Mergesort and Quicksort

CS 206

Sorting algorithms

- Insertion, selection and bubble sort have quadratic worst-case performance
- Mergesort and Quicksort have time O(n lg n)

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

- 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

Example

- Partition into lists of size n/2

$$\begin{tikzcd}
{[10, 4, 6, 3, 8, 2, 5, 7]} \arrow[d] \arrow[drrrr] & & & & & & & \\
{[10, 4, 6, 3]} \arrow[d] \arrow[drr] & & & & {[8, 2, 5, 7]} \arrow[d] \arrow[drr] & & & \\
{[10, 4]} \arrow[d] \arrow[dr] & & {[6, 3]} \arrow[d] \arrow[dr] & & {[8, 2]} \arrow[d] \arrow[dr] & & {[5, 7]} \arrow[d] \arrow[dr] & \\
{[4]} & {[10]} & {[3]} & {[6]} & {[2]} & {[8]} & {[5]} & {[7]}
\end{tikzcd}$$

Example Cont' d

- Merge

$$\begin{tikzcd}
 & & & {[2, 3, 4, 5, 6, 7, 8, 10]} & & & & \\
 & {[3, 4, 6, 10]} \arrow[urr] & & & & {[2, 5, 7, 8]} \arrow[ull] & & \\
{[4, 10]} \arrow[ur] & & {[3, 6]} \arrow[ul] & & {[2, 8]} \arrow[ur] & & {[5, 7]} \arrow[ul] & \\
{[4]} \arrow[u] & {[10]} \arrow[ul] & {[3]} \arrow[u] & {[6]} \arrow[ul] & {[2]} \arrow[u] & {[8]} \arrow[ul] & {[5]} \arrow[u] & {[7]} \arrow[ul]
\end{tikzcd}$$

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
  }
}
```

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:

$T(n) = 2T(n/2) + c_2n$

$T(n/2) = 2T(n/4) + c_2n/2$

$T(n) = 4T(n/4) + 2\ c_2n$

$T(n) = 8T(n/8) + 3\ c_2n$

$T(n) = 2^iT(n/2^i) + ic_2n$

Assuming $n = 2^k$, expansion halts when we get $T(1)$ on right side; this happens when $i=k$ $T(n) = 2^kT(1) + kc_2n$

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

$T(n) = c_1n + c_2\ n \lg n$;

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

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

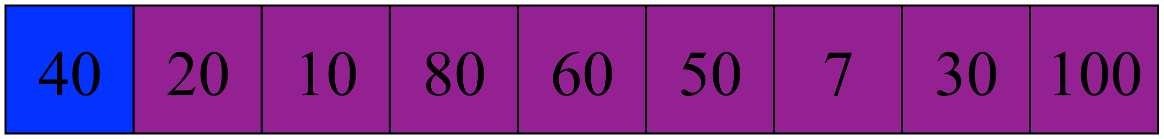
Example

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:



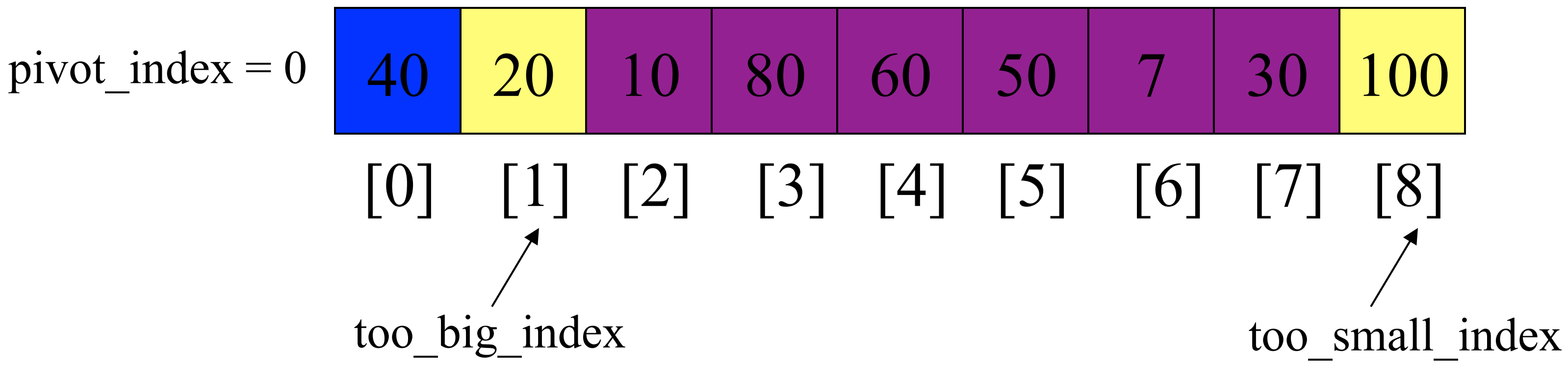
Partitioning Array

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

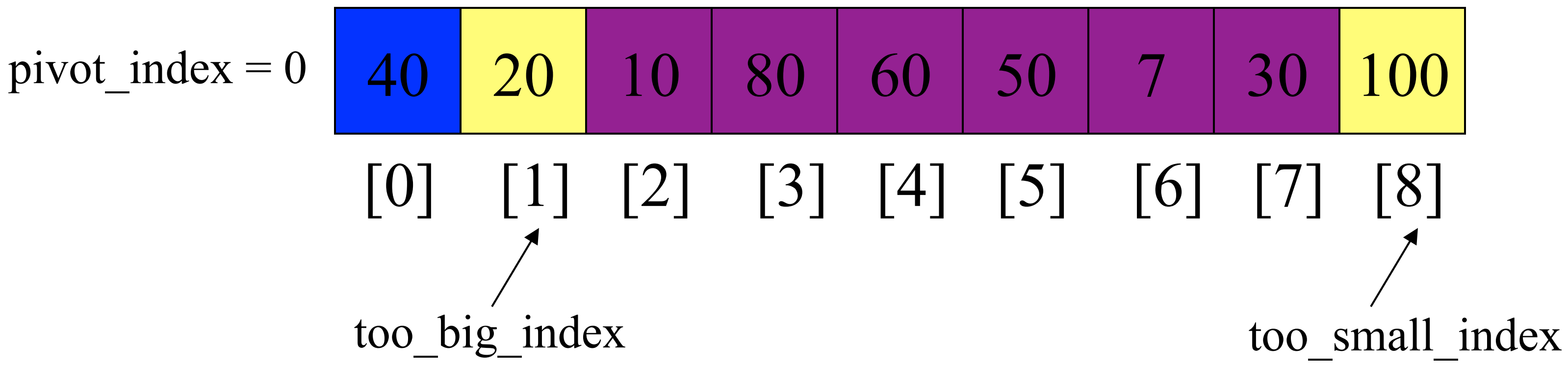
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

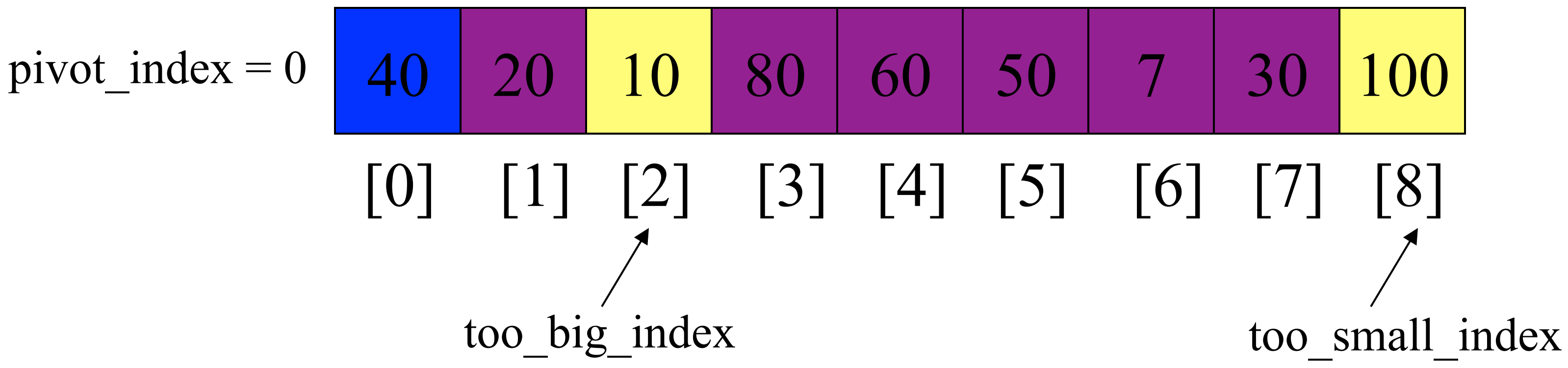
Partitioning loops through, swapping elements below/above pivot.


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

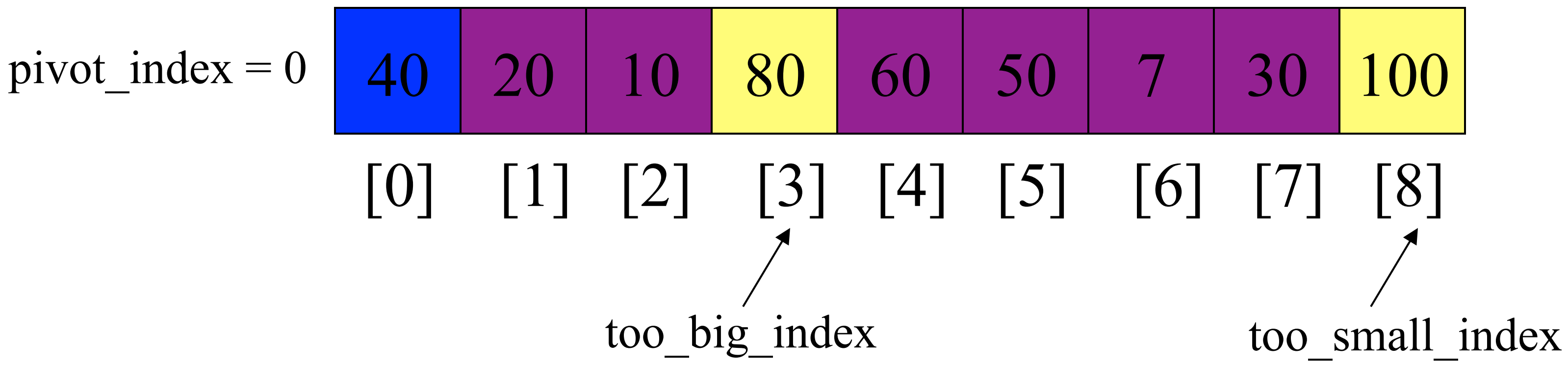
```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
```



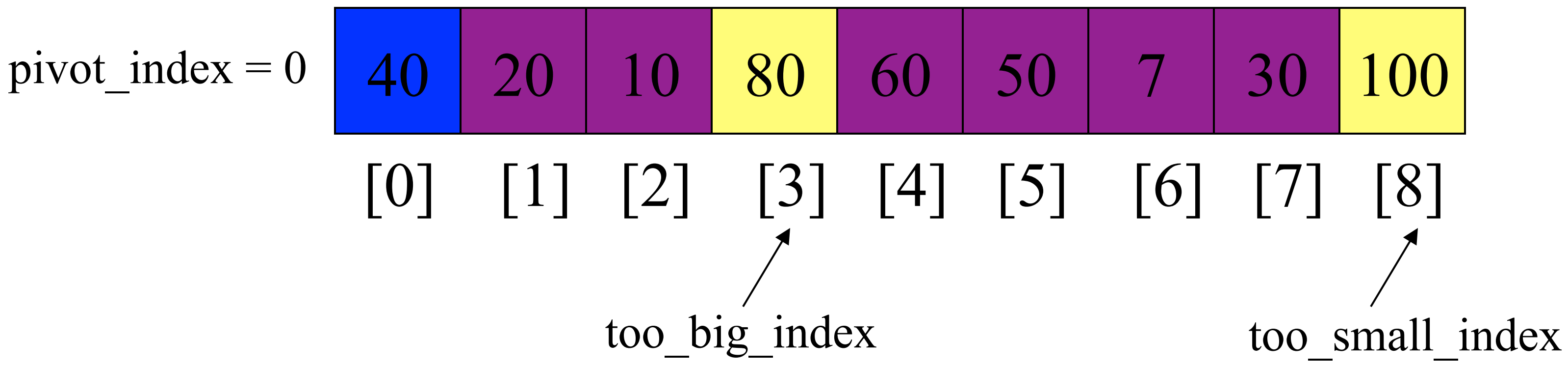
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
```



