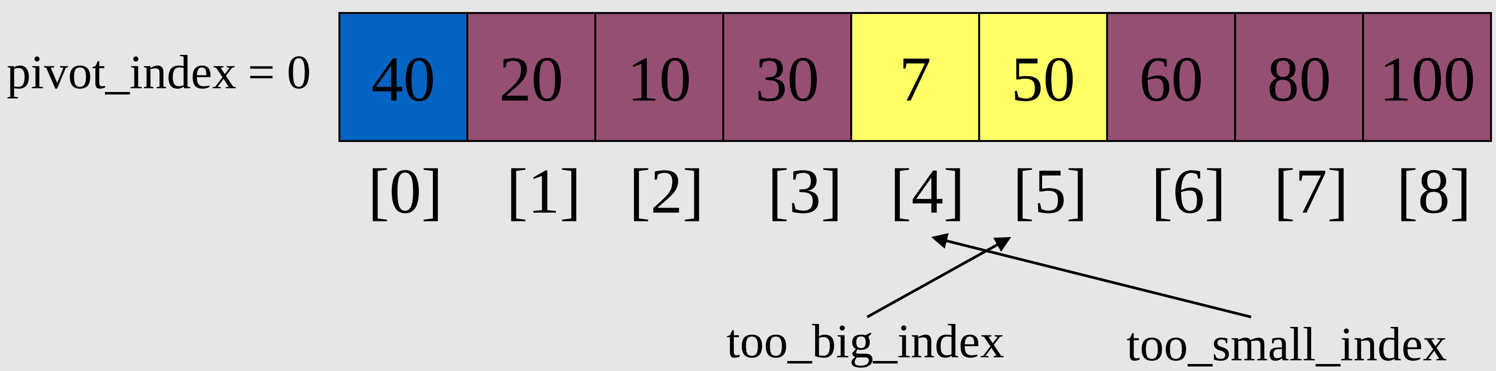
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While 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 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.



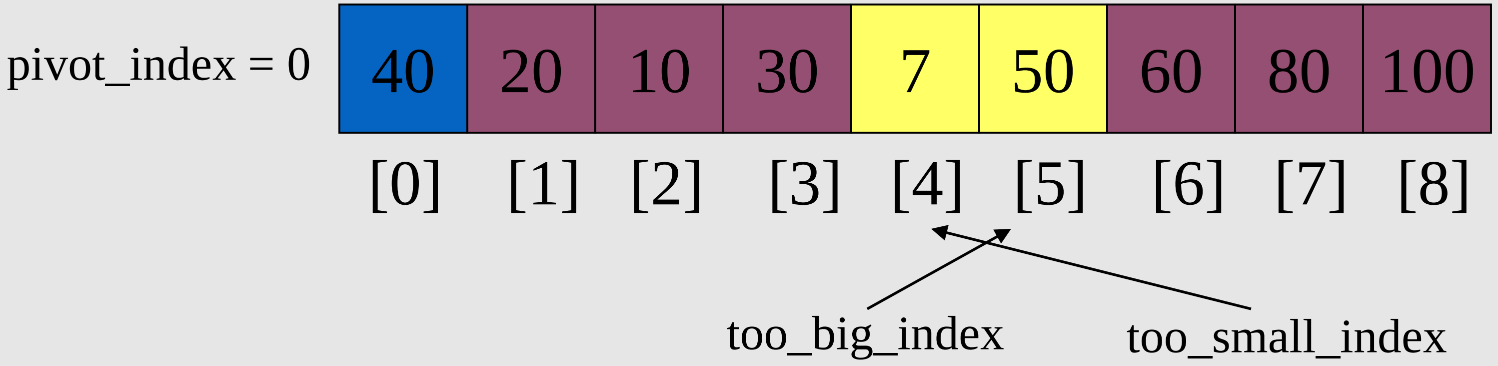
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

