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 that 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:

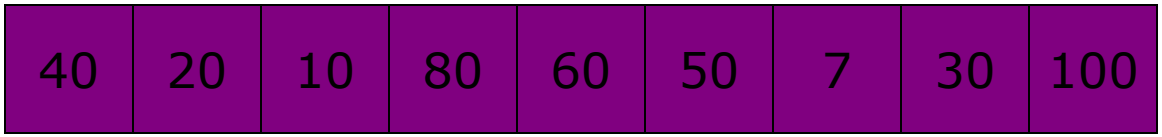
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

Partitioning loops through, swapping elements below/above pivot.

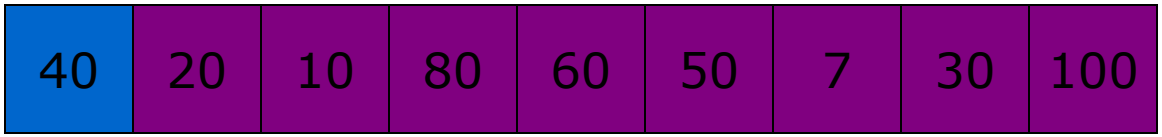
Example

We are given array of n integers to sort:



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:



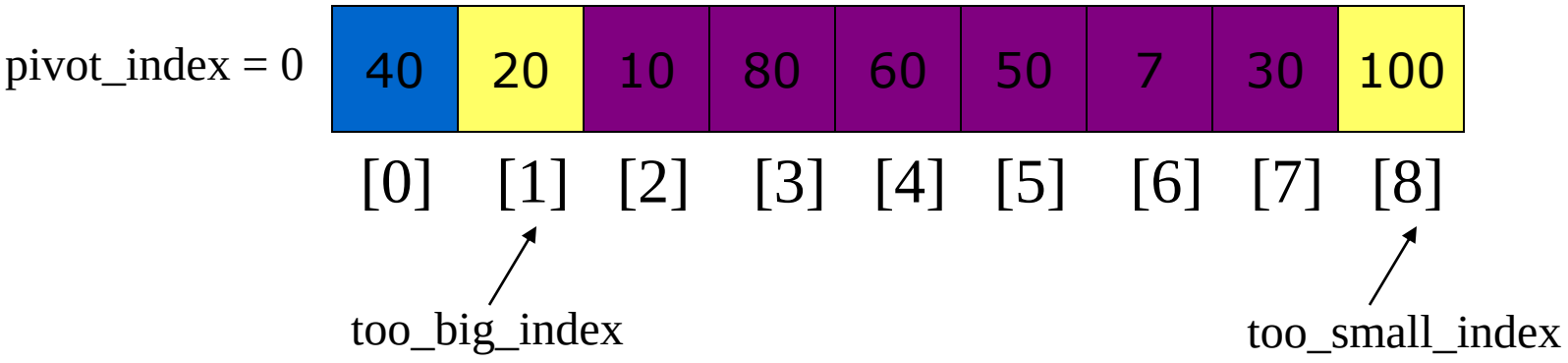
Partitioning Array

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

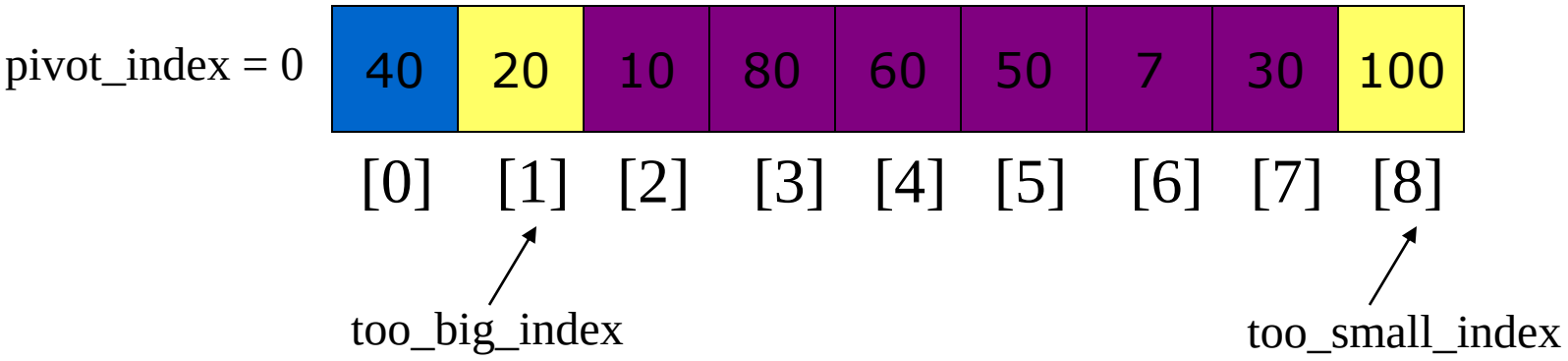
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

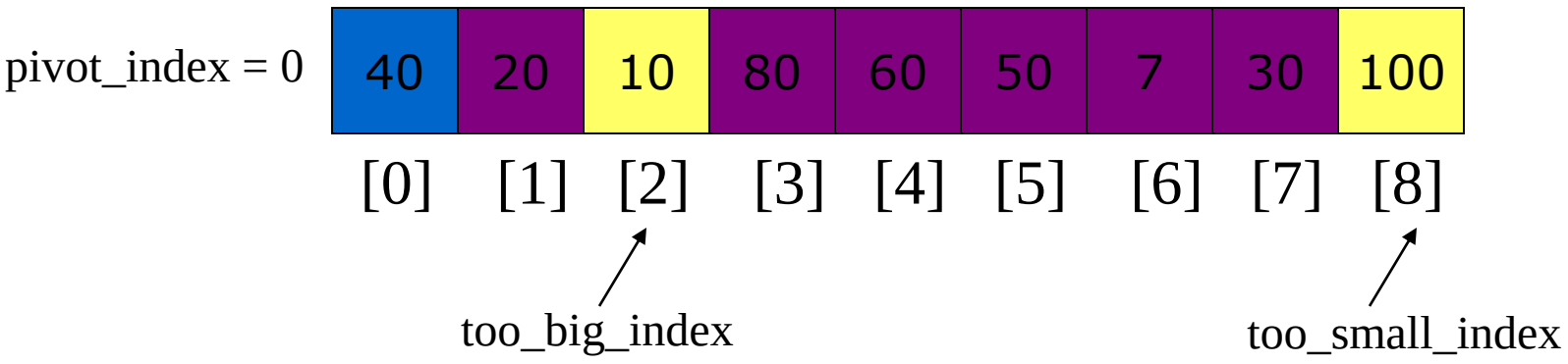
Partitioning loops through, swapping elements below/above pivot.


