Sorting Algorithms

1. Selection
2. Bubble
3. Insertion
4. Merge
5. Quick

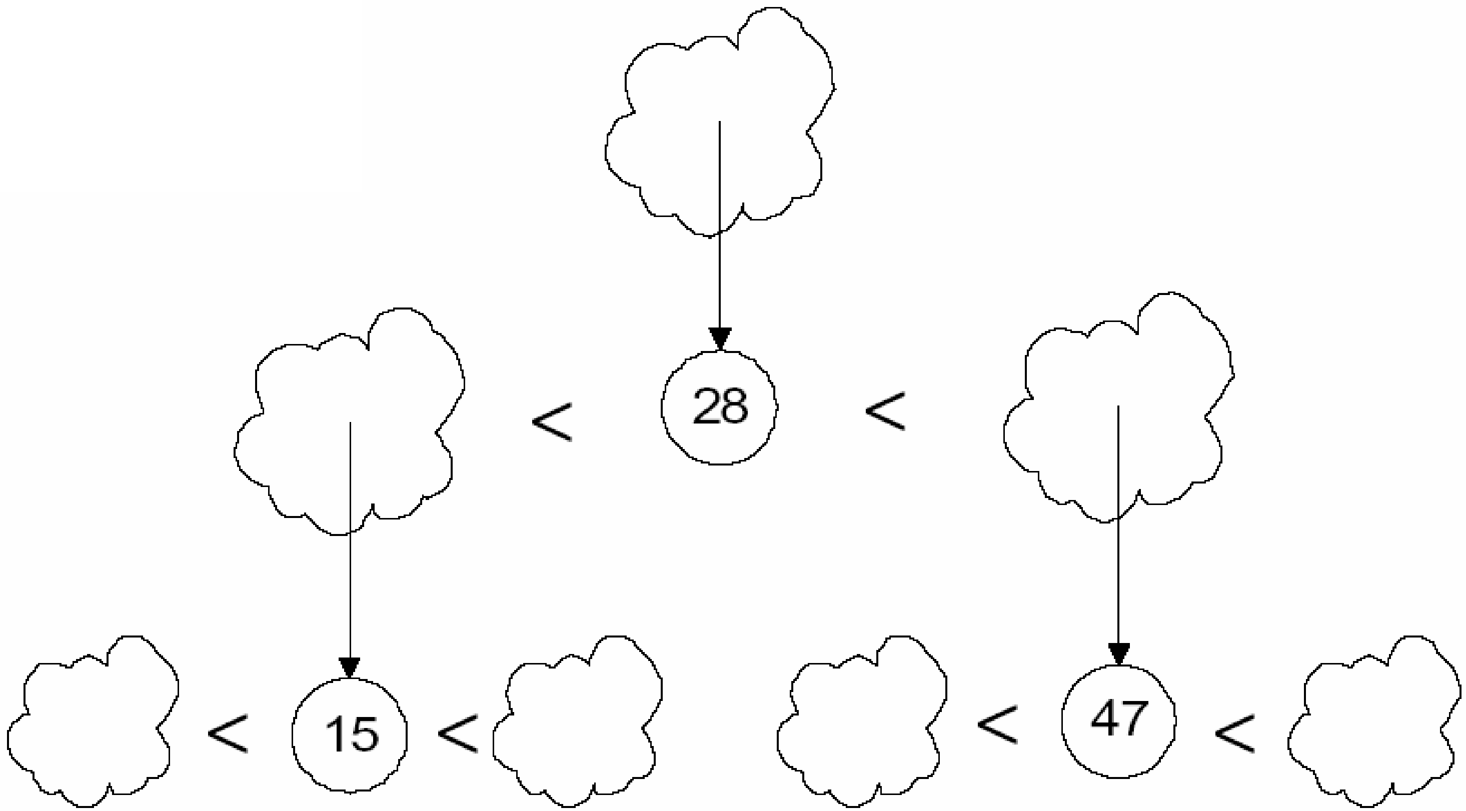
Sorting algorithms

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

$$O(n \log n)$$

- Mergesort and Quicksort

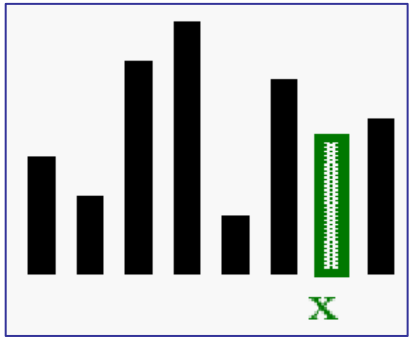
Idea of Quicksort



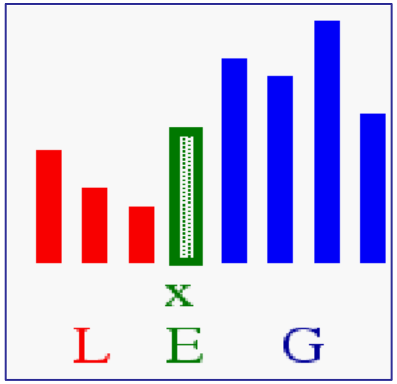
Pick a “pivot”. Divide into less-than & greater-than pivot.
Sort each side recursively.

Idea of Quicksort

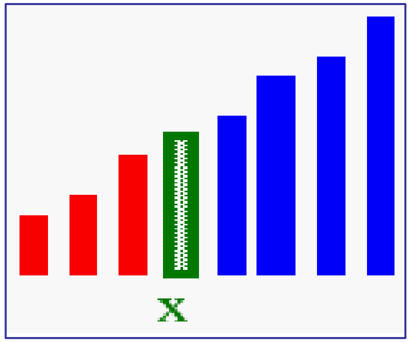
1. **Select**: pick an element



2. **Divide**: rearrange elements so that **x goes to its final position E**



3. **Recur and Conquer**: recursively sort



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

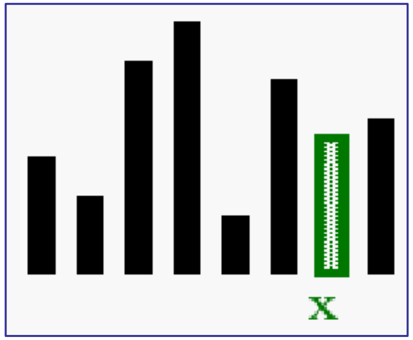
Example

We are given array of n integers to sort:

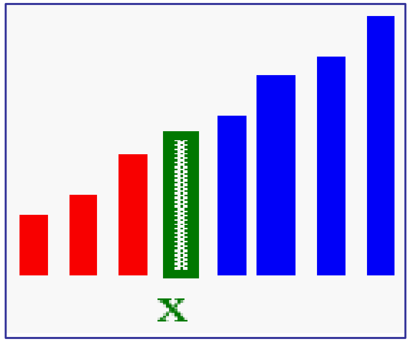
40	20	10	80	60	50	7	30	100

Idea of Quicksort

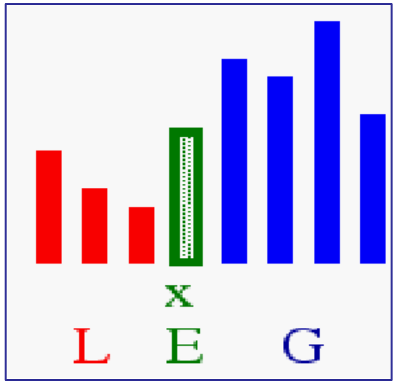
1. **Select**: pick an element



2. **Divide**: rearrange elements so that **x goes to its final position E**



3. **Recur and Conquer**: recursively sort



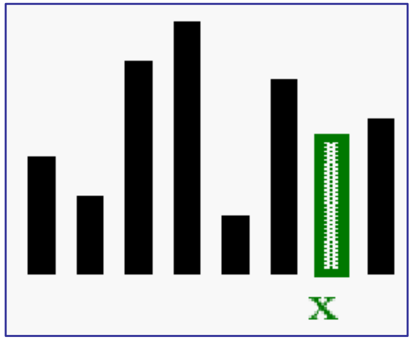
Pick Pivot Element

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

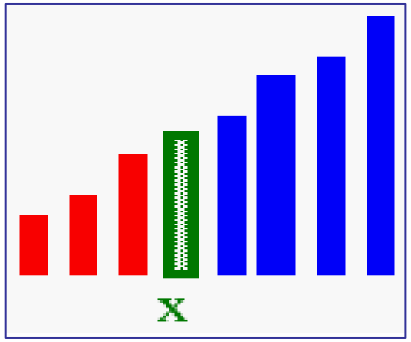
40	20	10	80	60	50	7	30	100

Idea of Quicksort

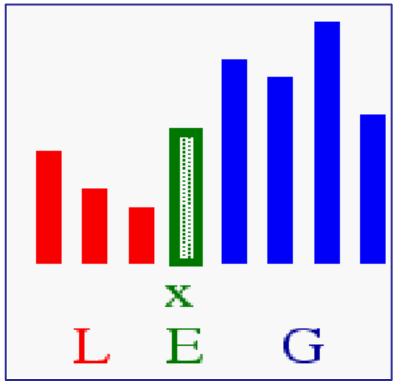
1. **Select**: pick an element



2. **Divide**: rearrange elements so that **x goes to its final position E**



3. **Recur and Conquer**: recursively sort

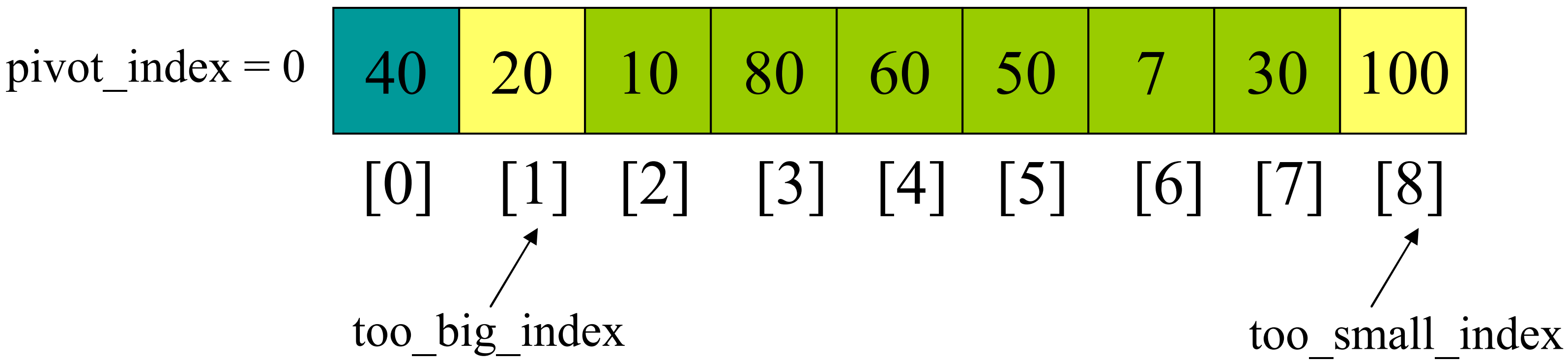


Partitioning Array

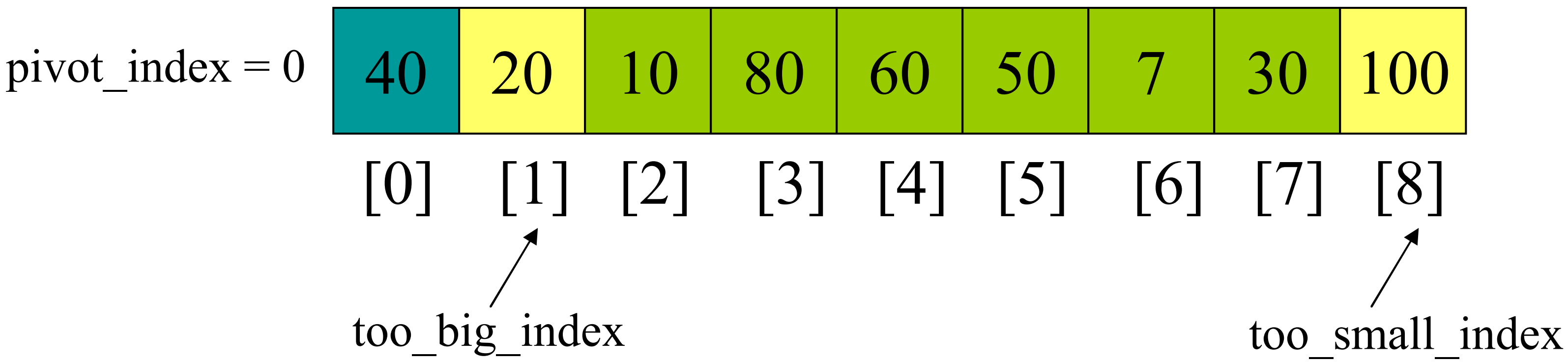
- **Given a pivot, partition the elements of the array such that the resulting array consists of:**
  - One sub-array that contains elements >= pivot
  - Another sub-array that contains elements < pivot

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

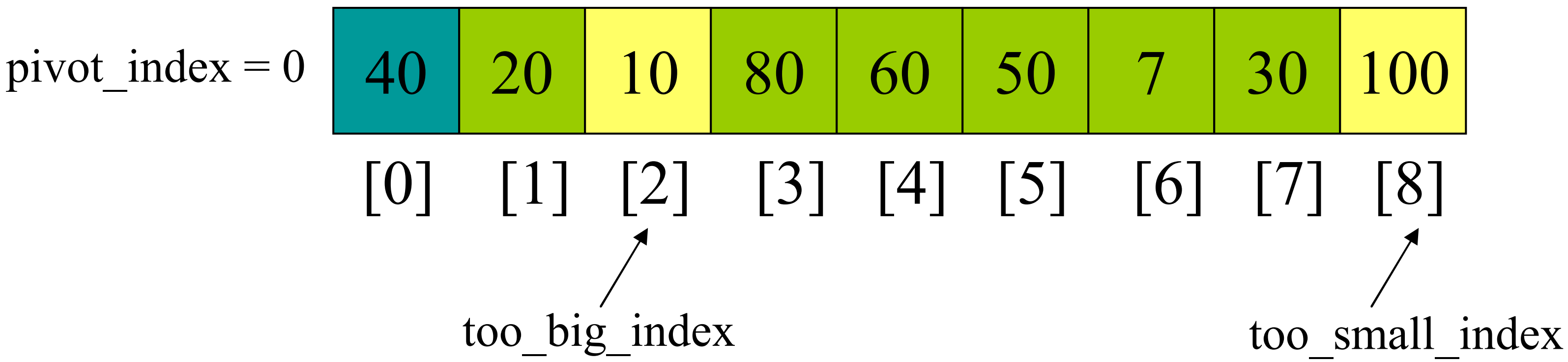
- **Partitioning loops through, swapping elements below/above pivot.**