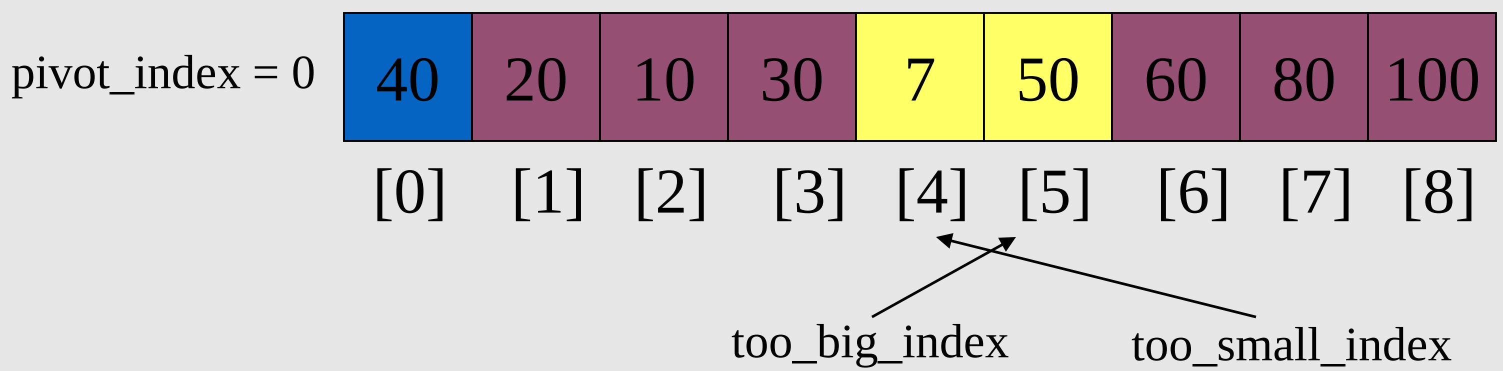
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

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 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.
5. Swap data[too_small_index] and data[pivot_index]



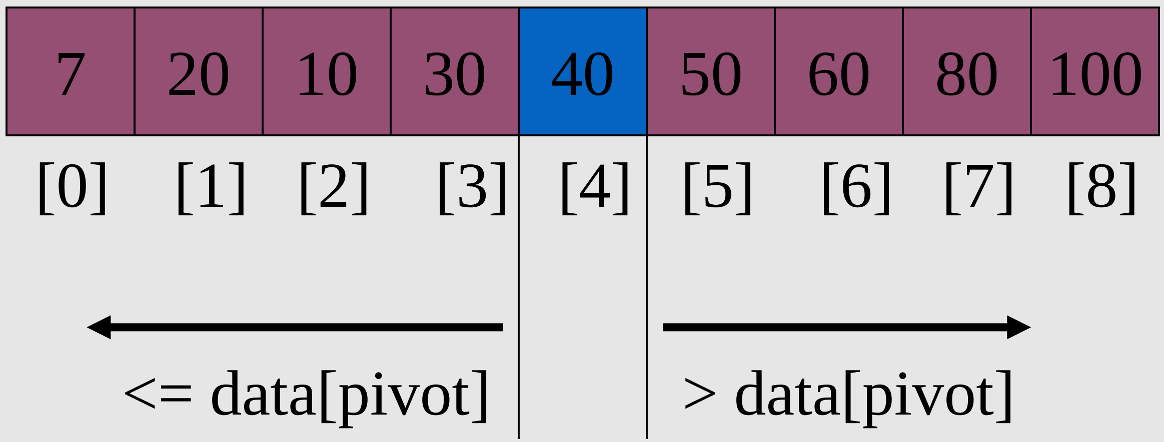
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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

pivot_index = 4

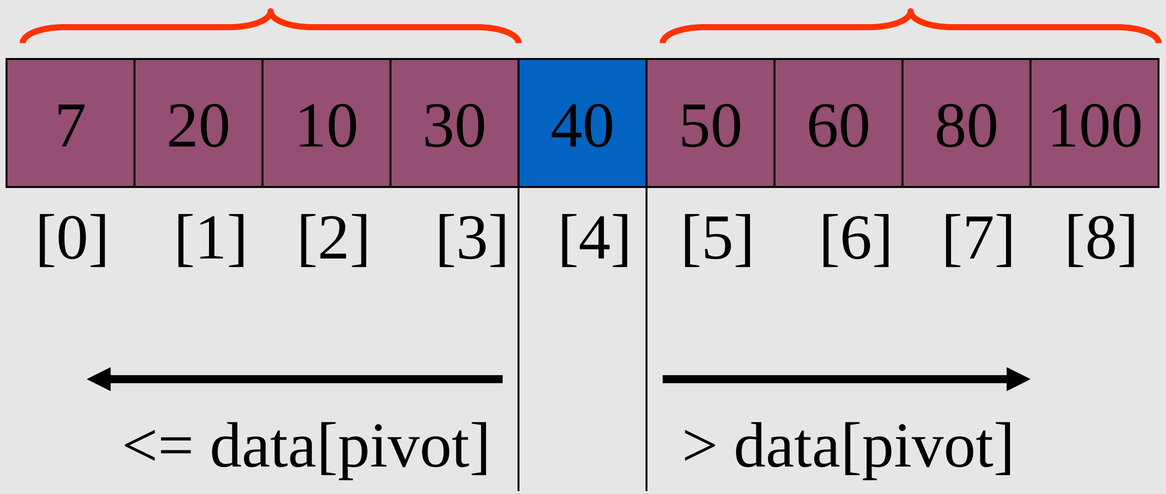
7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