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

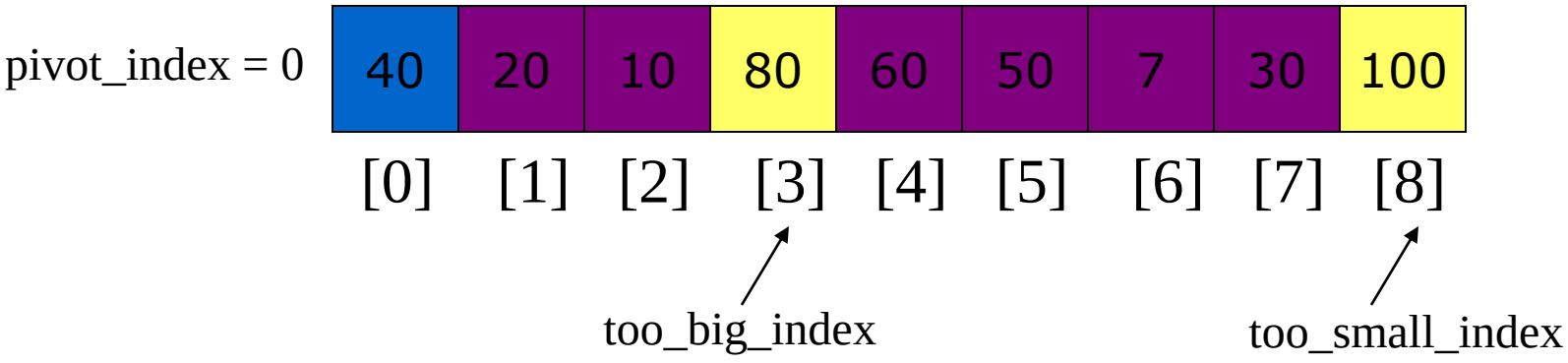
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



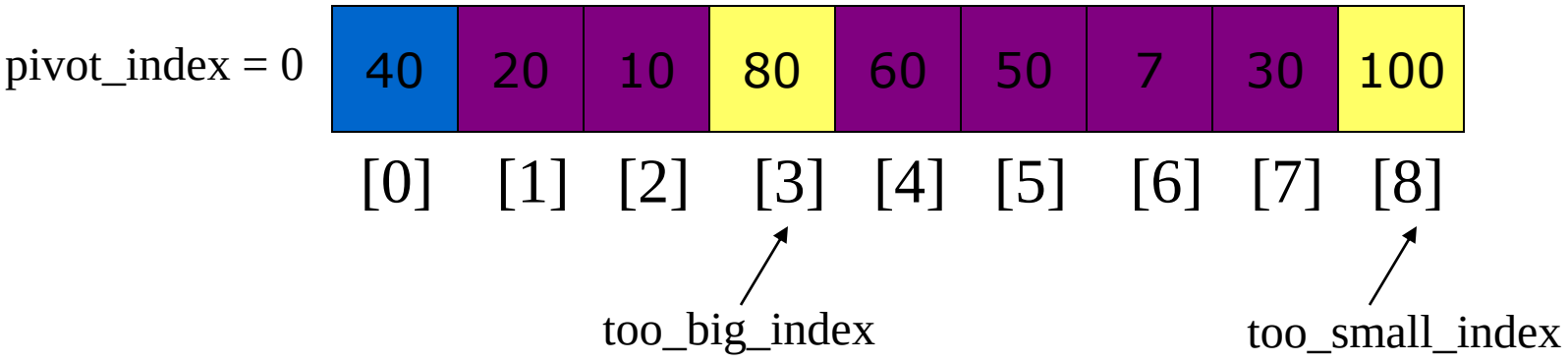
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



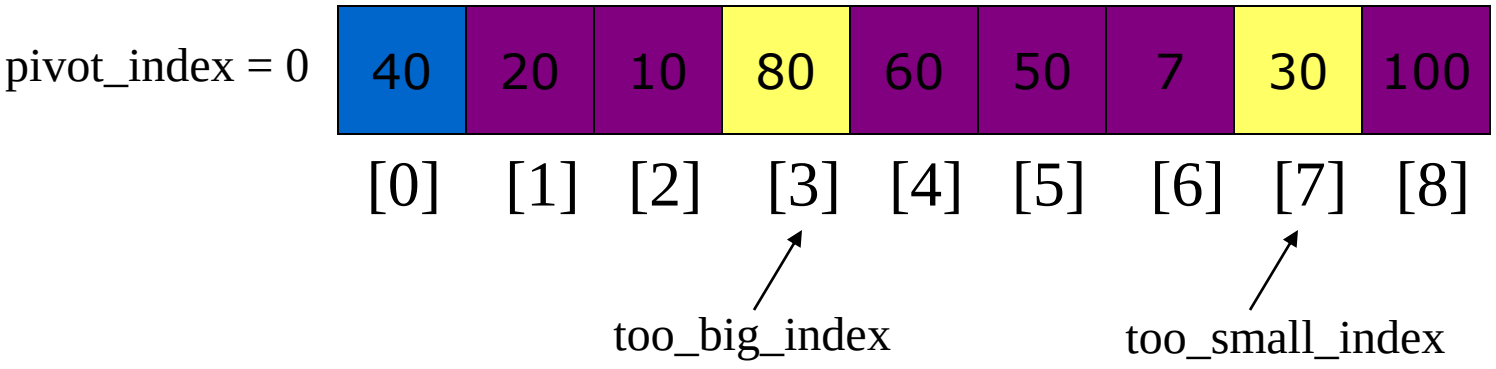
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



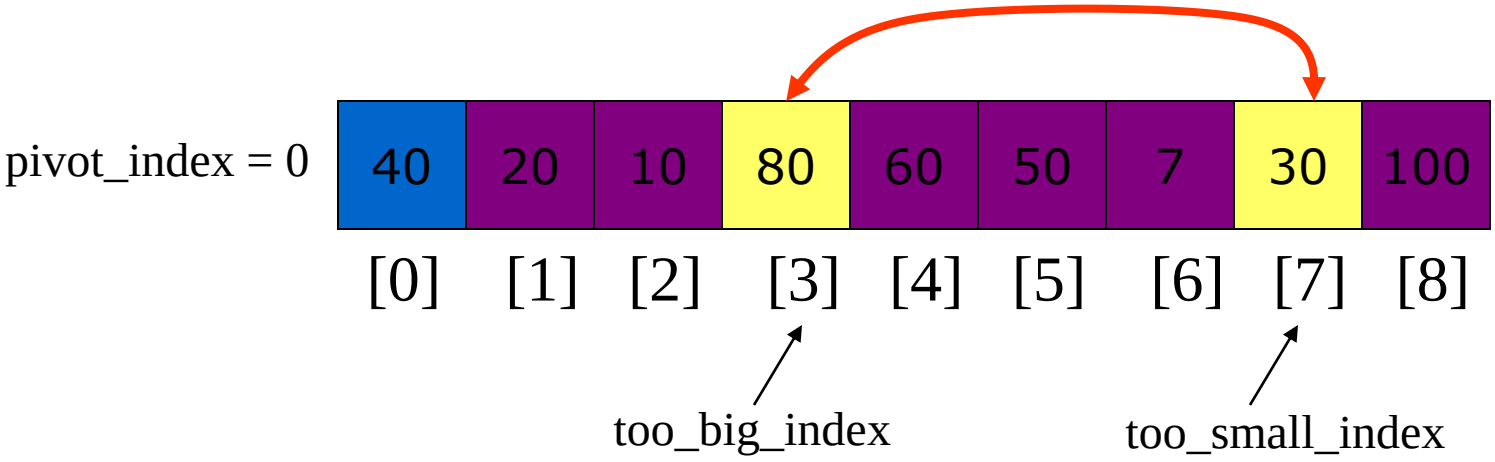
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