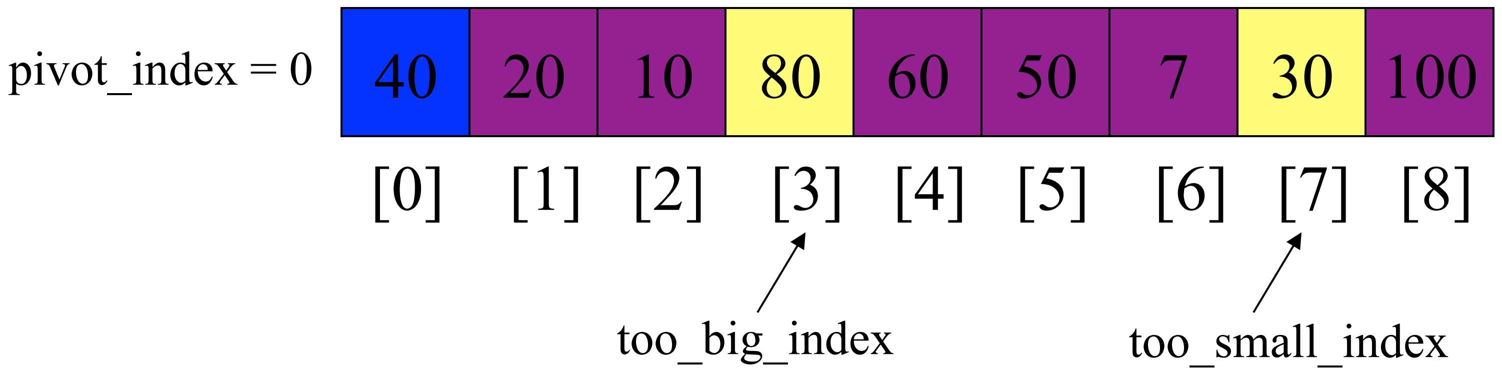
```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
```



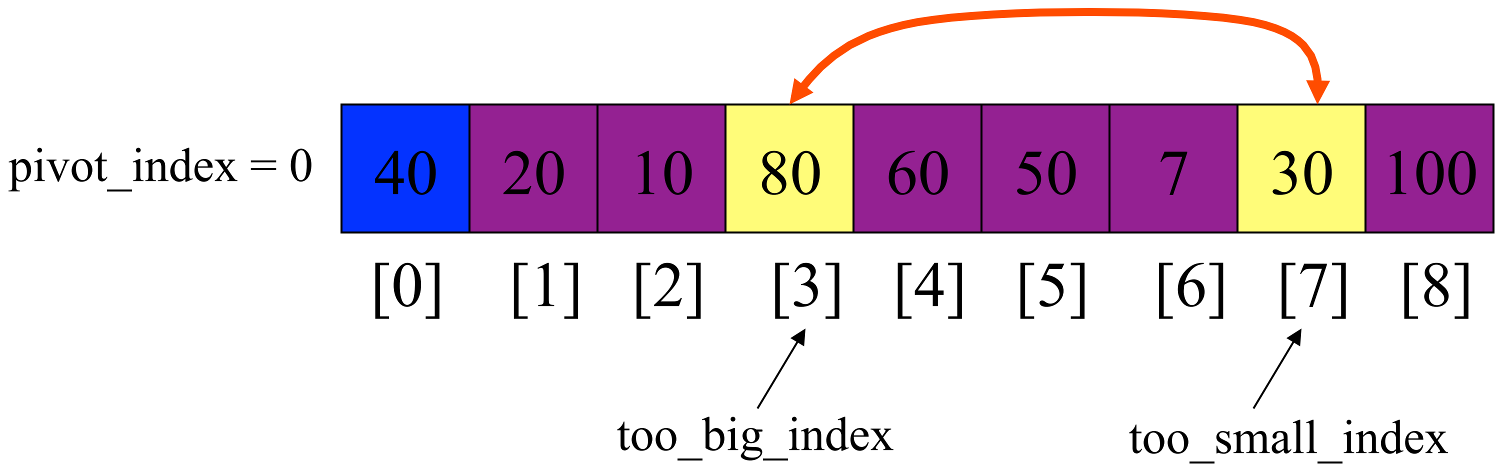
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
```



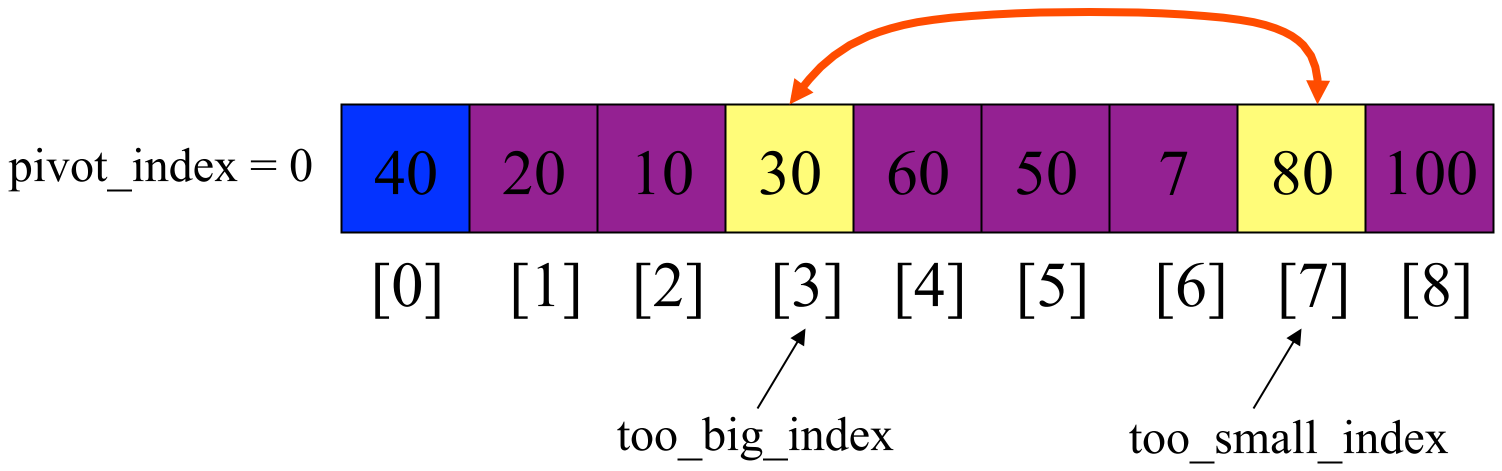
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
```