Partition Result



Recursion: Quicksort Sub-arrays



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$

Quicksort Analysis

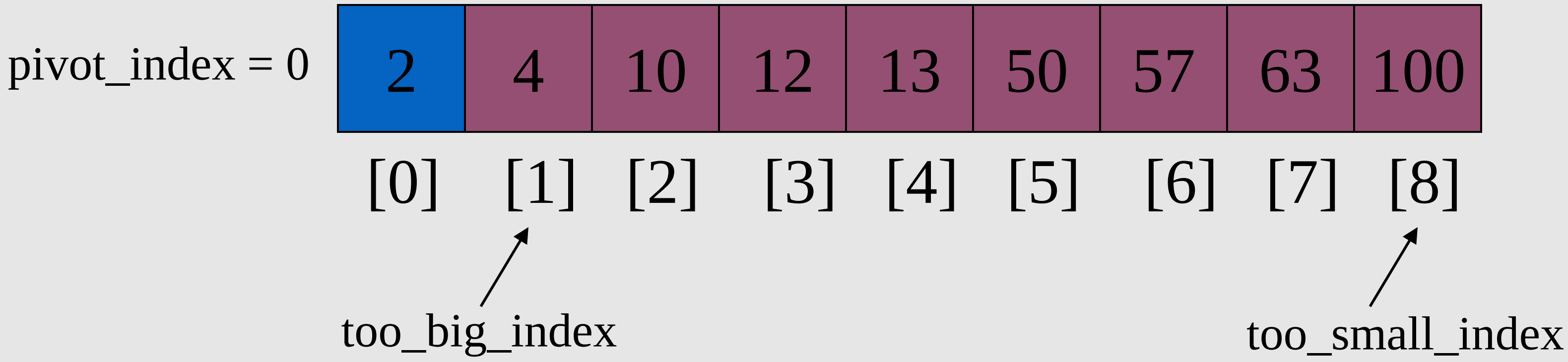
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

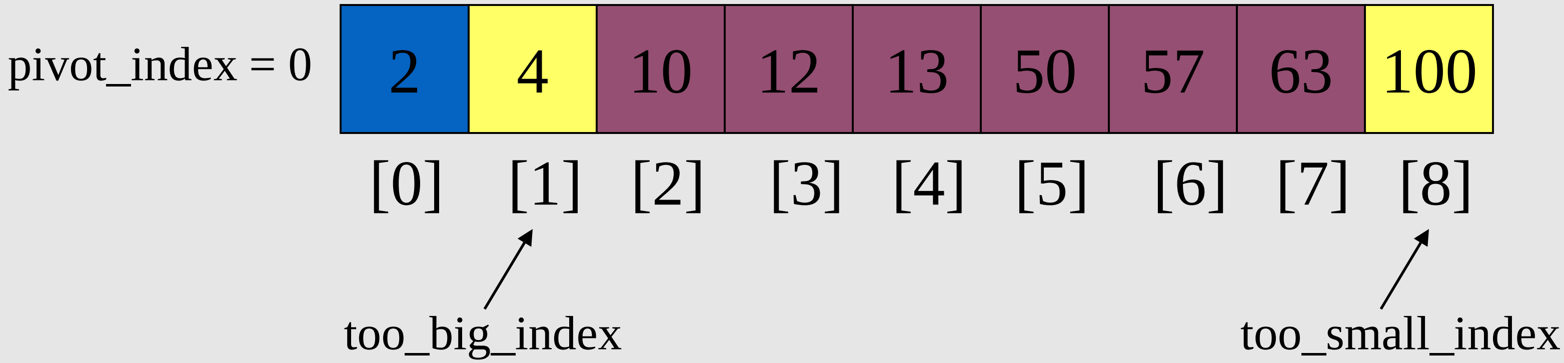
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Quicksort: Worst Case