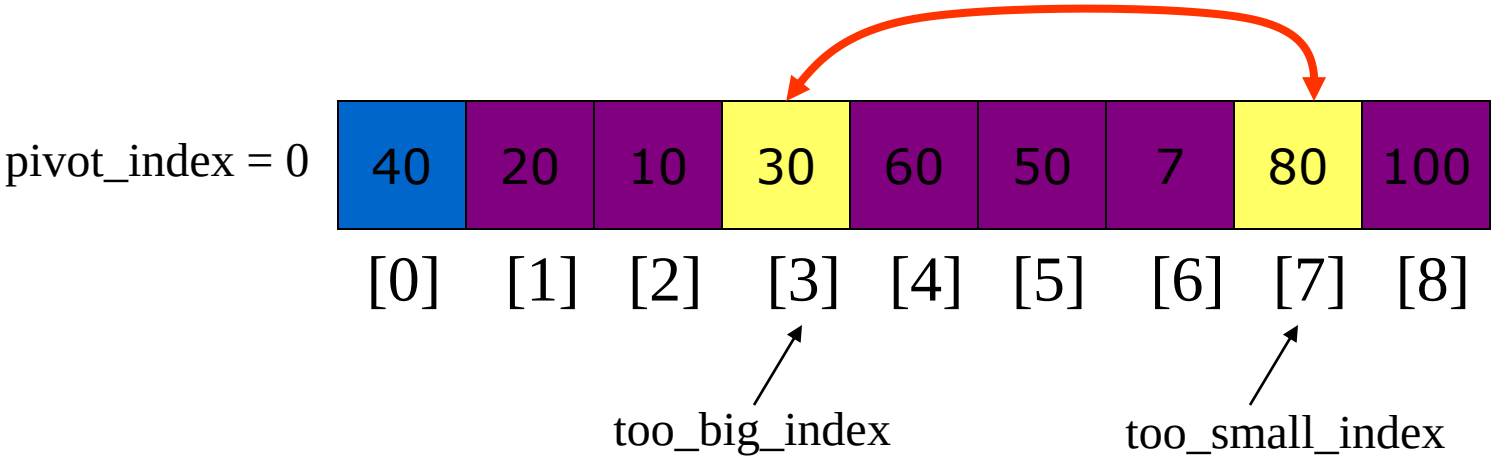
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



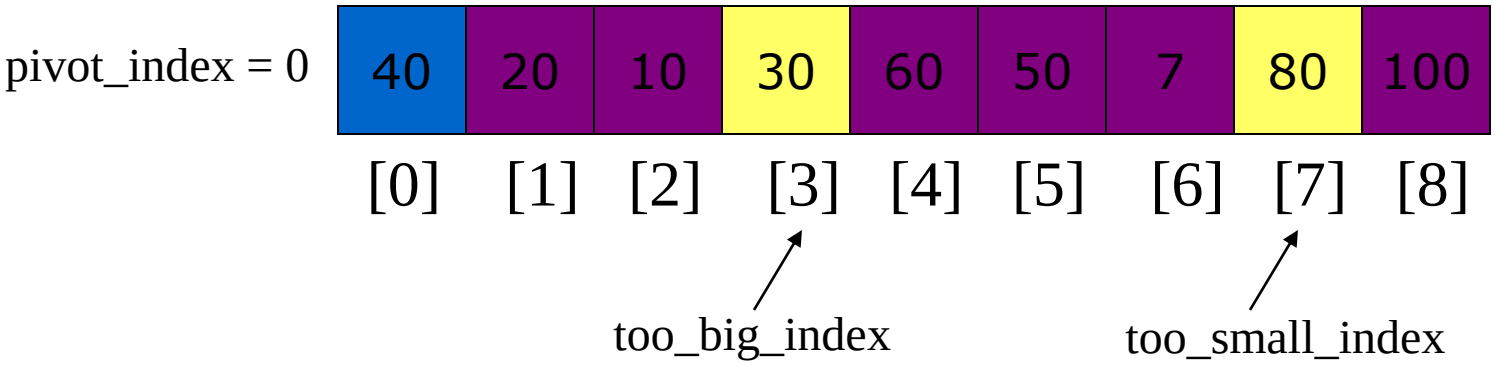
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



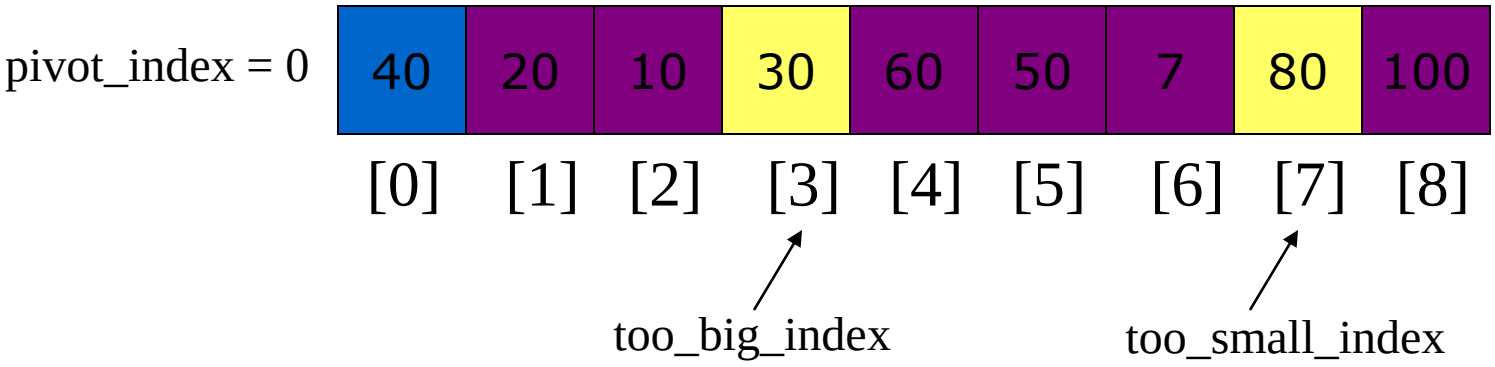
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