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

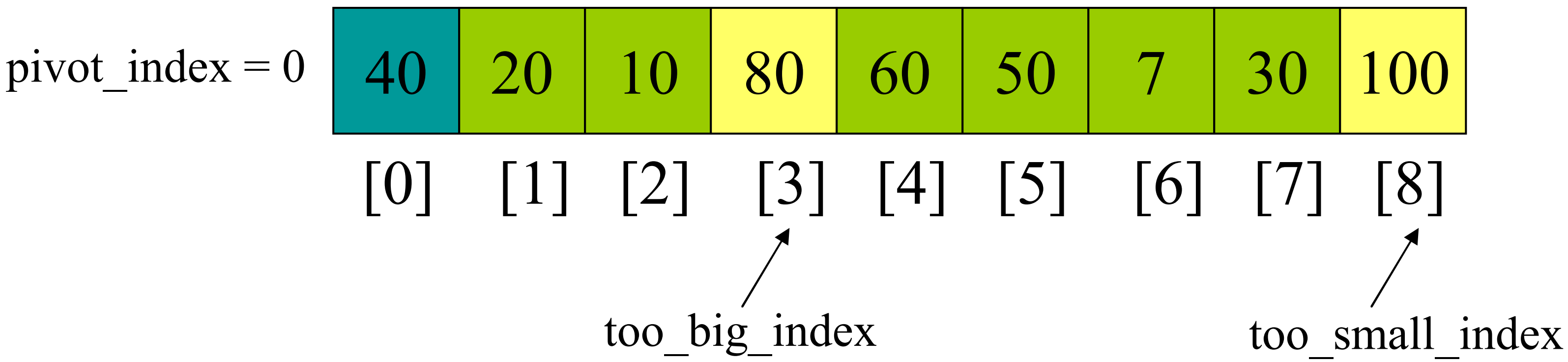
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