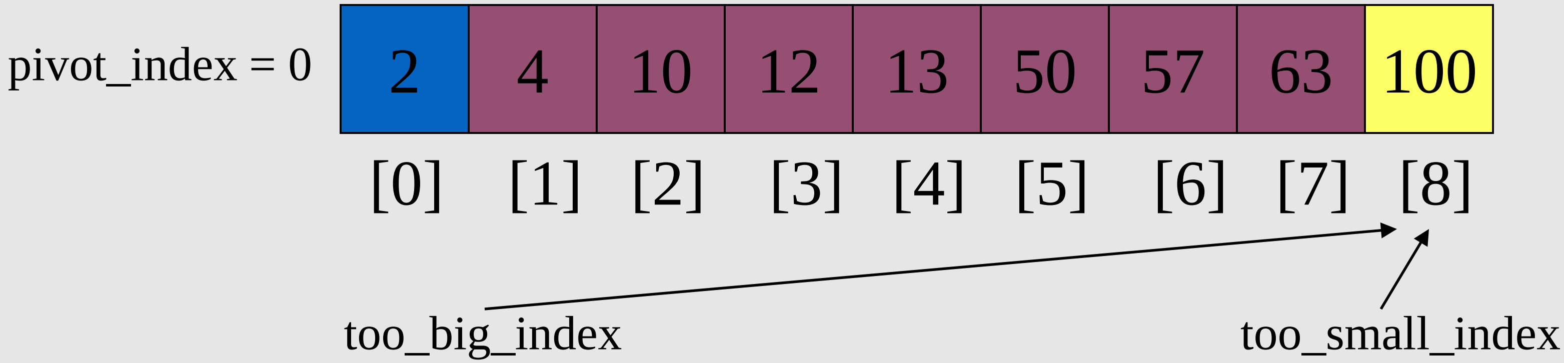
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



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.
5. Swap data[too_small_index] and data[pivot_index]



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.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]

   ++too_big_index

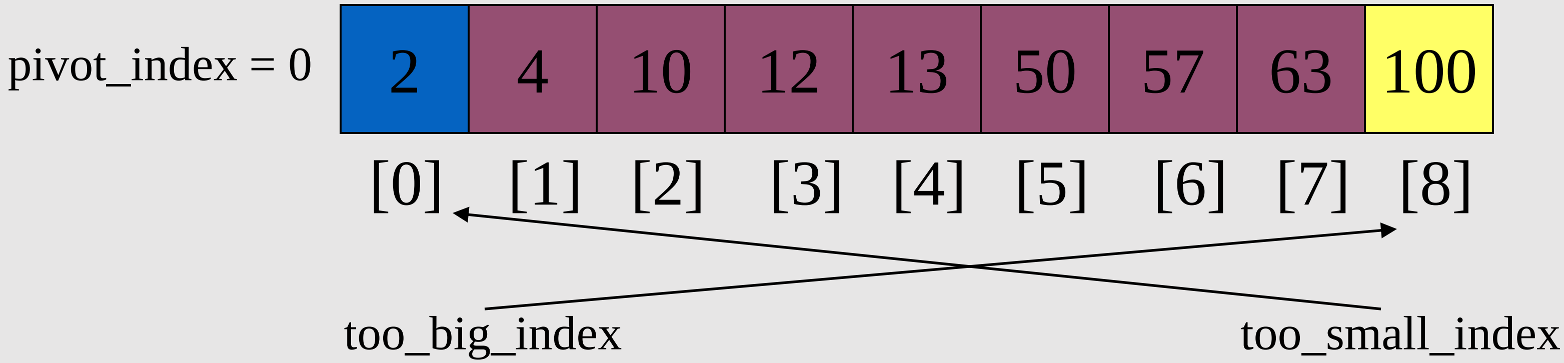
2. While data[too_small_index] > data[pivot]