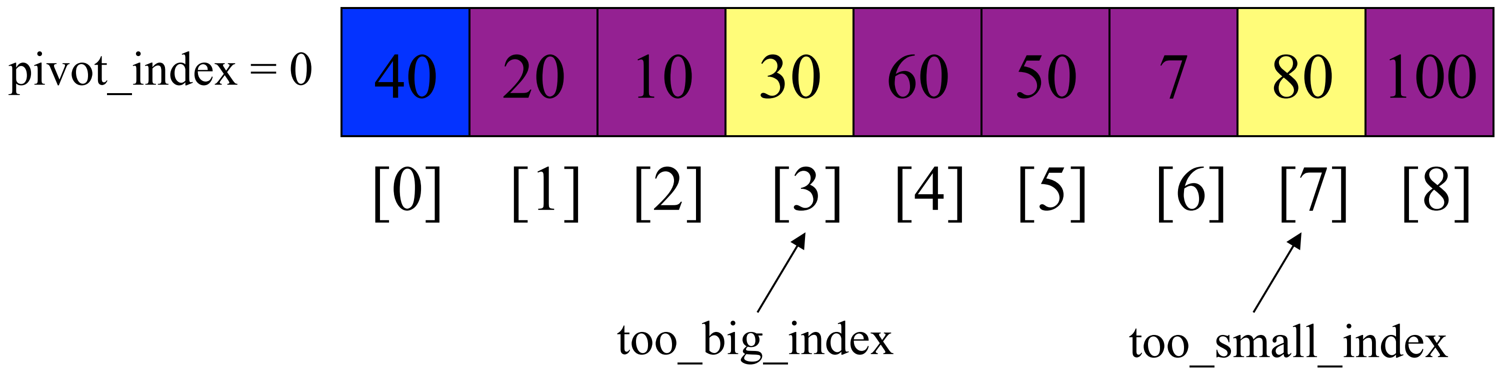
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
```



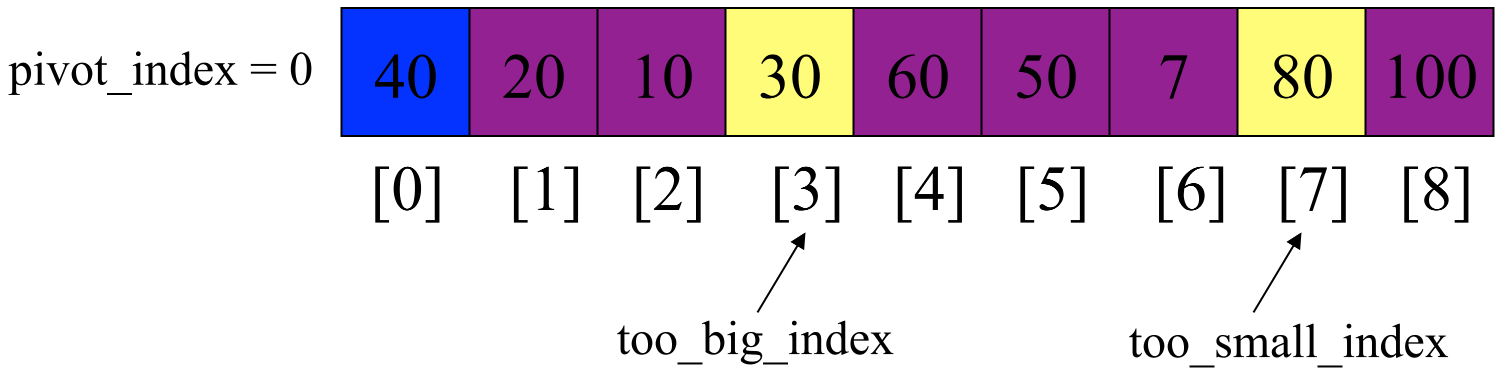
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
```



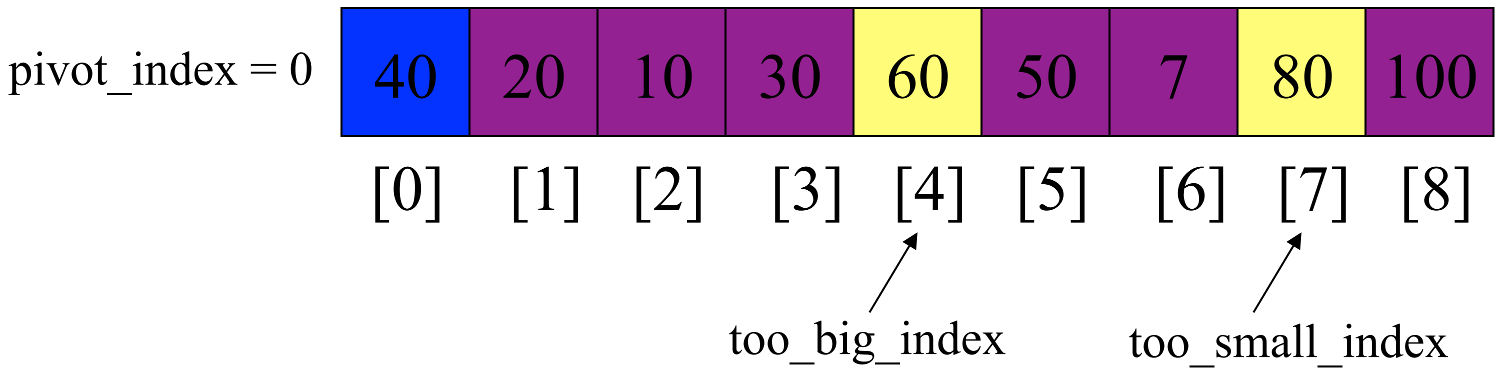
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



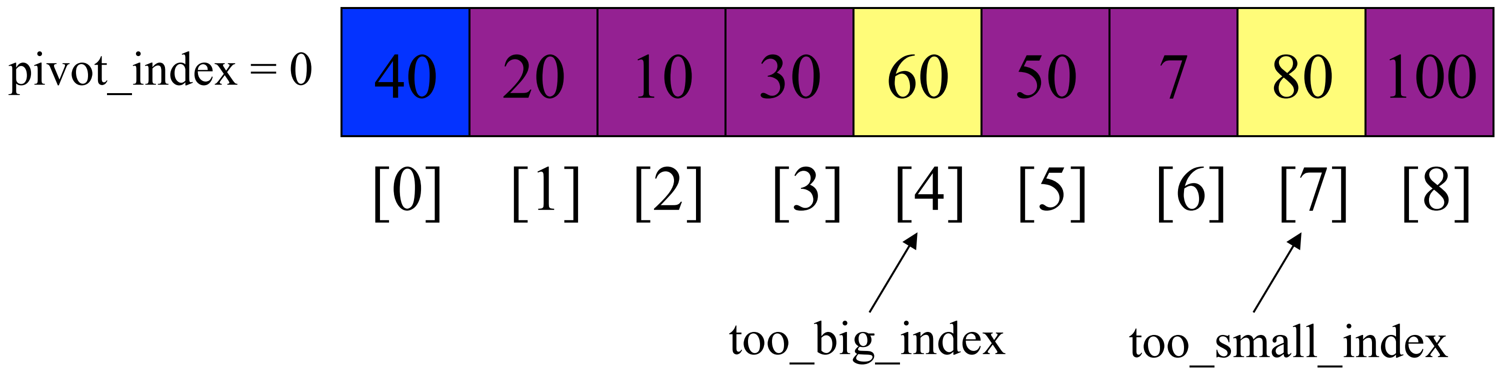
```
→ 1. While data[too_big_index] <= data[pivot]
          ++too_big_index
  2. While data[too_small_index] > data[pivot]
          --too_small_index
  3. If too_big_index < too_small_index
          swap data[too_big_index] and data[too_small_index]
  4. If too_small_index > too_big_index, go to 1.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



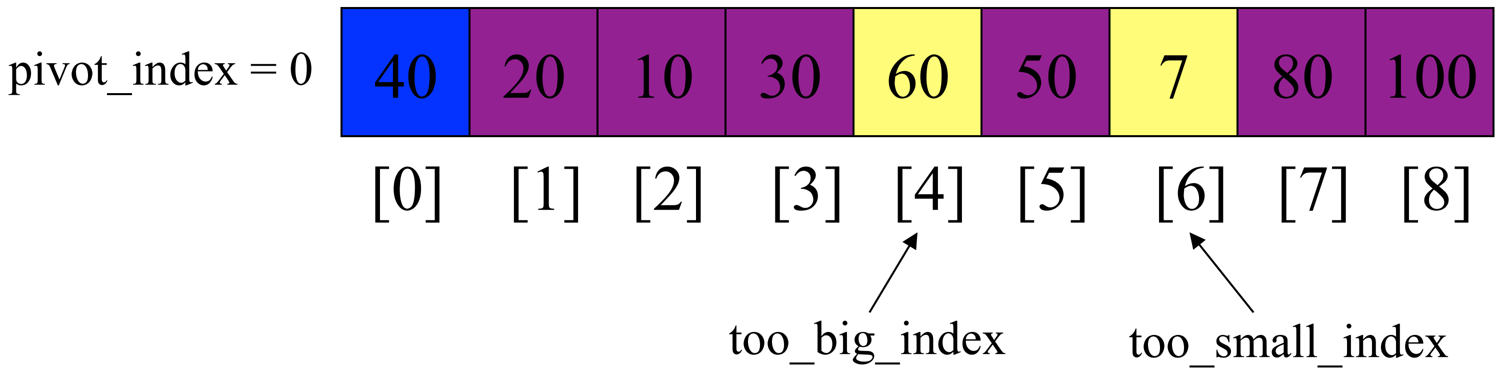
```
→ 1. While data[too_big_index] <= data[pivot]
         ++too_big_index
  2. While data[too_small_index] > data[pivot]
         --too_small_index
  3. If too_big_index < too_small_index
         swap data[too_big_index] and data[too_small_index]
  4. If too_small_index > too_big_index, go to 1.
```



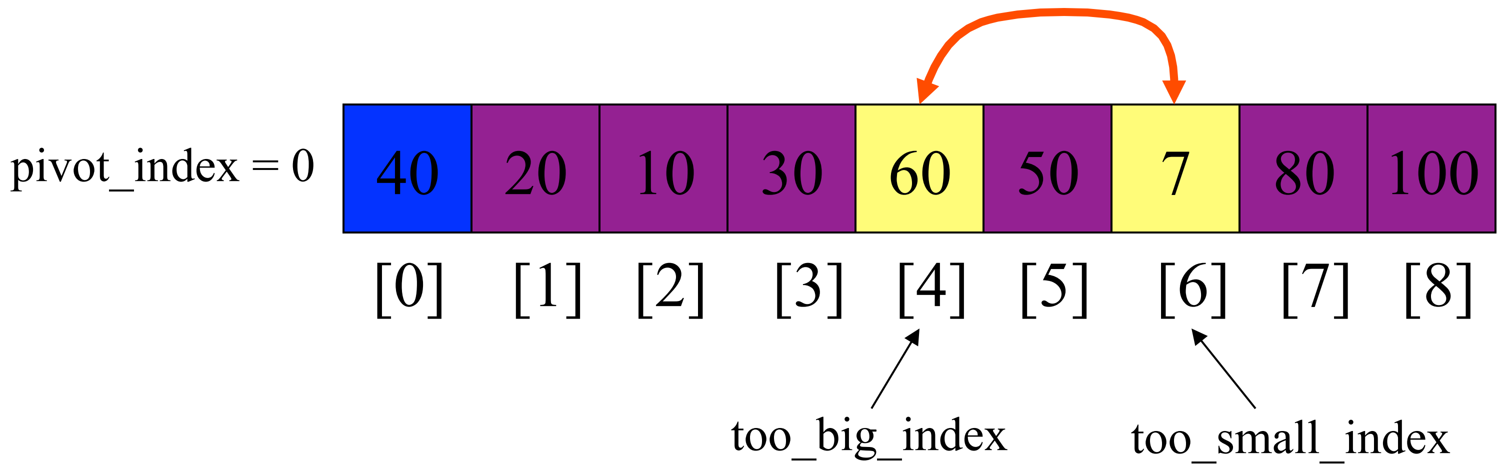
```
   1. While data[too_big_index] <= data[pivot]
          ++too_big_index
→ 2. While data[too_small_index] > data[pivot]
          --too_small_index
   3. If too_big_index < too_small_index
          swap data[too_big_index] and data[too_small_index]
   4. If too_small_index > too_big_index, go to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