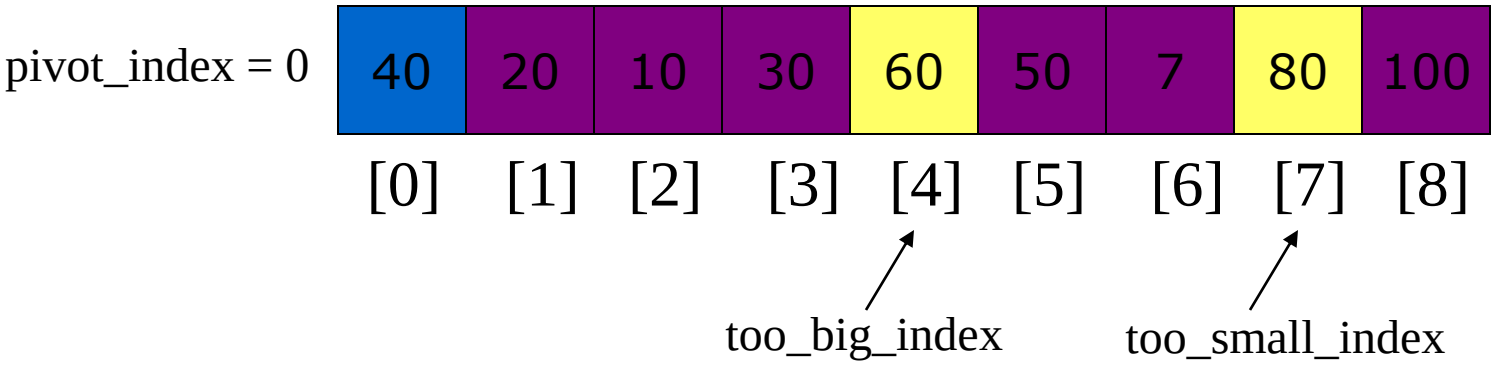
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



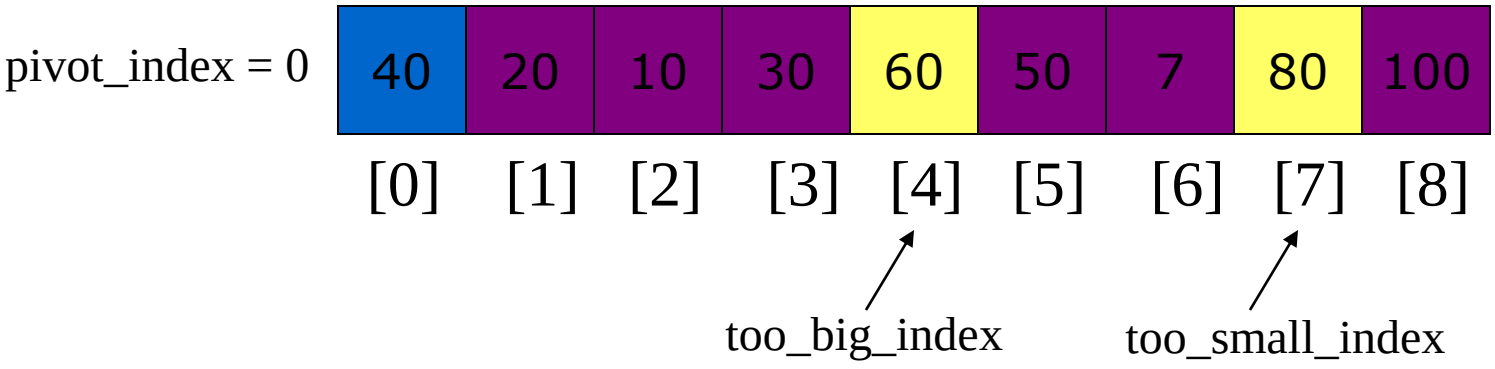
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



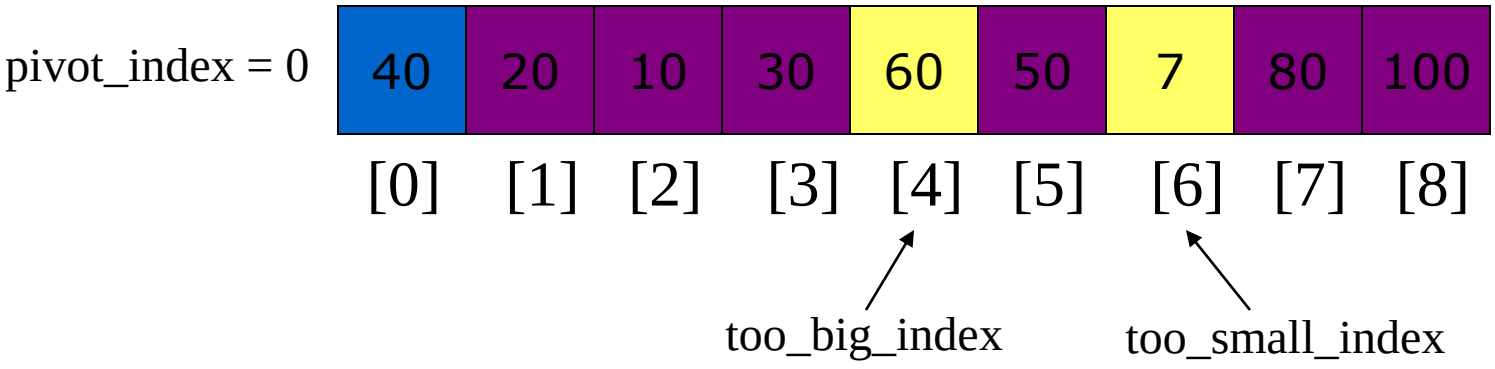
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



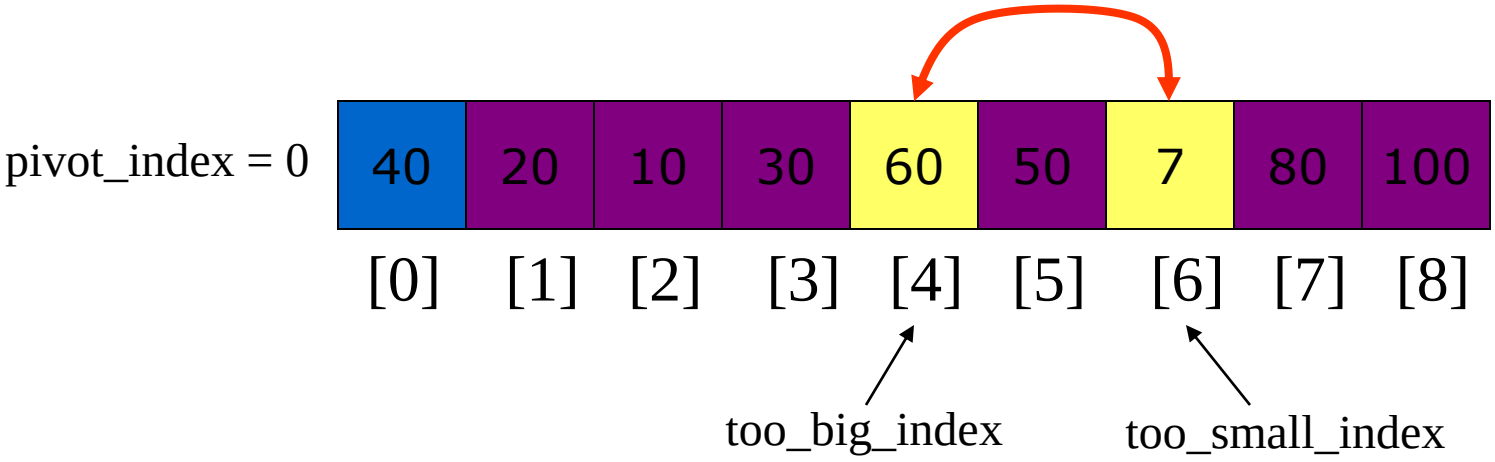
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