   --too_small_index

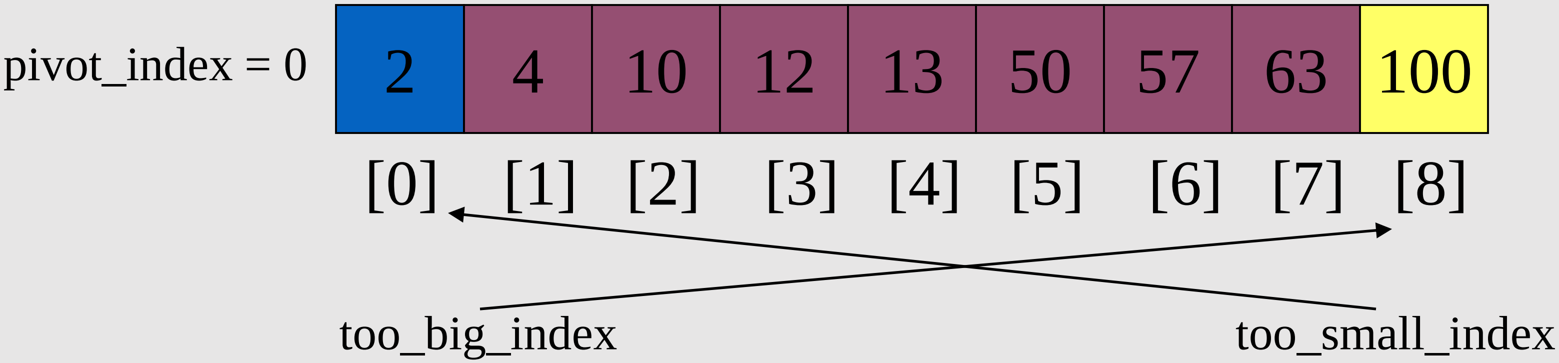
3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

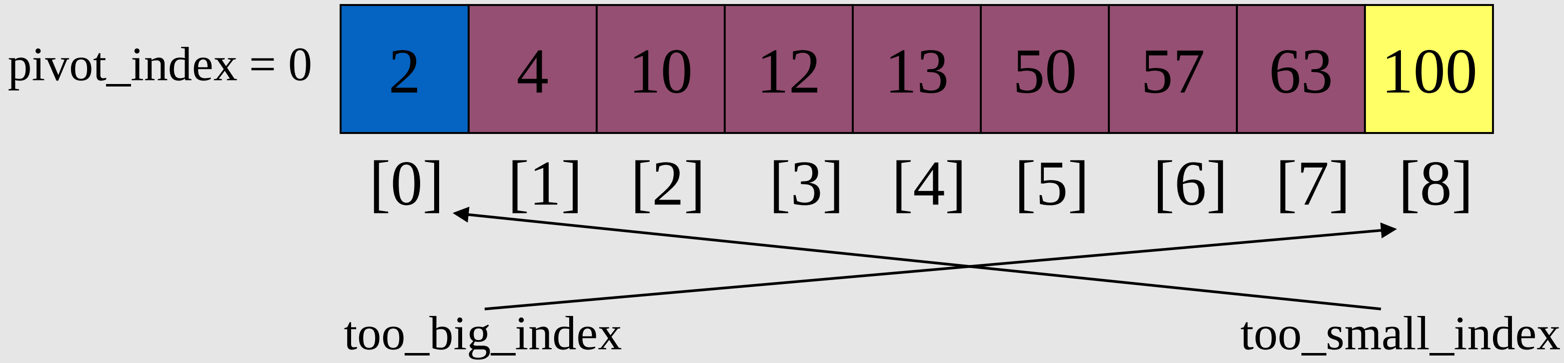
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