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



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

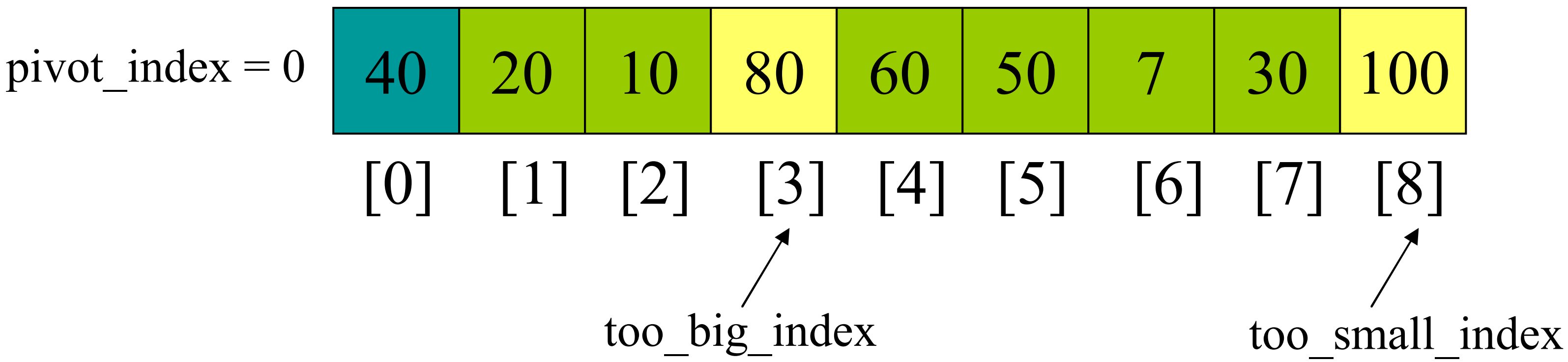


1. While data[too_big_index] <= data[pivot]

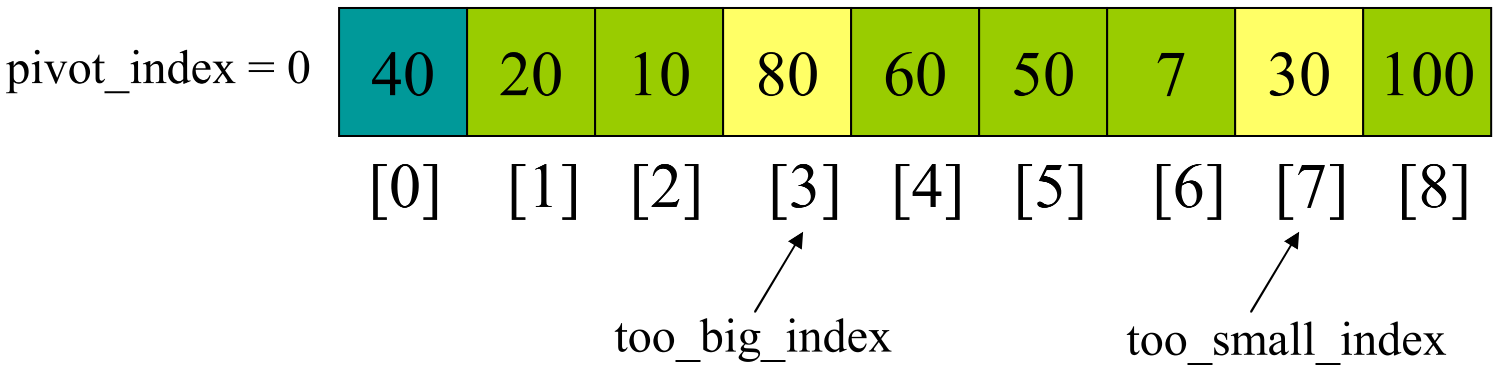
    ++too_big_index

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

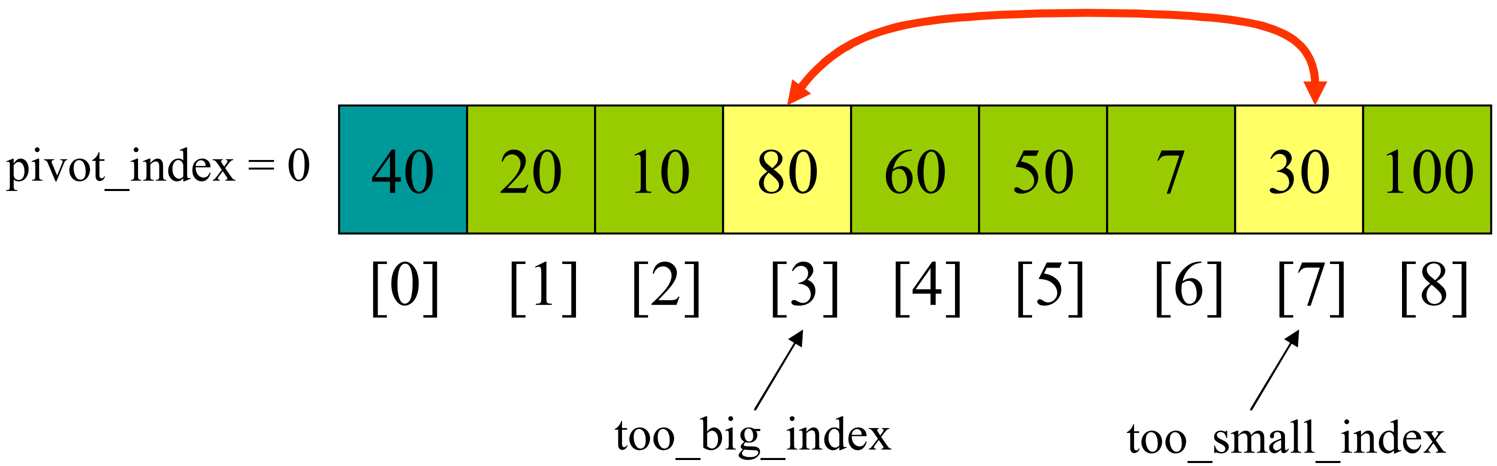
    --too_small_index



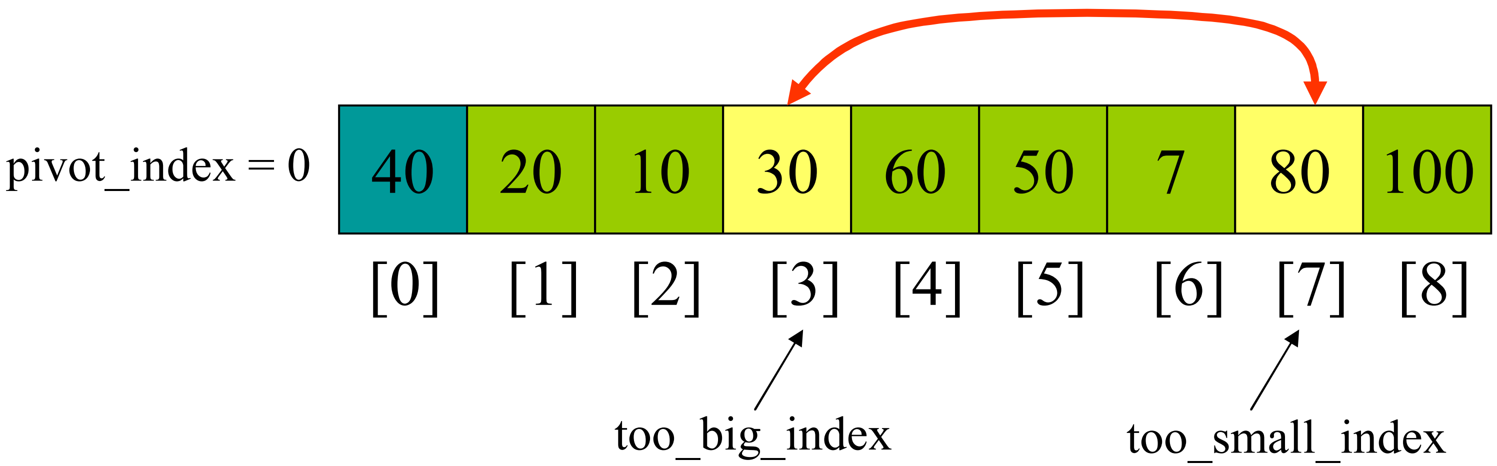
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index



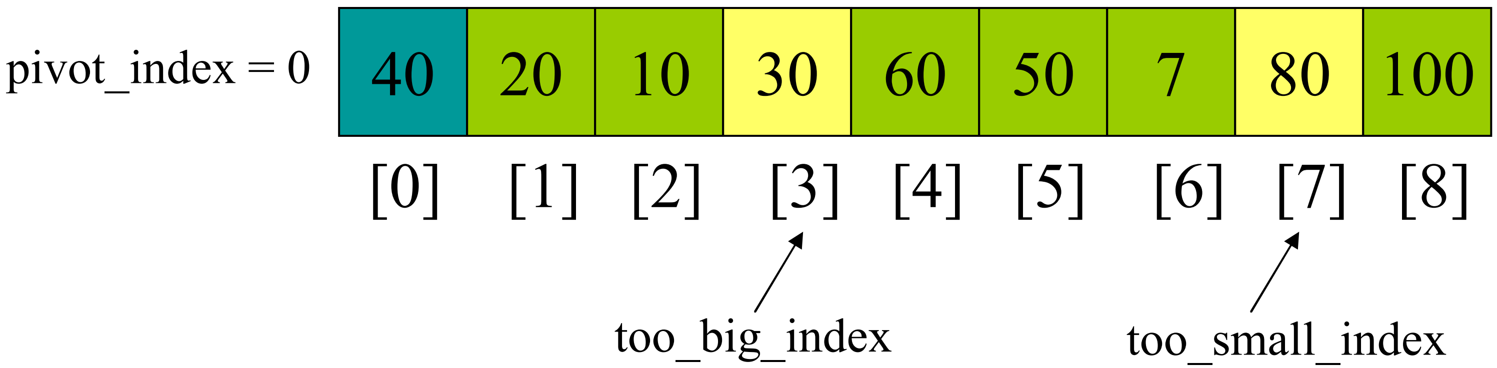
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]



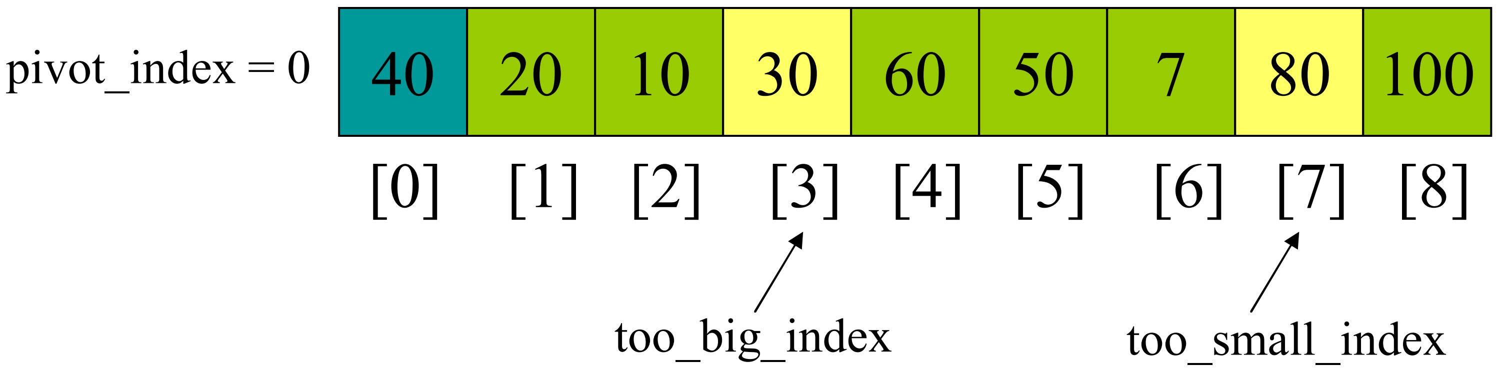
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