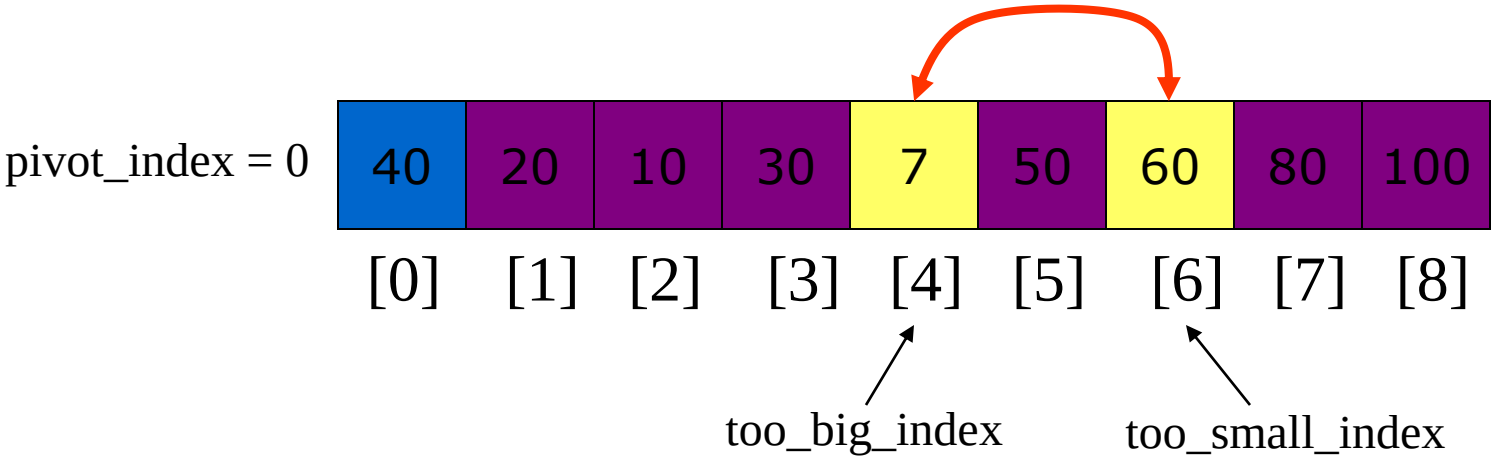
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



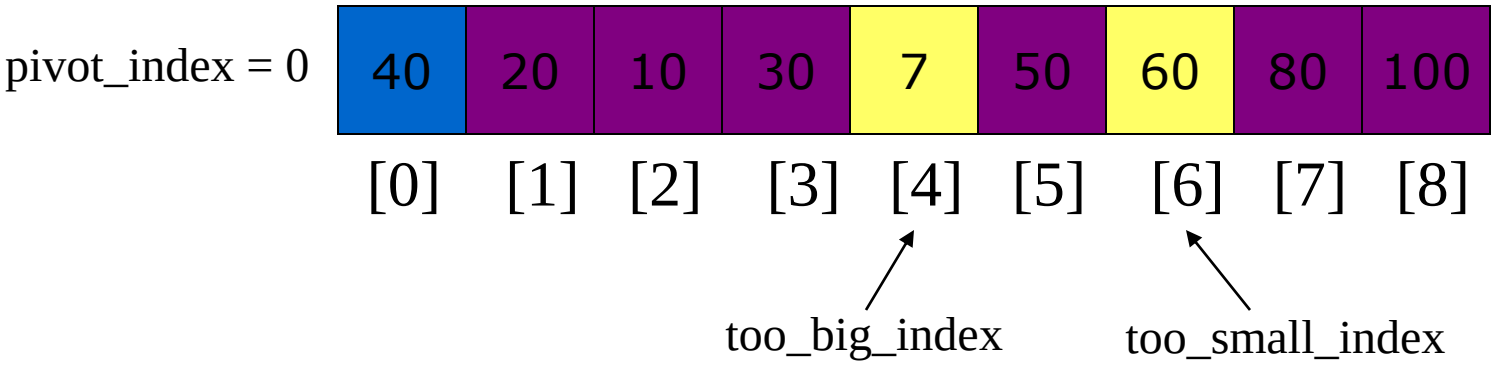
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



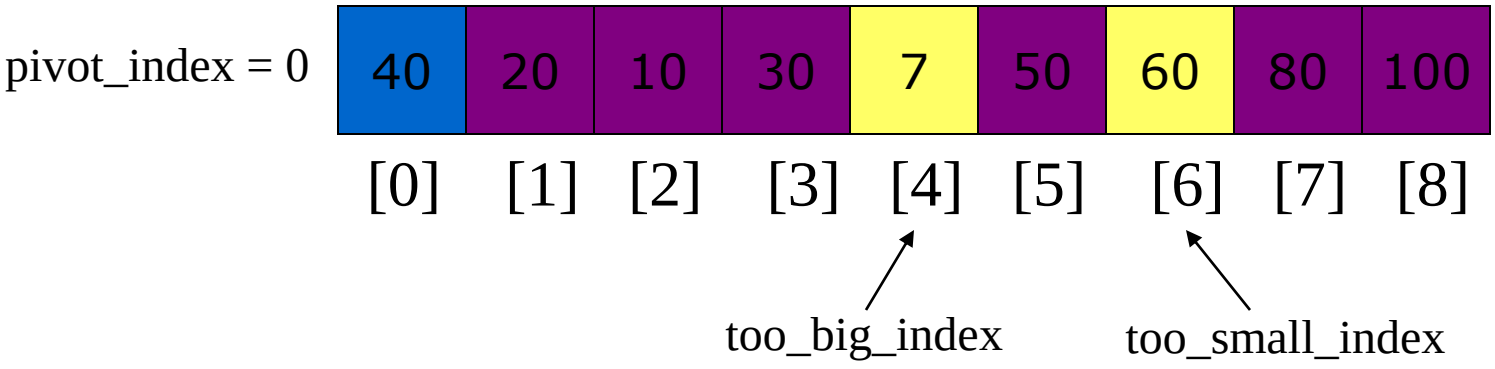
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