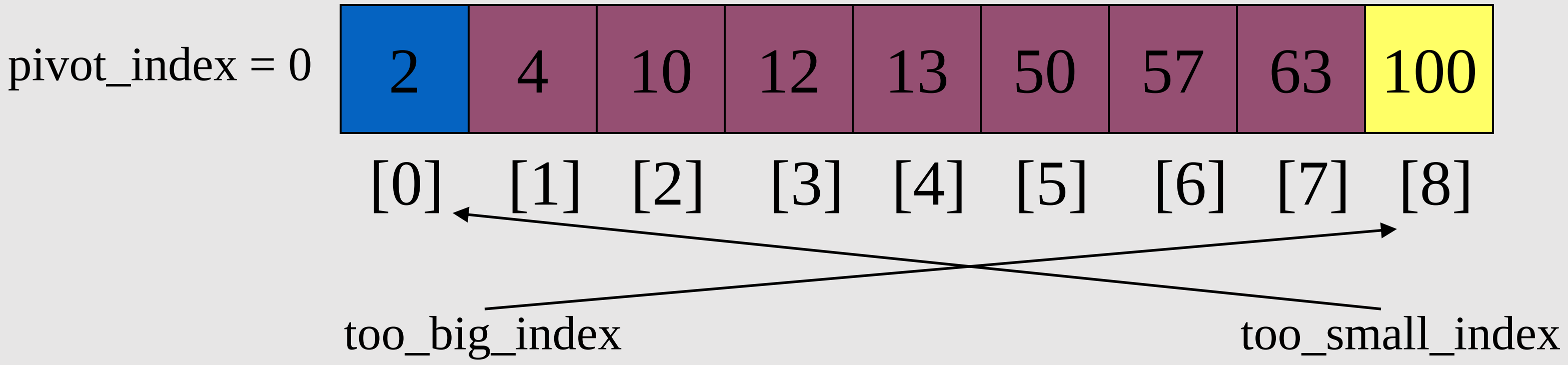
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



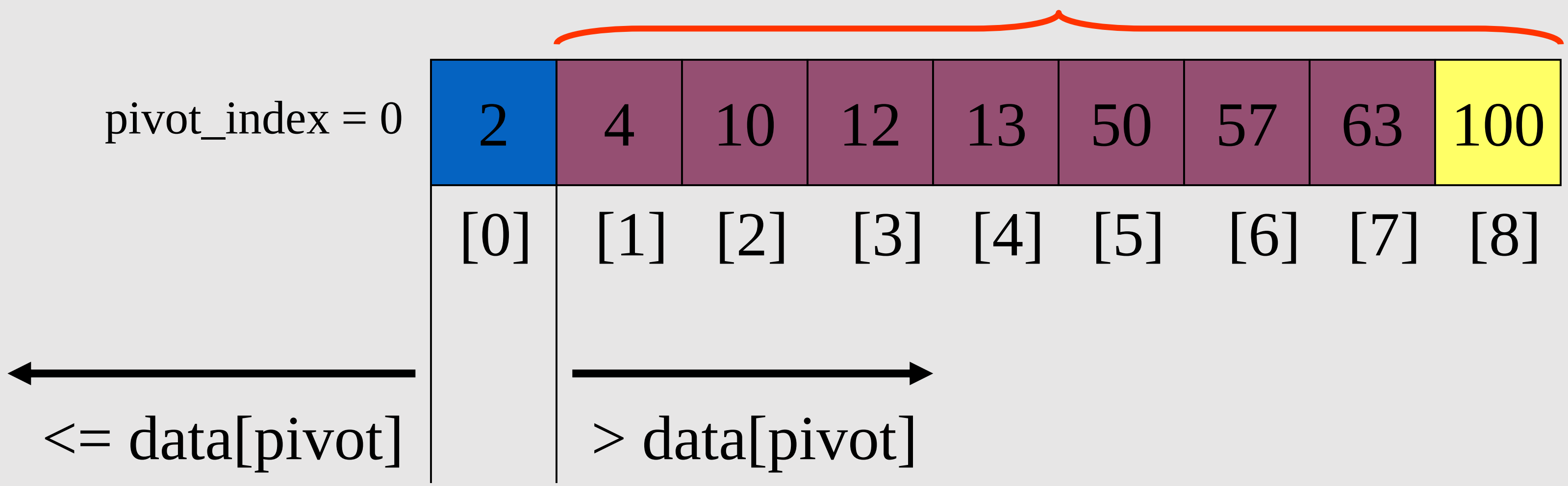
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



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.
5. Swap data[too_small_index] and data[pivot_index]



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.
5. Swap data[too_small_index] and data[pivot_index]



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

data[0], data[n/2], and data[n-1].

Use this median value as pivot.

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    - if(data[first] > data[second]) swap them
  - Sub-array of size 3: left as an exercise.