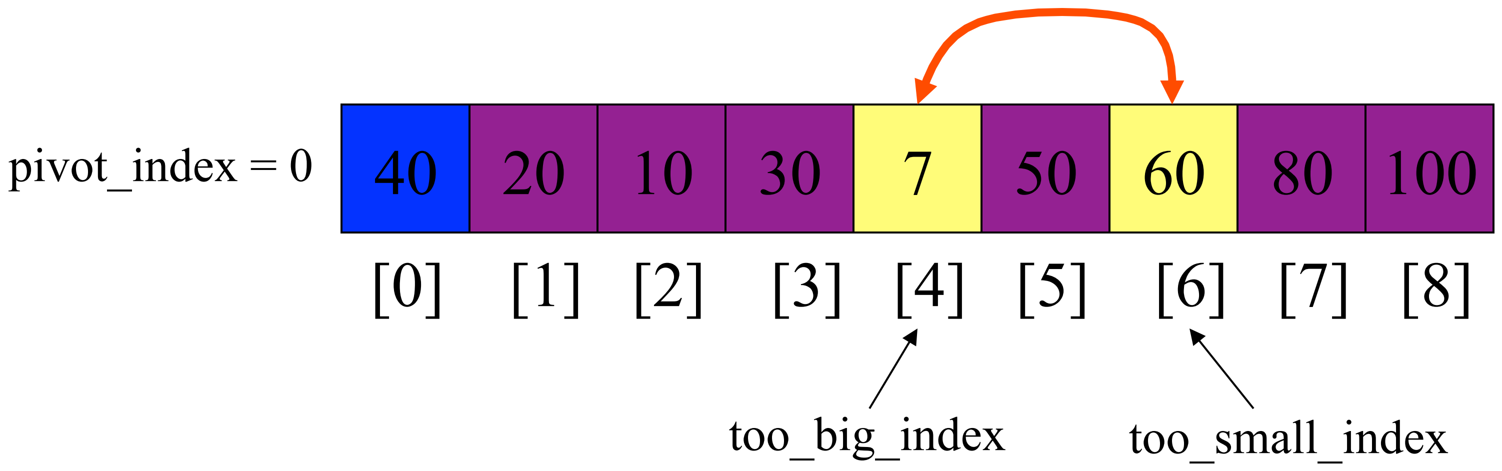
```
   1. While data[too_big_index] <= data[pivot]
          ++too_big_index
→  2. While data[too_small_index] > data[pivot]
          --too_small_index
   3. If too_big_index < too_small_index
          swap data[too_big_index] and data[too_small_index]
   4. If too_small_index > too_big_index, go to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



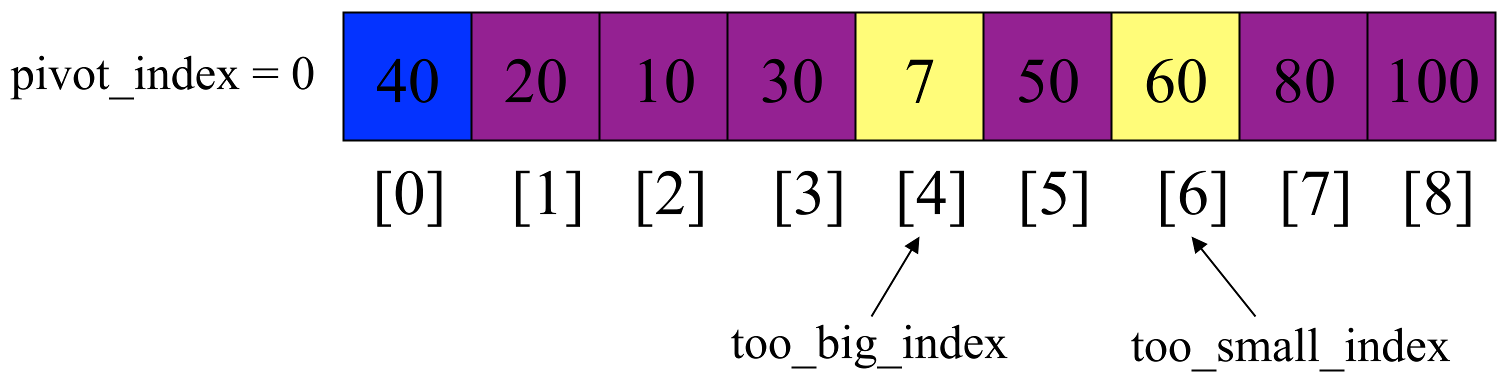
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



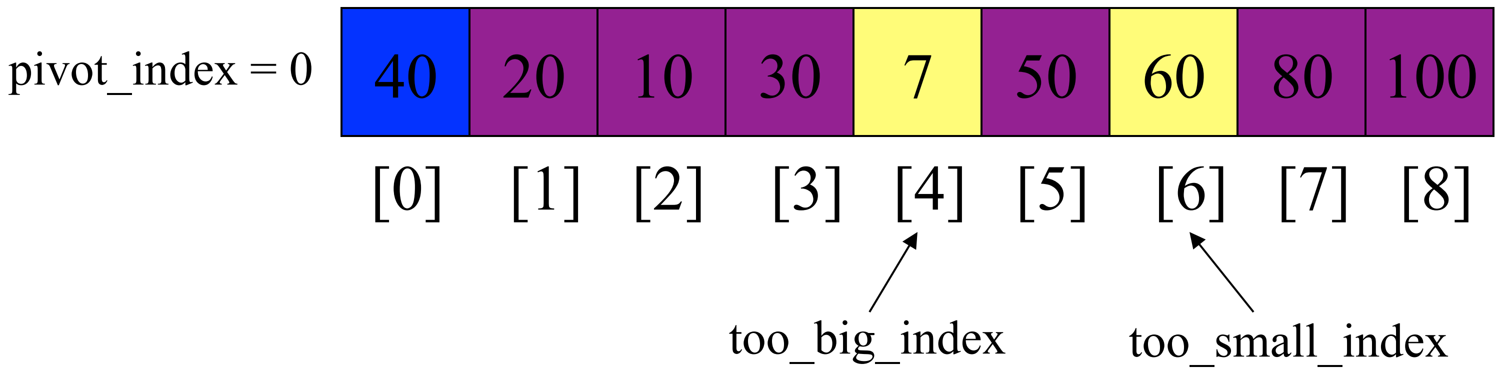
```
→ 1. While data[too_big_index] <= data[pivot]
        ++too_big_index
  2. While data[too_small_index] > data[pivot]
        --too_small_index
→ 3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
  4. If too_small_index > too_big_index, go to 1.
```



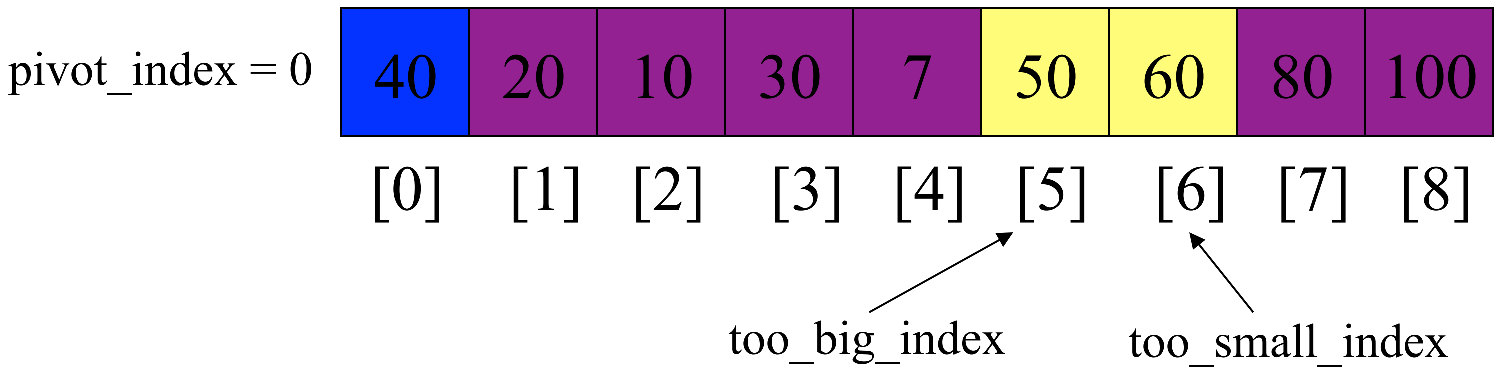
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



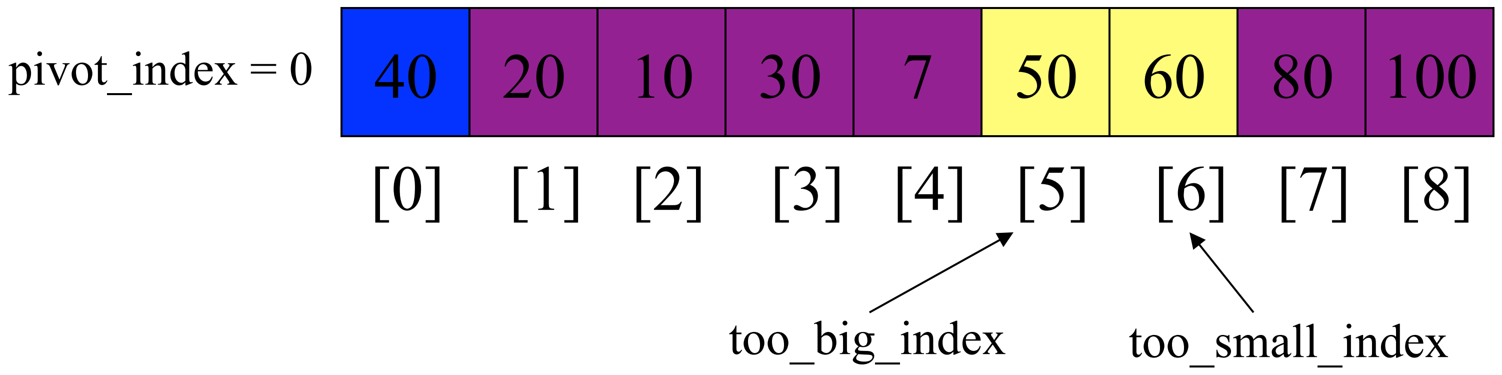
```
→ 1. While data[too_big_index] <= data[pivot]
            ++too_big_index
  2. While data[too_small_index] > data[pivot]
            --too_small_index
  3. If too_big_index < too_small_index
            swap data[too_big_index] and data[too_small_index]
  4. If too_small_index > too_big_index, go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