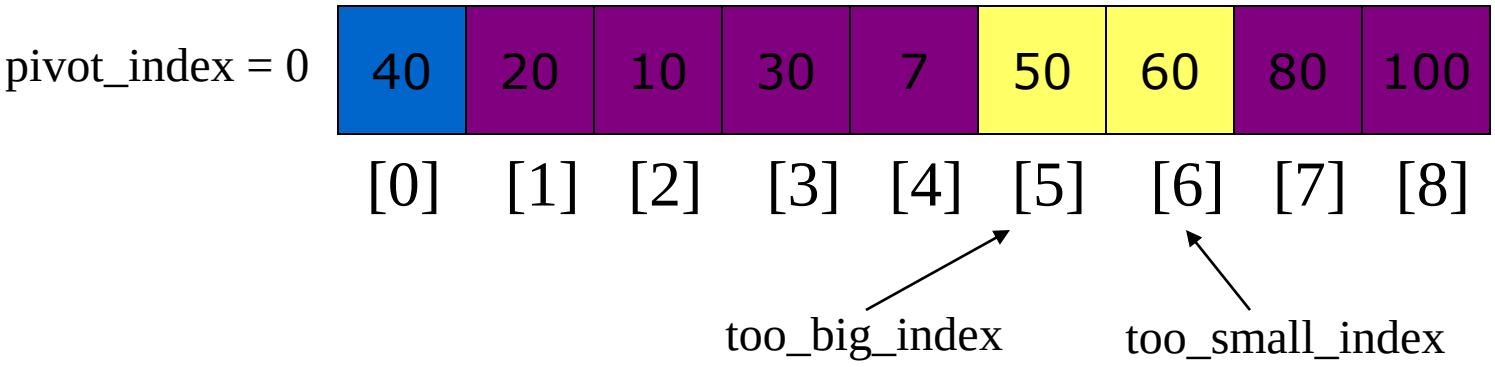
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



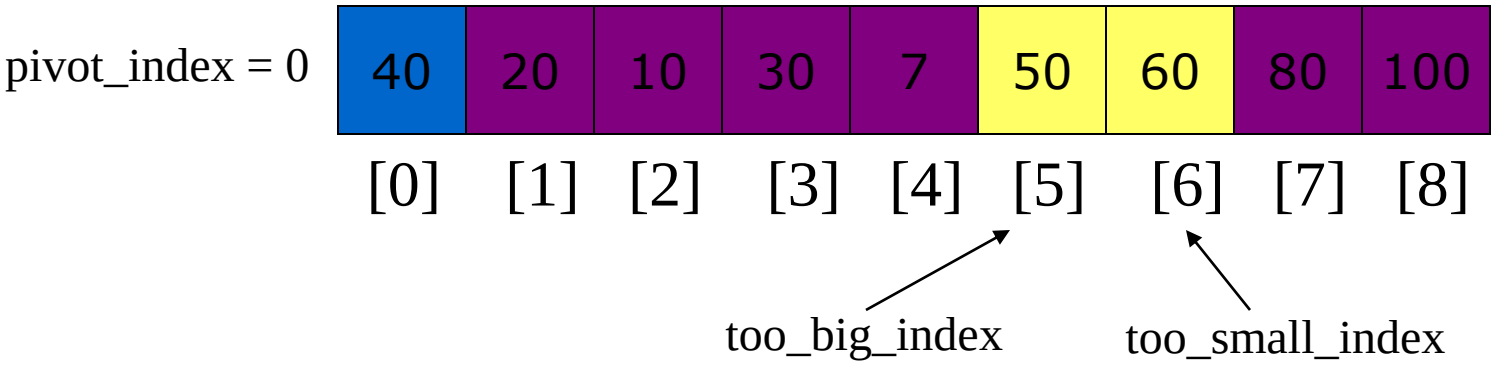
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



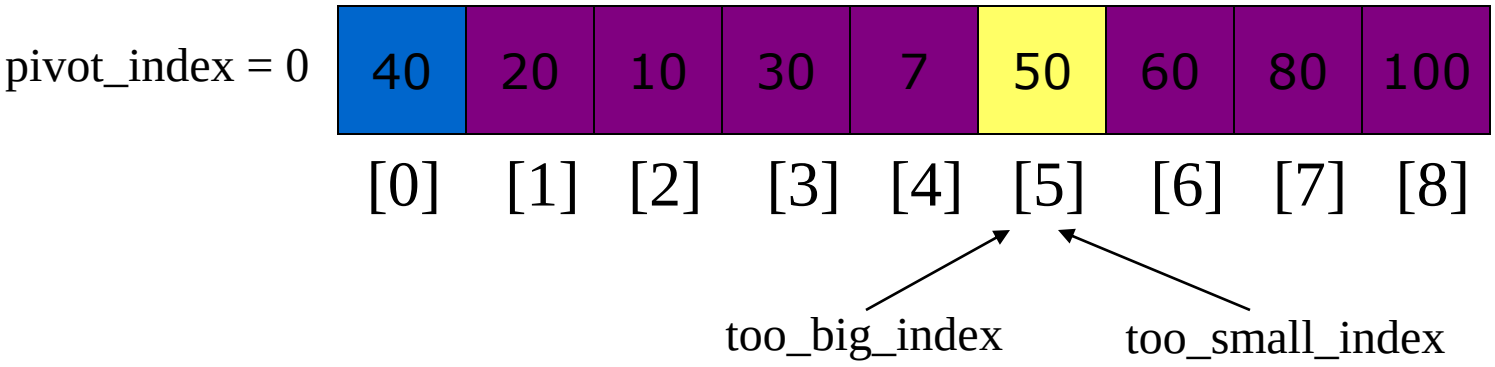
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



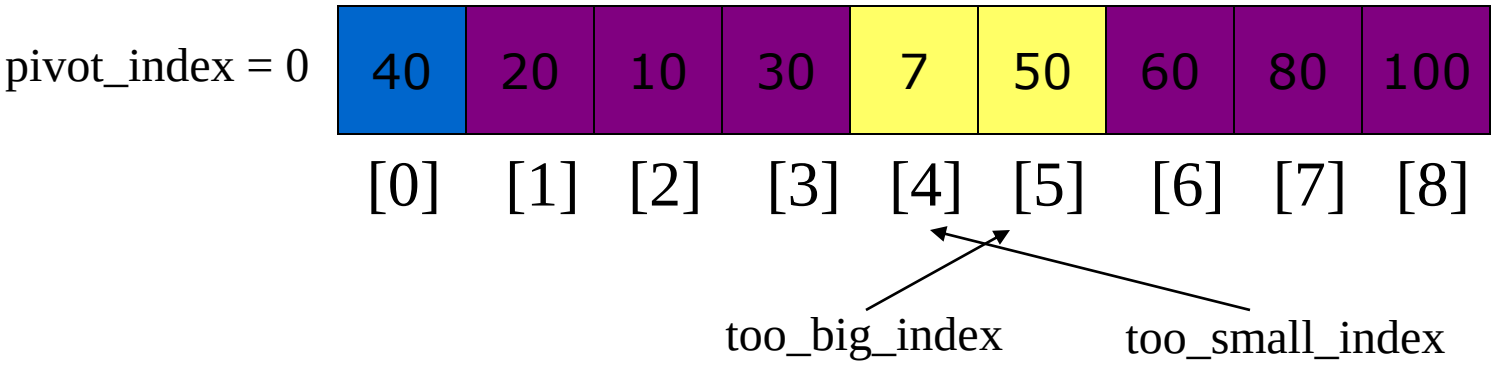
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