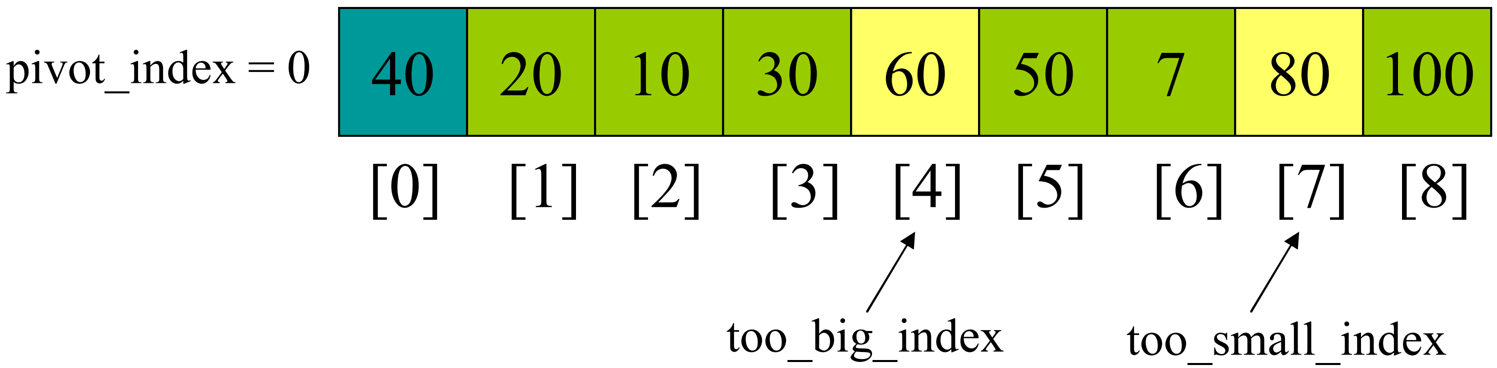
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



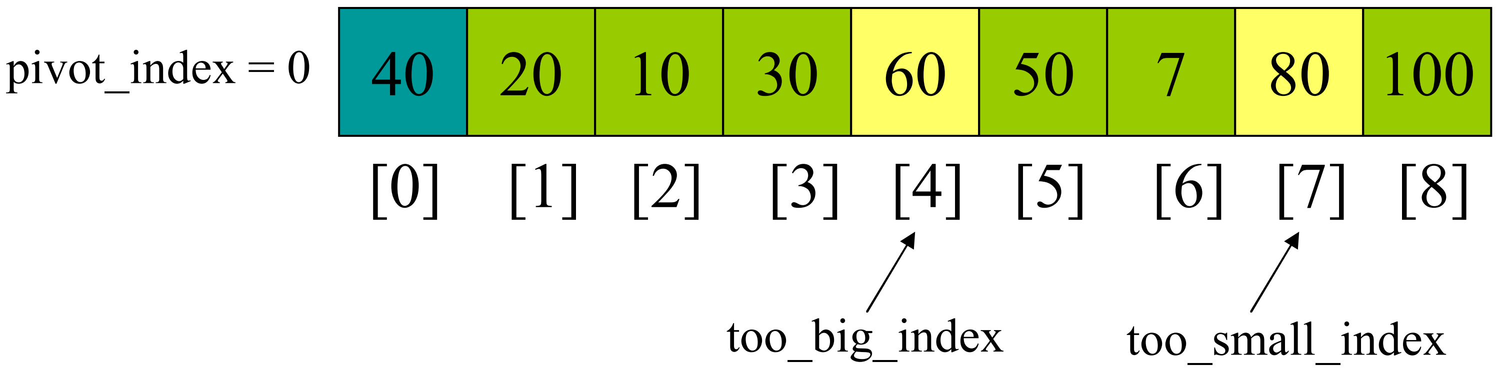
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



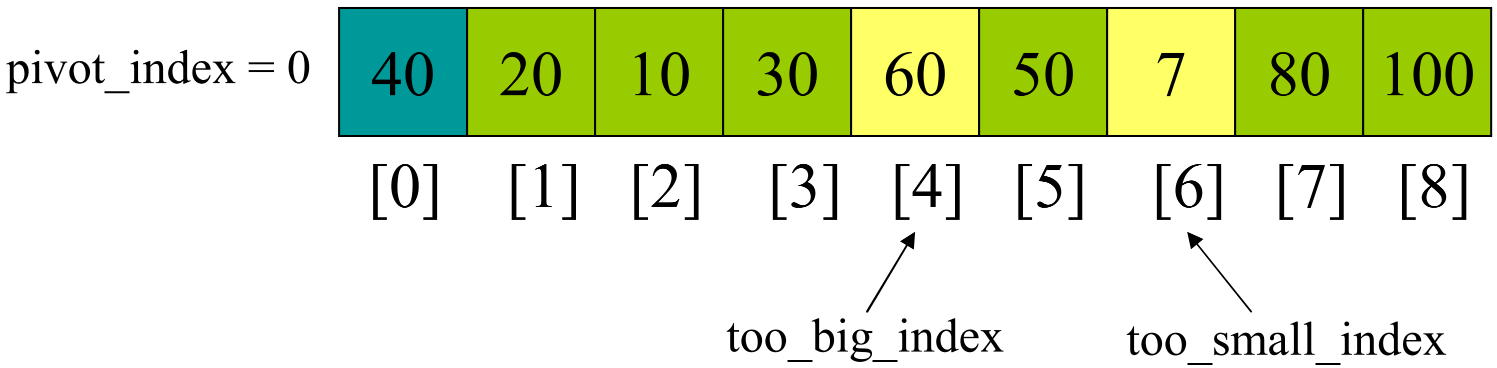
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