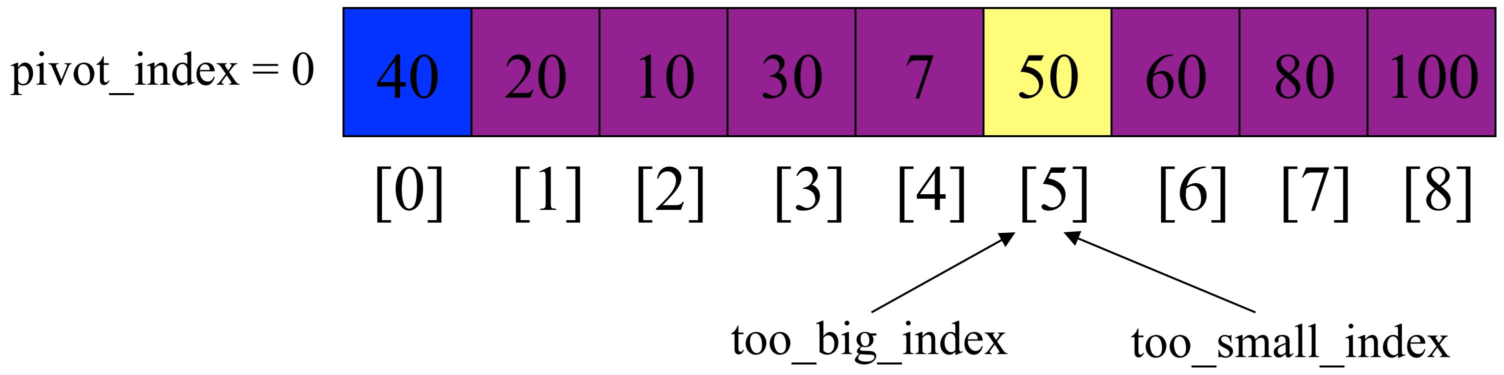
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



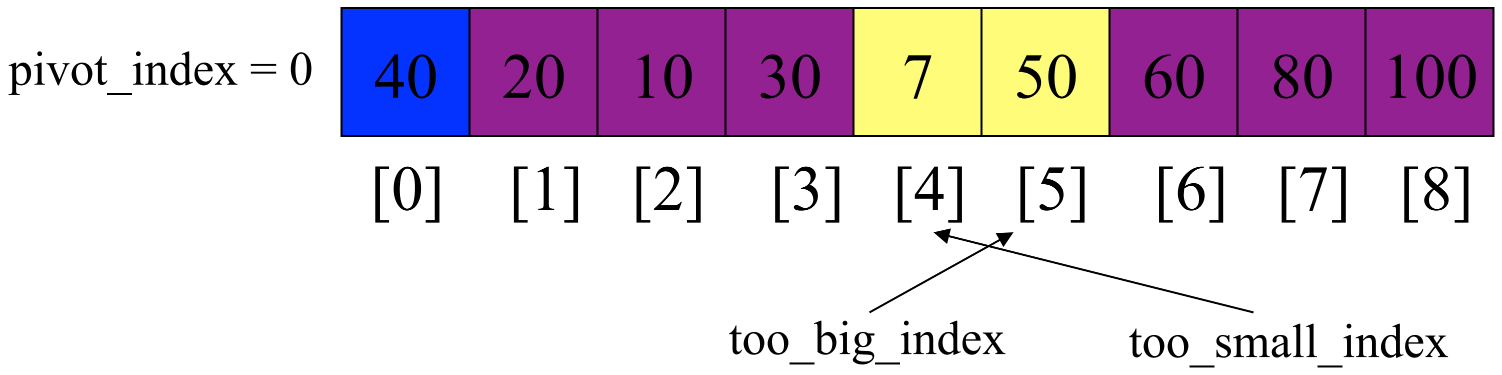
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



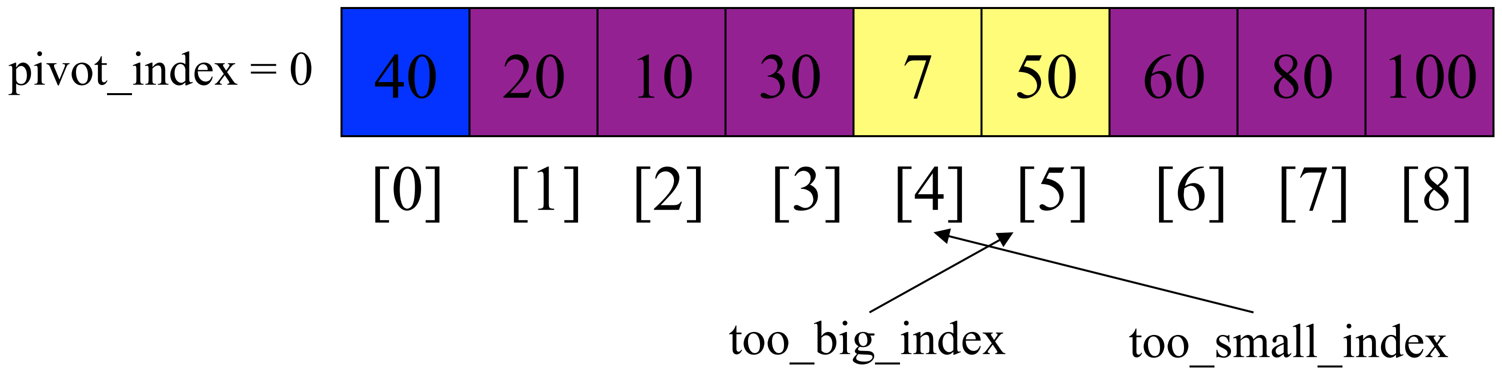
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



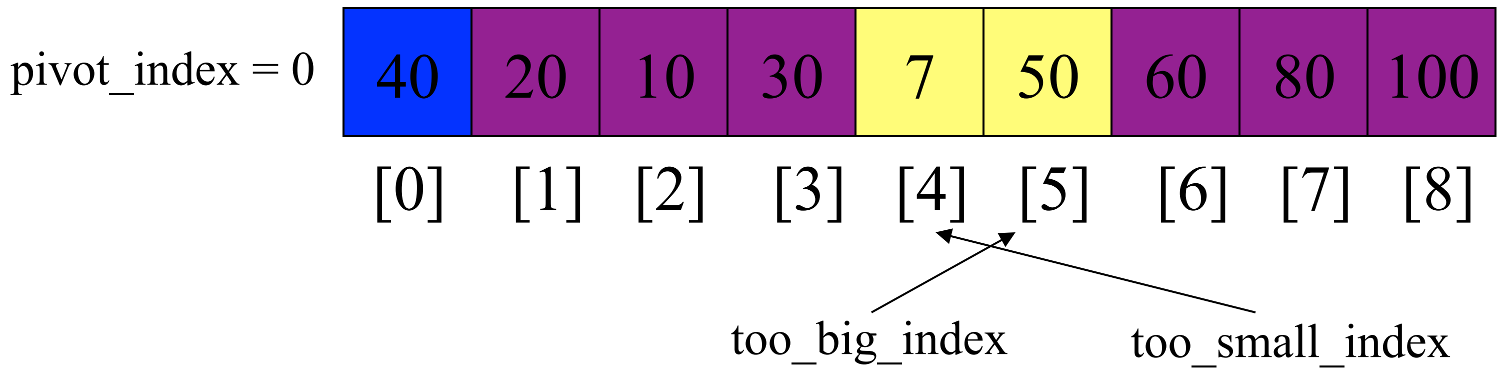
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