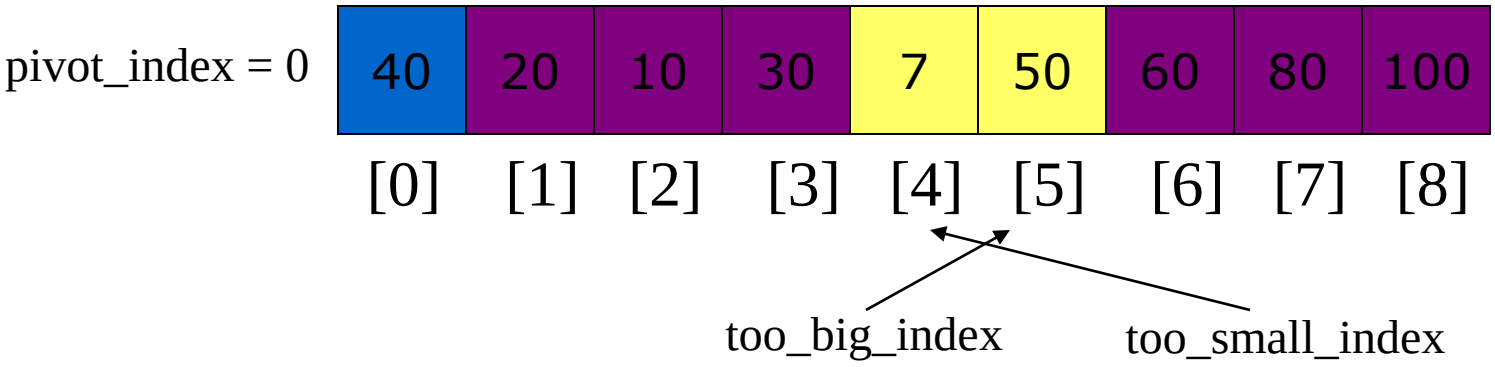
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



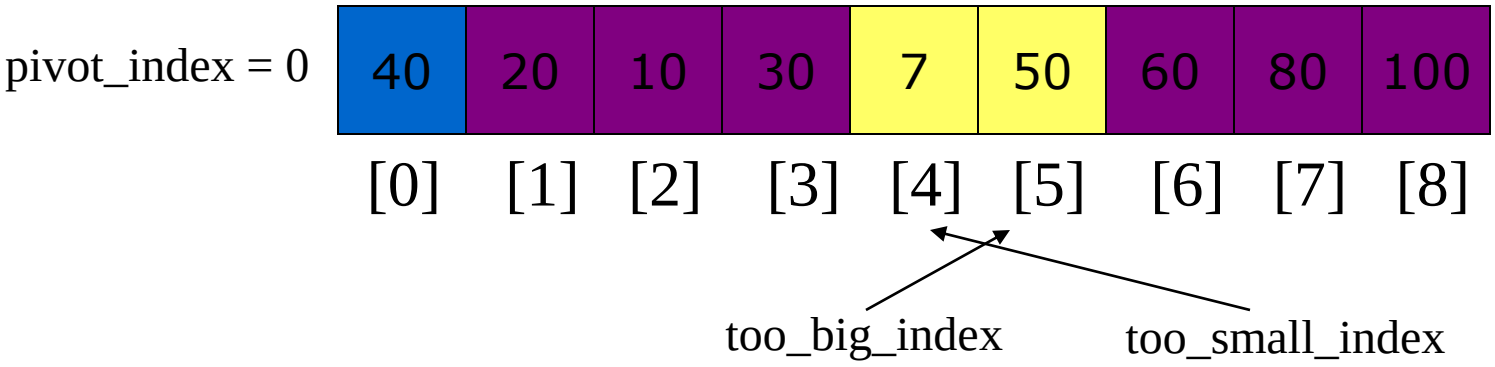
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



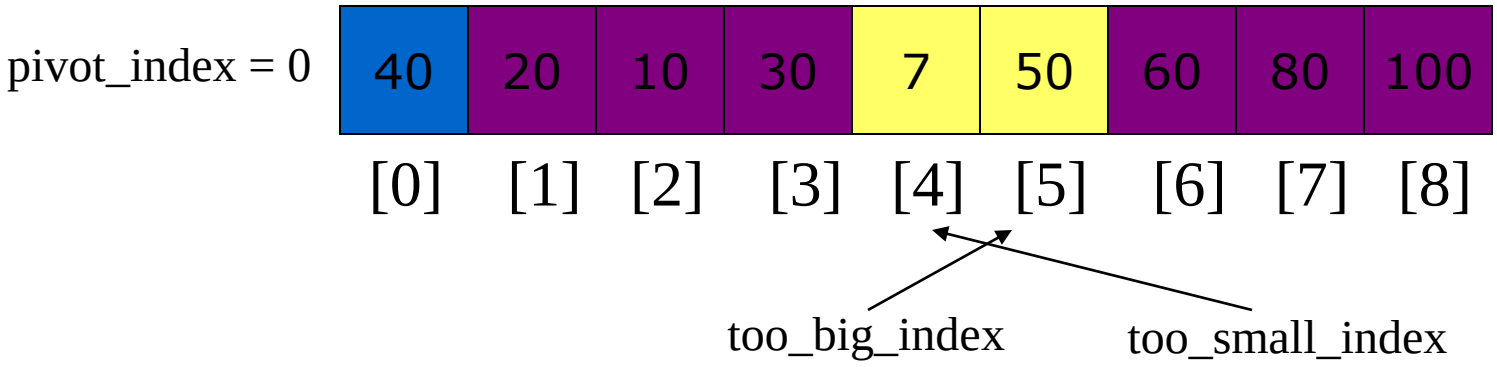
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