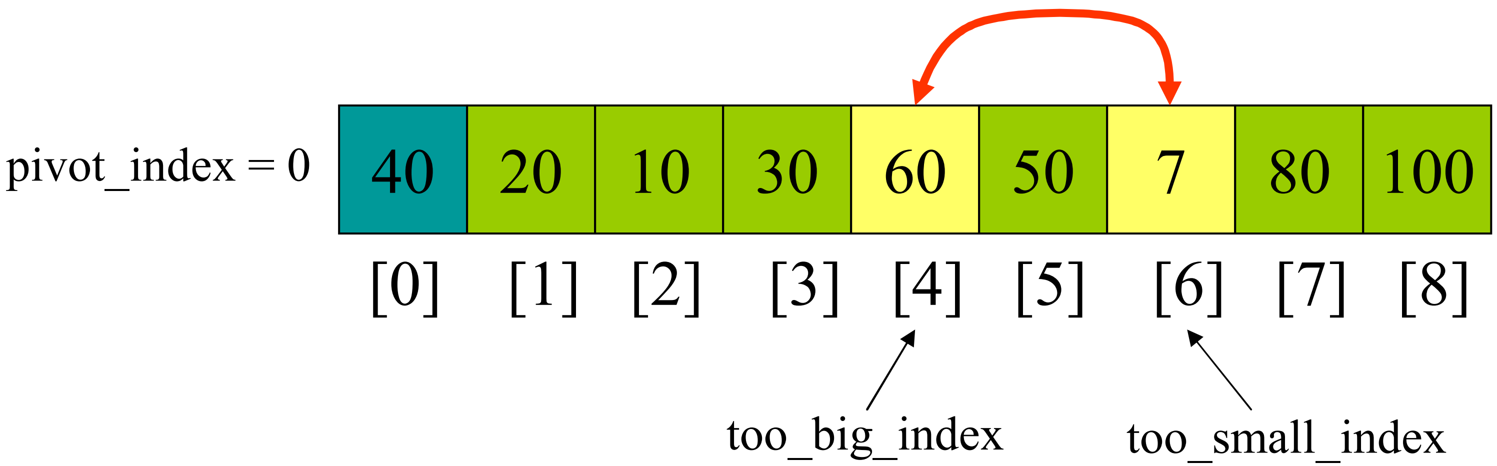
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



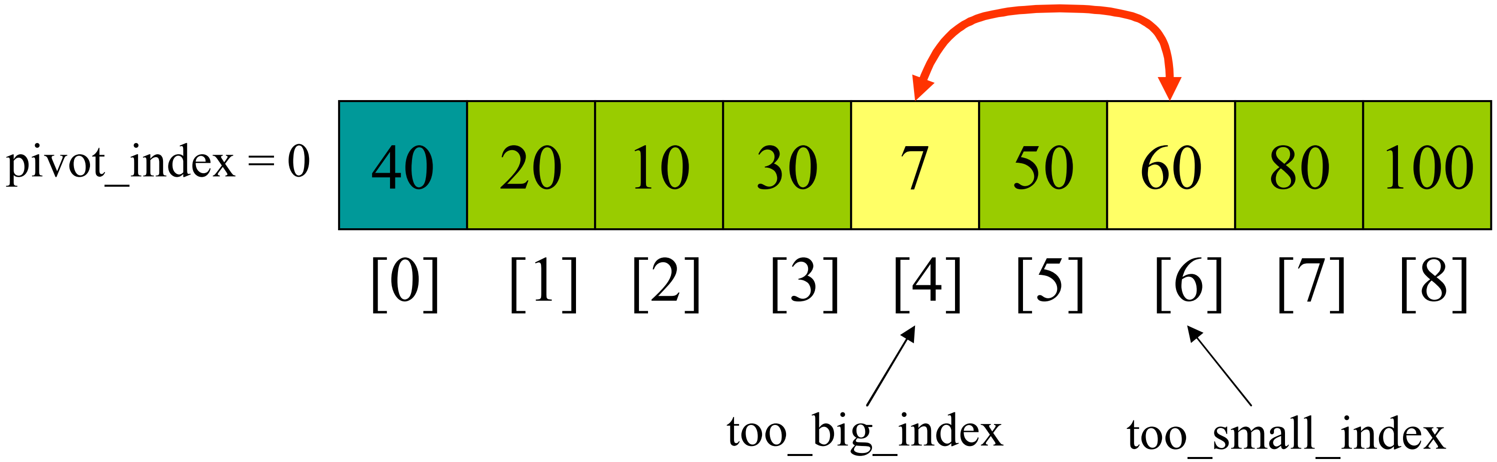
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



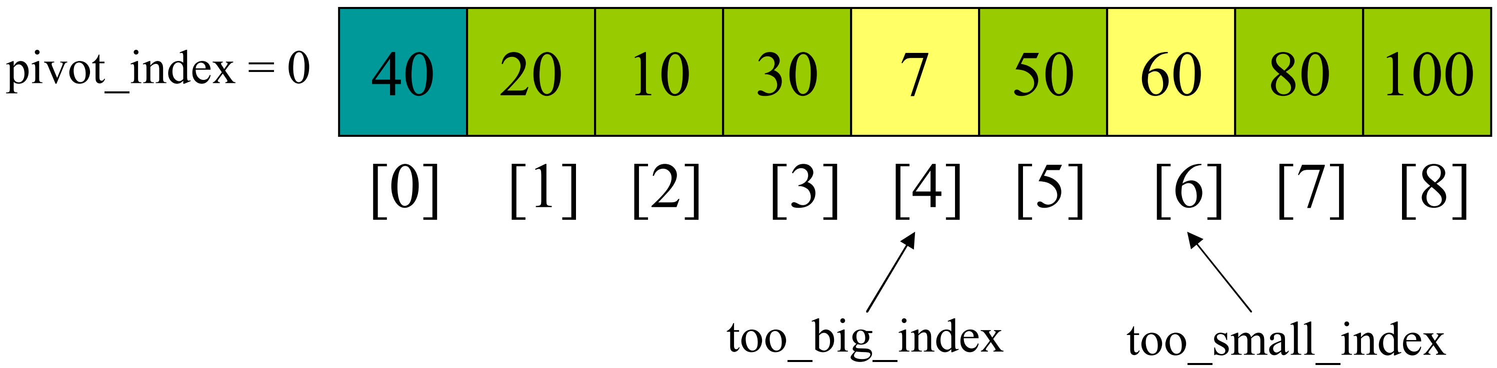
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



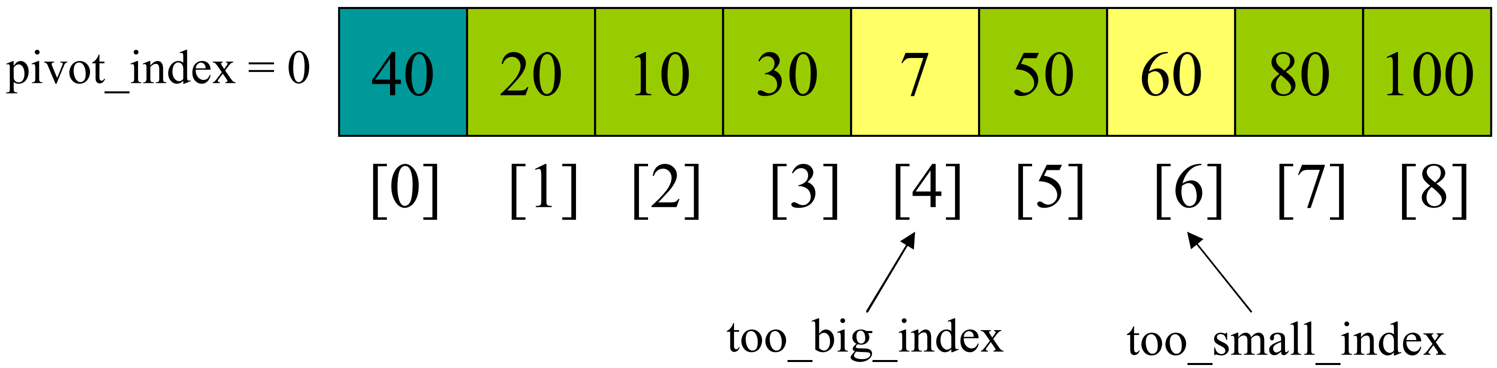
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