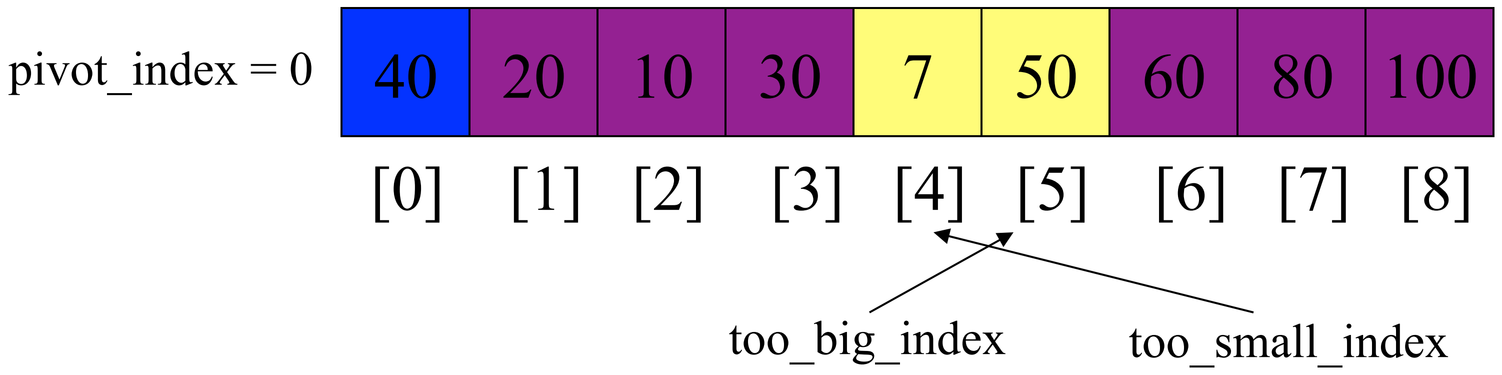
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
```



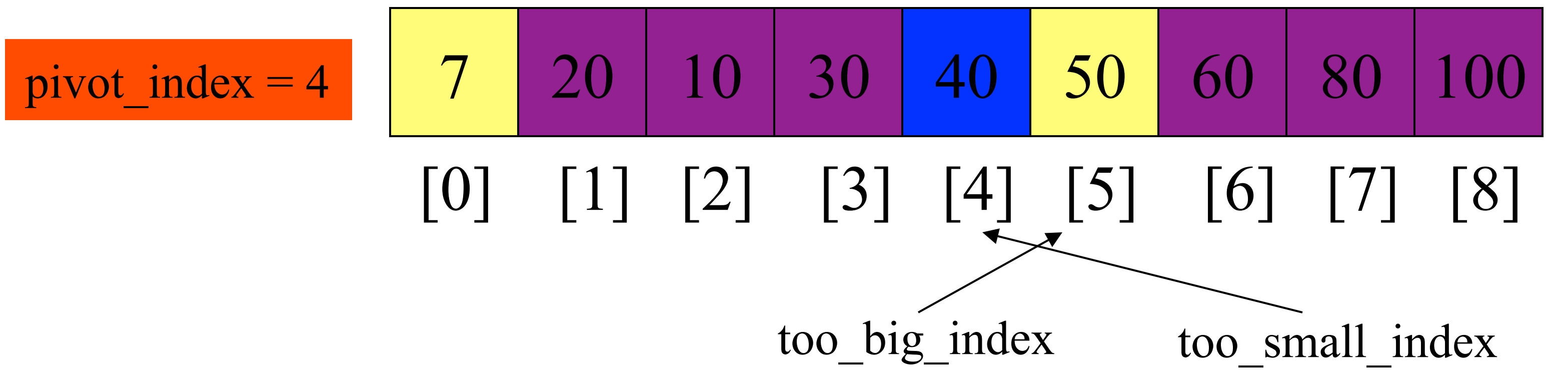
```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  If too_small_index > too_big_index, go to 1.
```



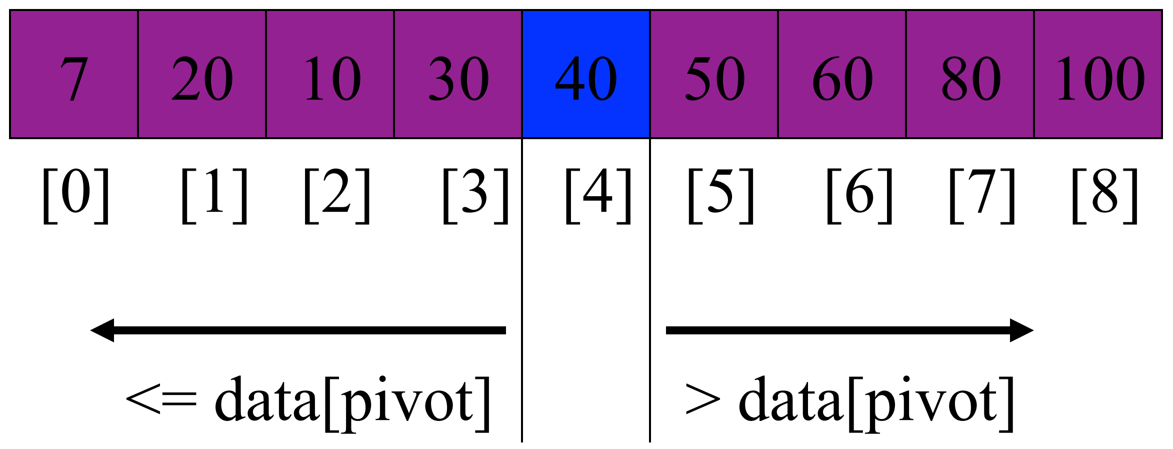
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
→ 5. Swap data[too_small_index]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



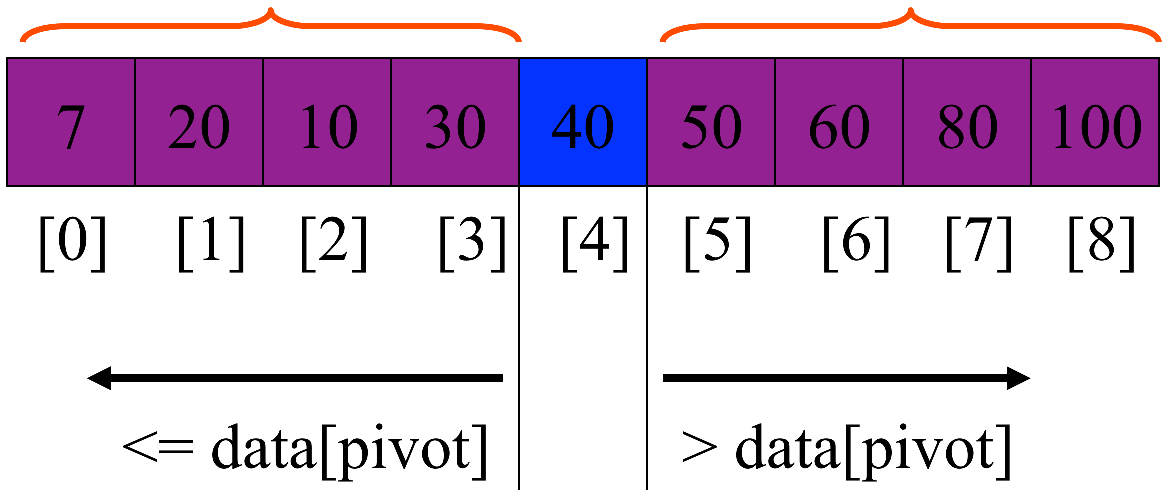
```
   1. While data[too_big_index] <= data[pivot]
          ++too_big_index
   2. While data[too_small_index] > data[pivot]
          --too_small_index
   3. If too_big_index < too_small_index
          swap data[too_big_index] and data[too_small_index]
   4. If too_small_index > too_big_index, go to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

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(lg 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(lg 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(lg n)
  - Number of accesses in partition? O(n)

Quicksort Analysis

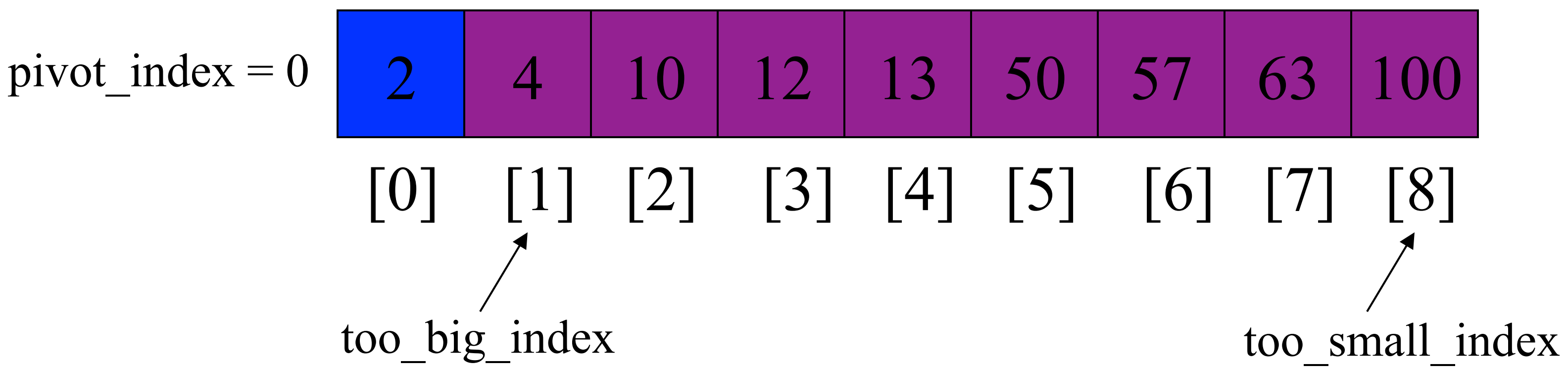
- Assume that keys are random, uniformly distributed.
- Best case running time: O(n lg n)

Quicksort Analysis

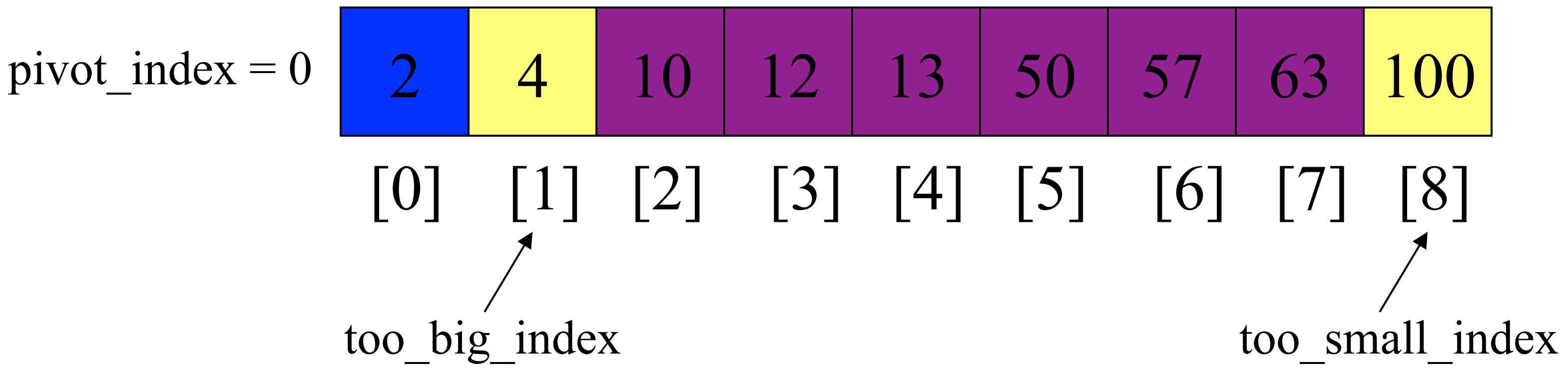
- Assume that keys are random, uniformly distributed.
- Best case running time: O(n lg n)
- Worst case running time?