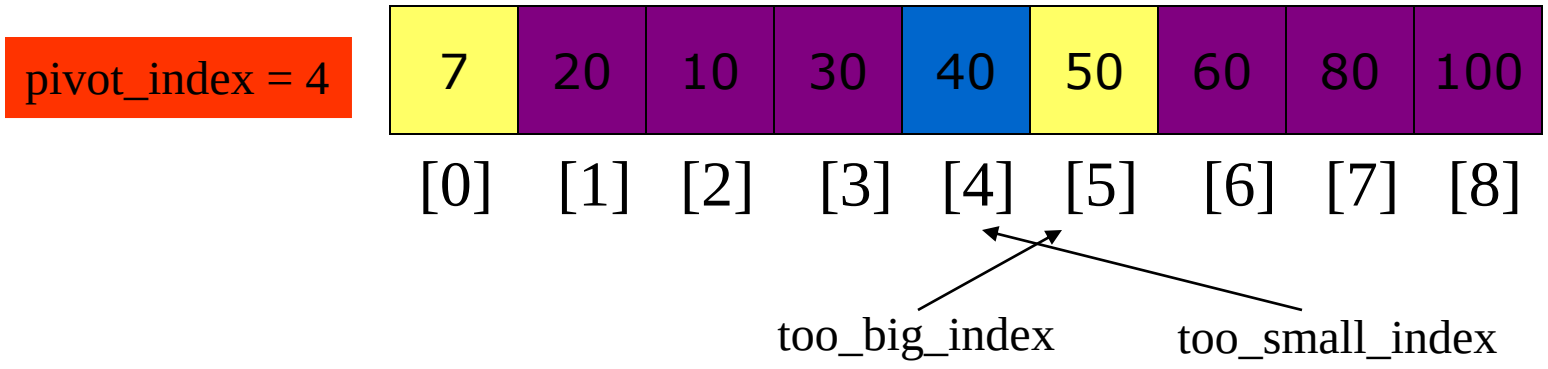
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



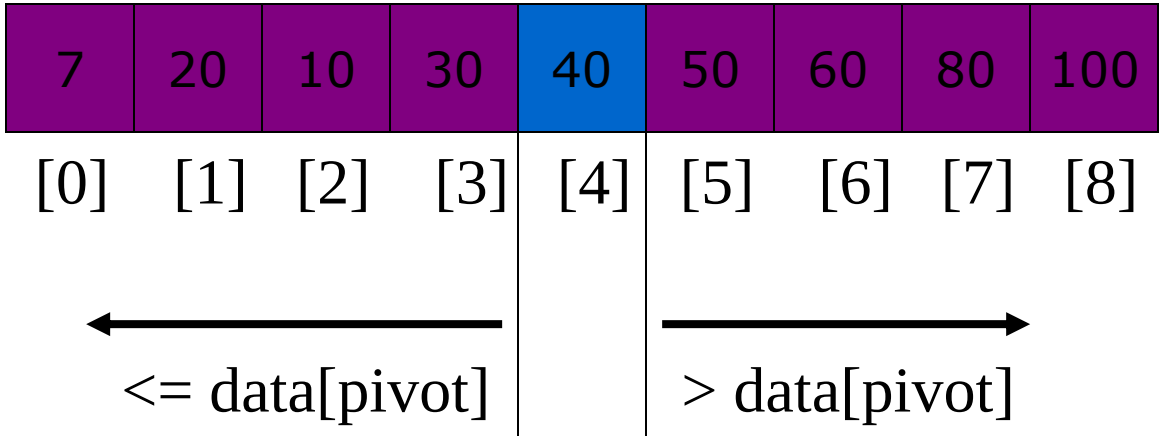
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



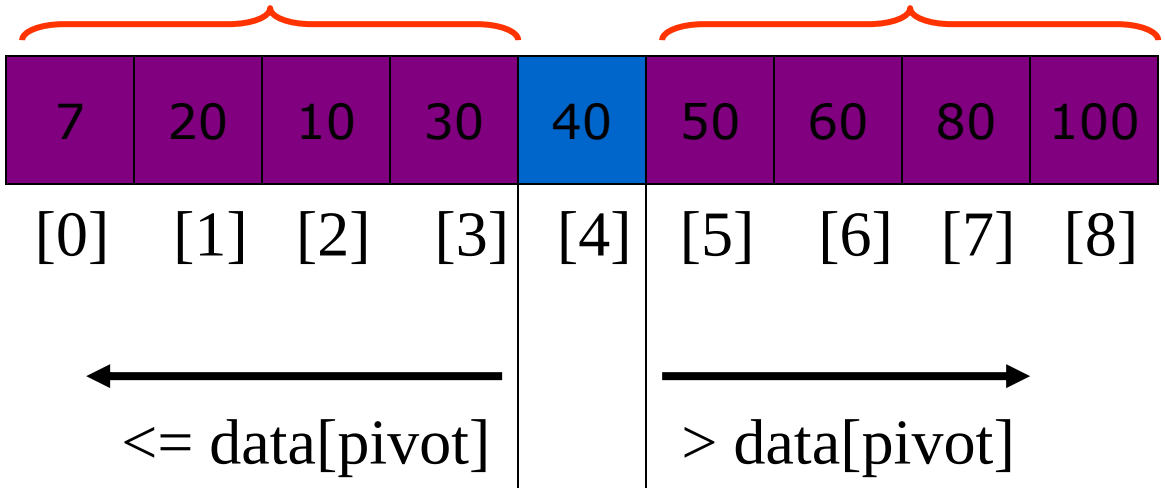
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



Partition Result



Recursion: Quicksort Sub-arrays



Quicksort Analysis

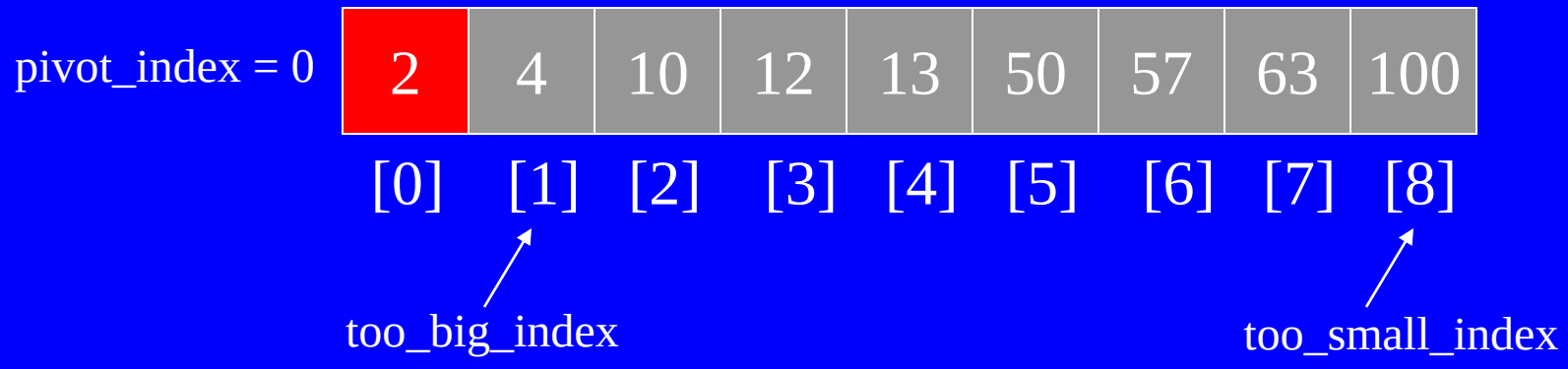
- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? O(n)

Quicksort Analysis

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

Quicksort: Worst Case

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



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: $O(n^2)$!!!