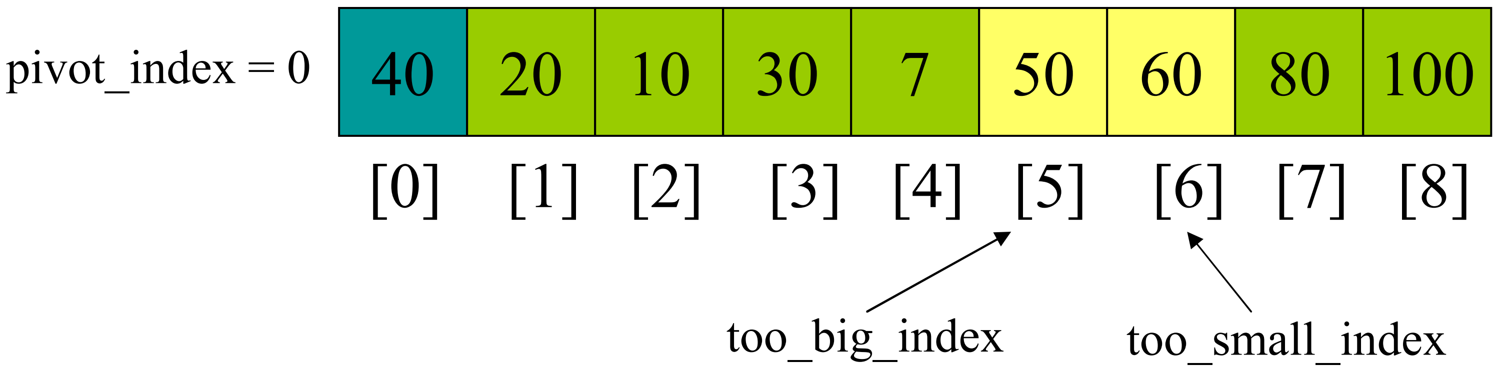
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



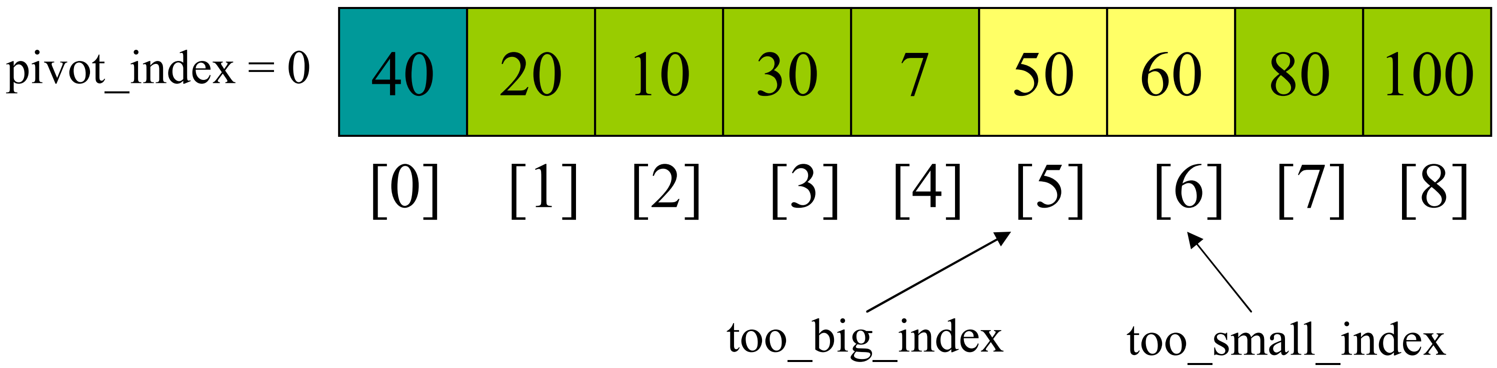
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



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

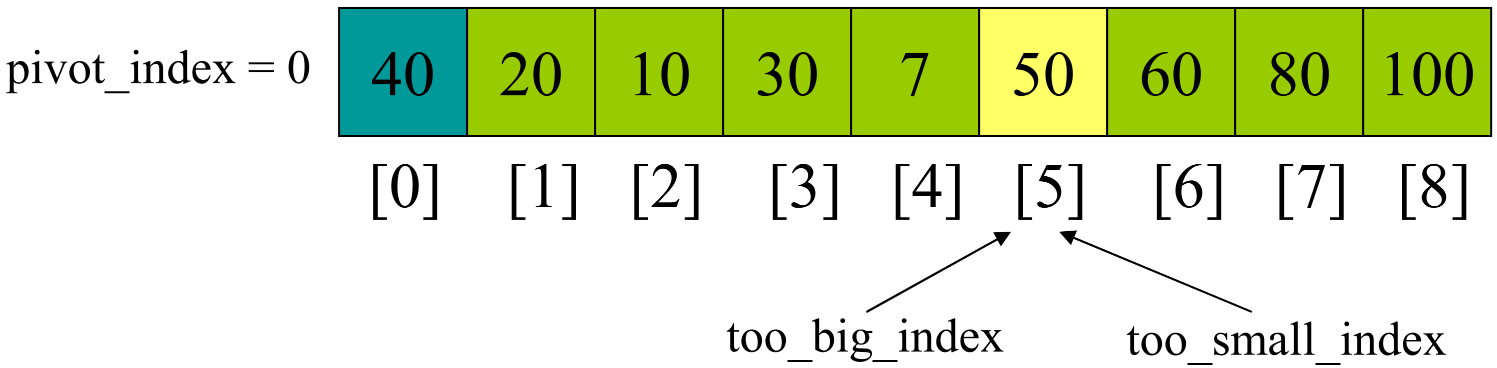
2. While data[too_small_index] > data[pivot]