Quicksort: Worst Case

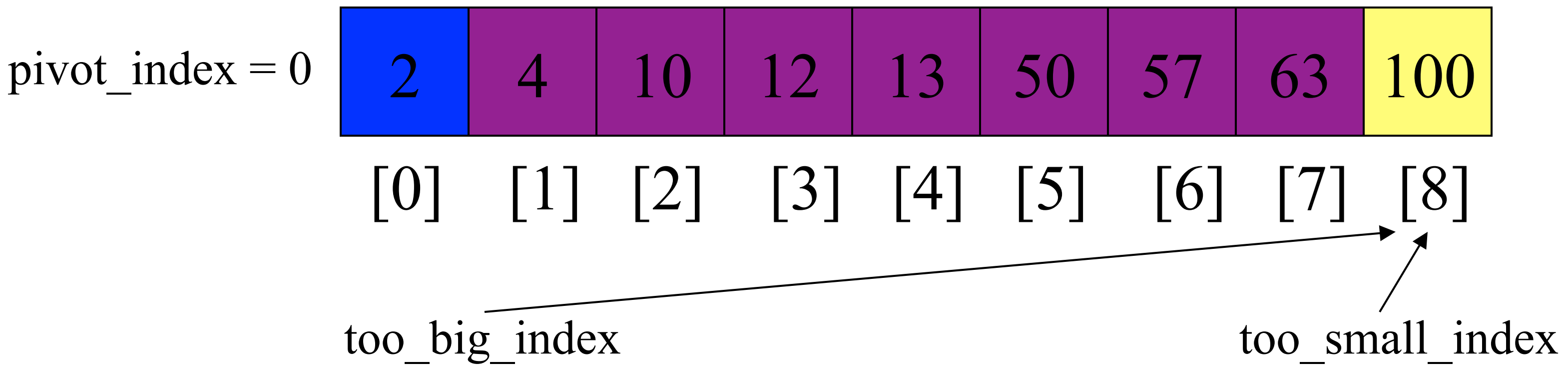
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



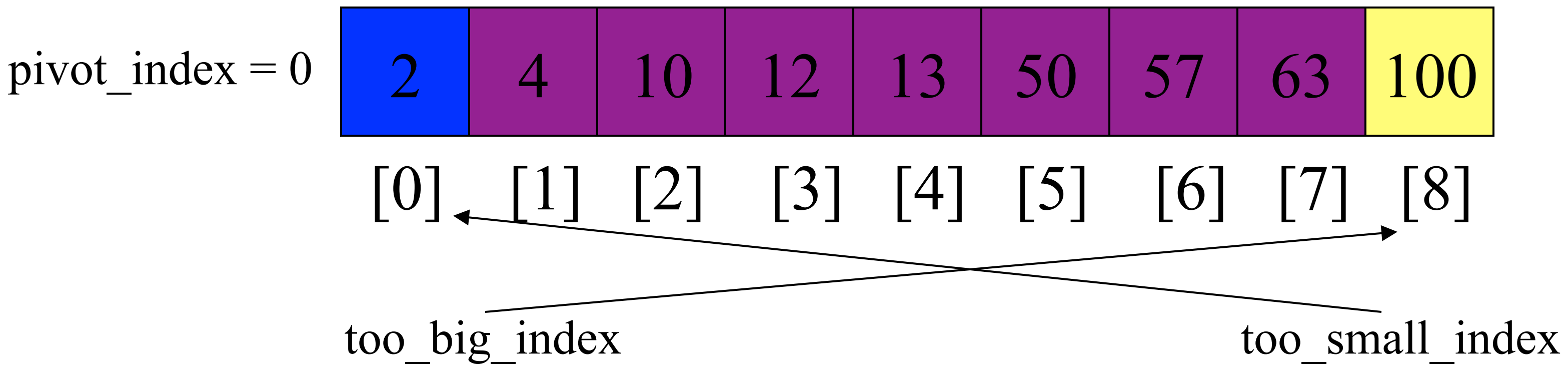
```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  If too_small_index > too_big_index, go to 1.
5.  Swap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



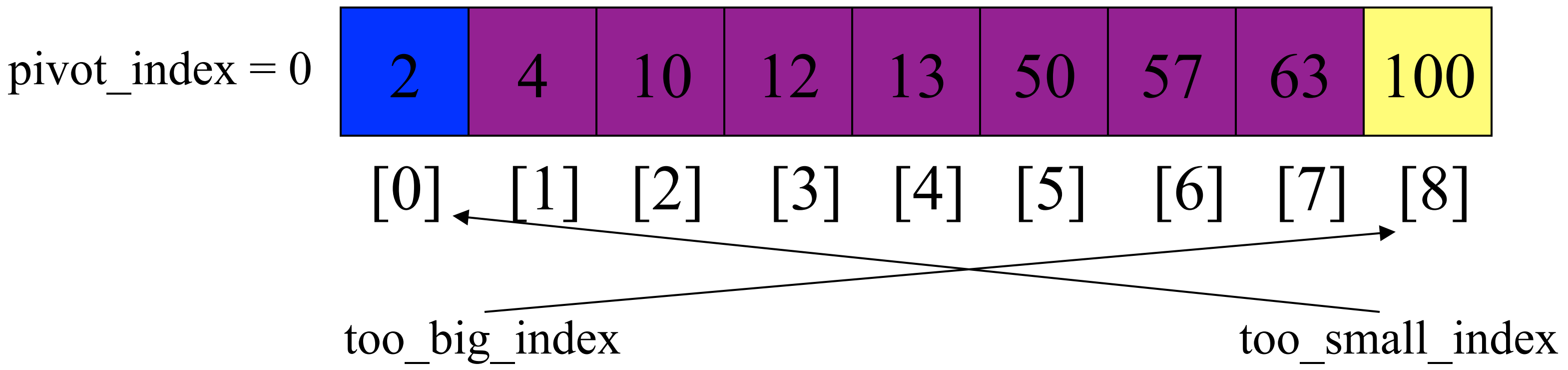
```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  If too_small_index > too_big_index, go to 1.
5.  Swap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



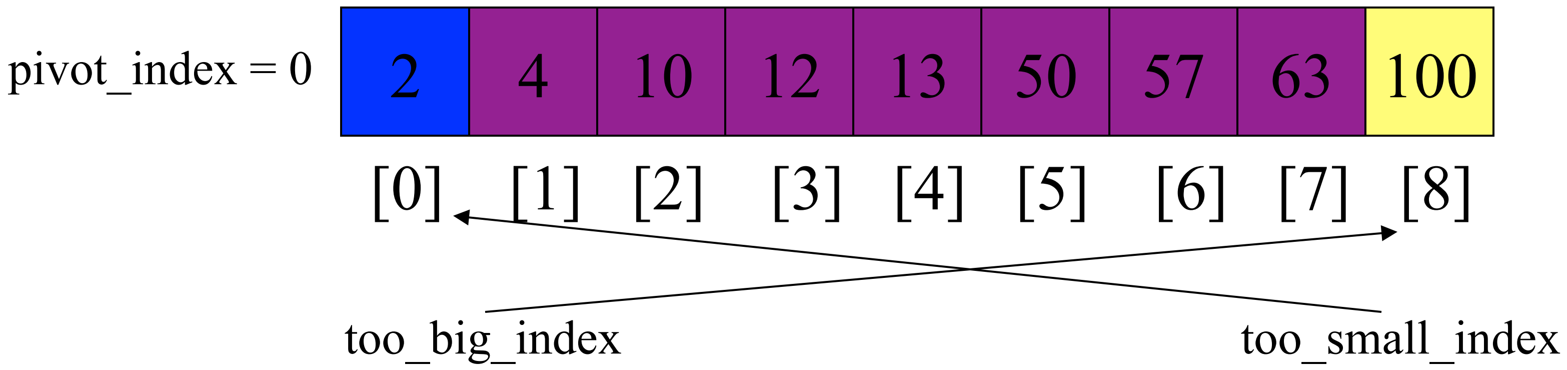
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



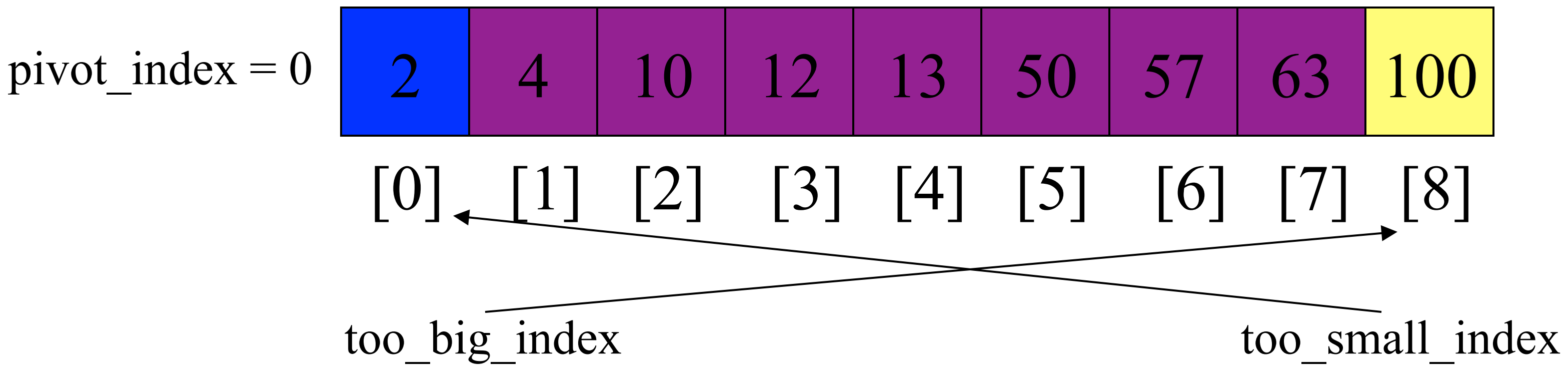
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



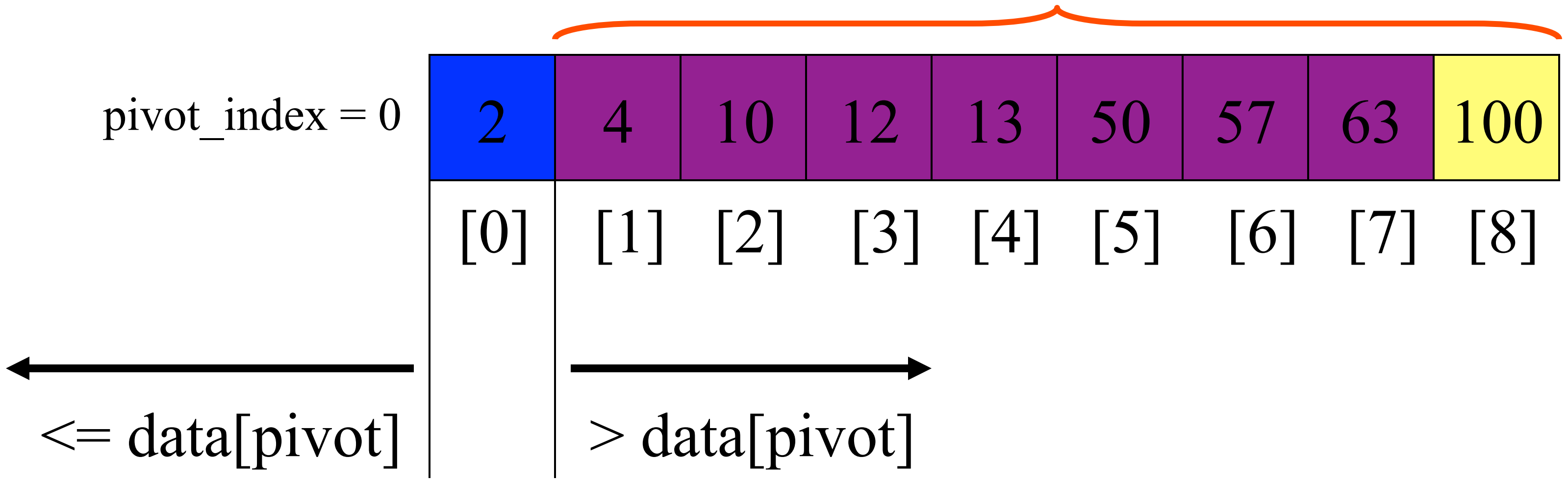
```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] 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. If 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.  If 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 lg 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 lg 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 lg 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 lg 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: $O(n \lg n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

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

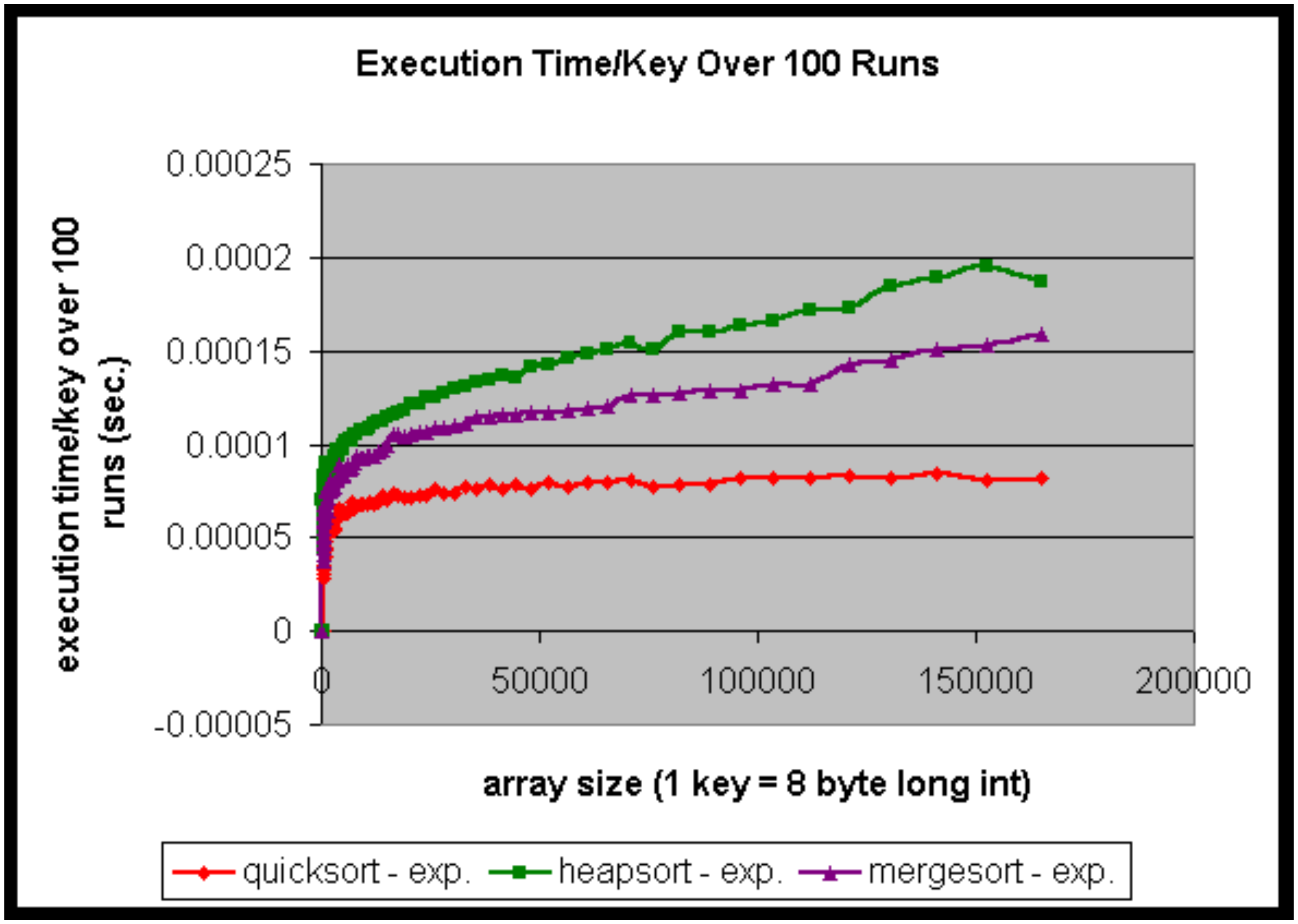
Use this median value as pivot.

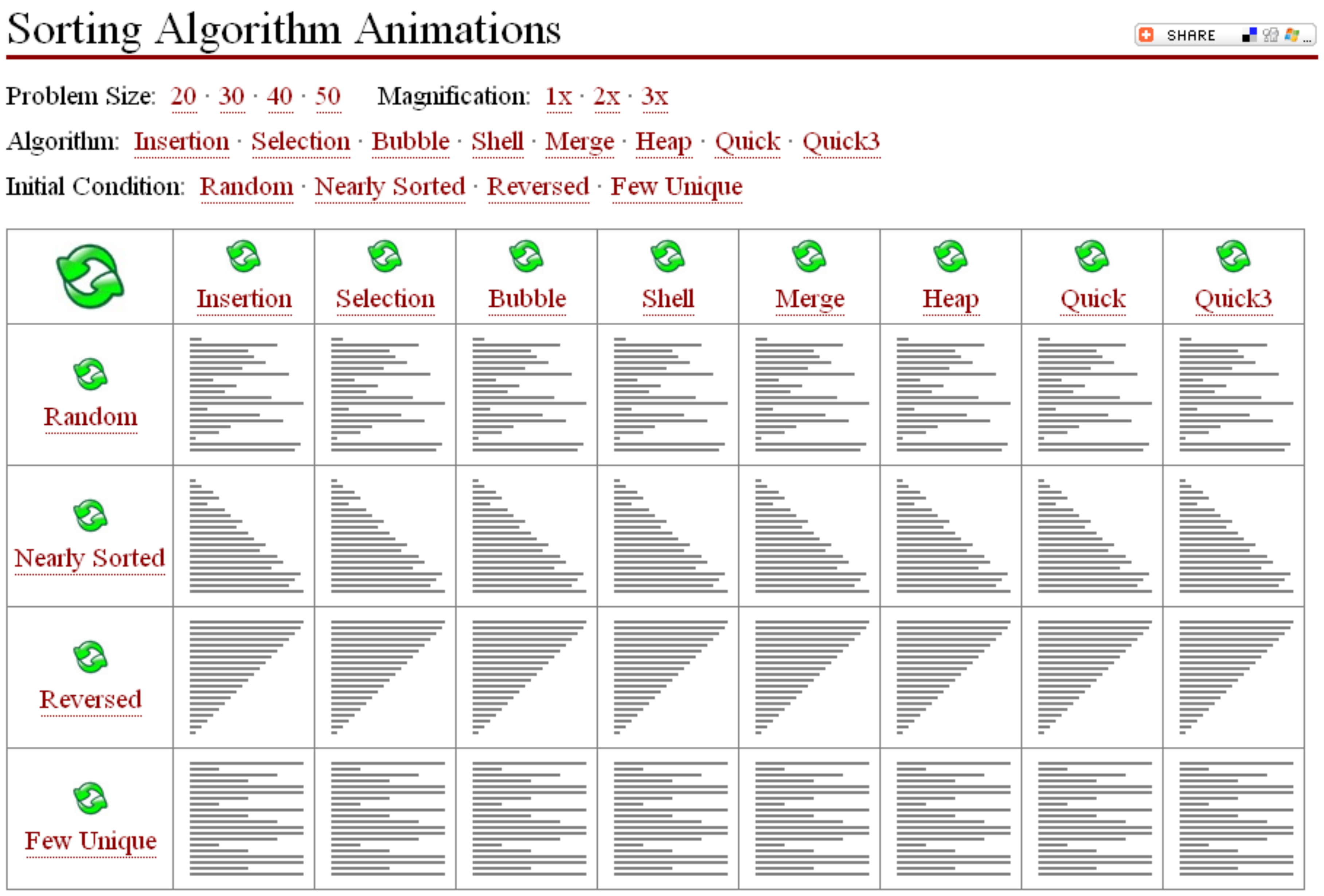
Randomize array initially

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.

Empirical Comparison





Sorting Algorithm Animations

Problem Size: 20 · 30 · 40 · 50 Magnification: 1x · 2x · 3x

Algorithm: Insertion · Selection · Bubble · Shell · Merge · Heap · Quick · Quick3

Initial Condition: Random · Nearly Sorted · Reversed · Few Unique

	Insertion	Selection	Bubble	Shell	Merge	Heap	Quick	Quick3
Random								
Nearly Sorted								
Reversed								
Few Unique								

http://www.sorting-algorithms.com/