    --too_small_index

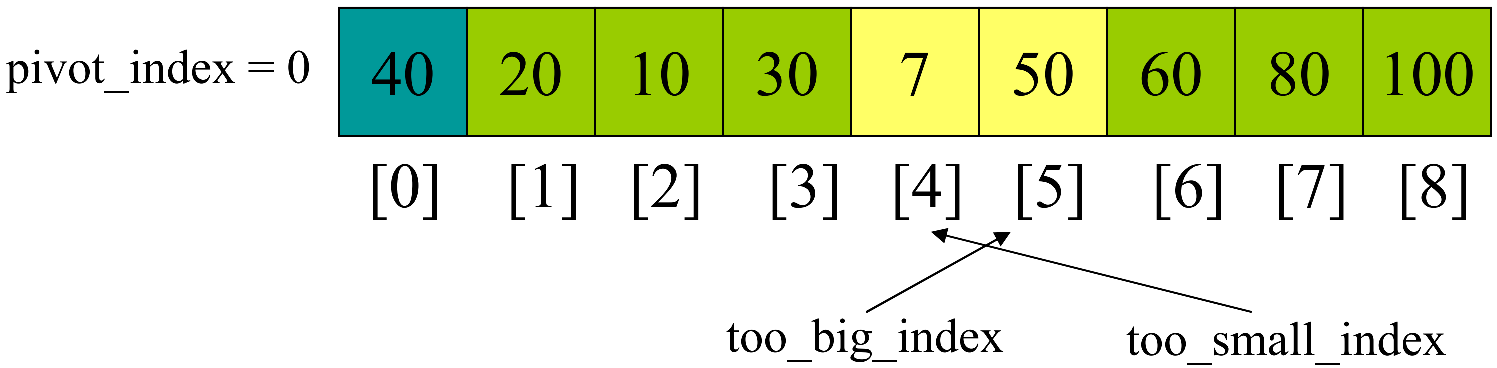
3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

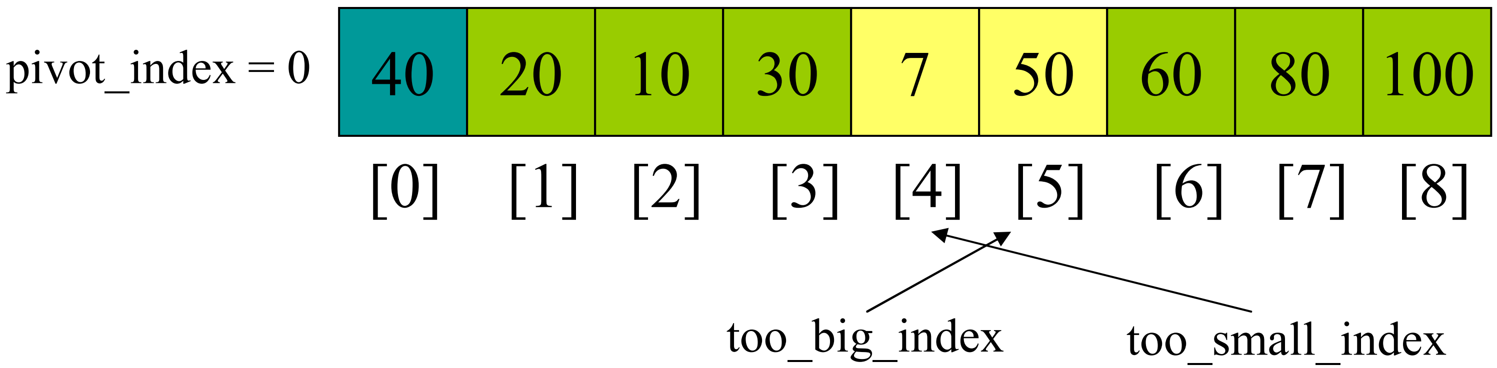
4. While too_small_index > too_big_index, go to 1.



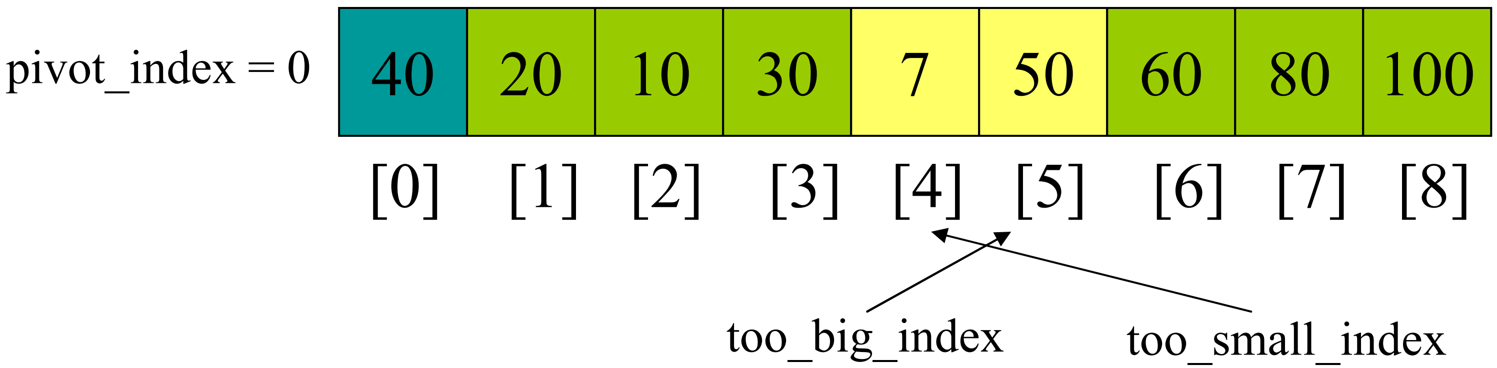
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



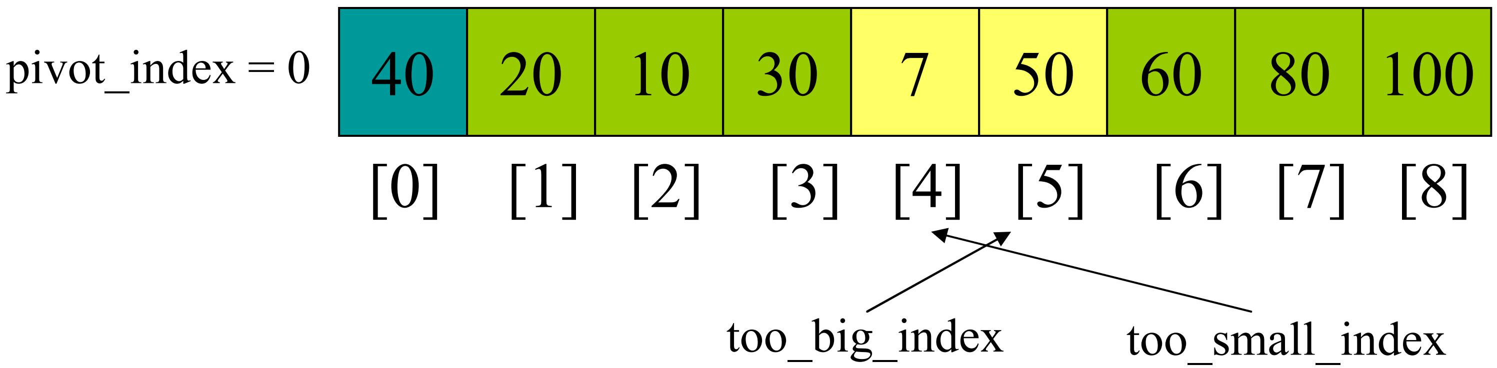
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



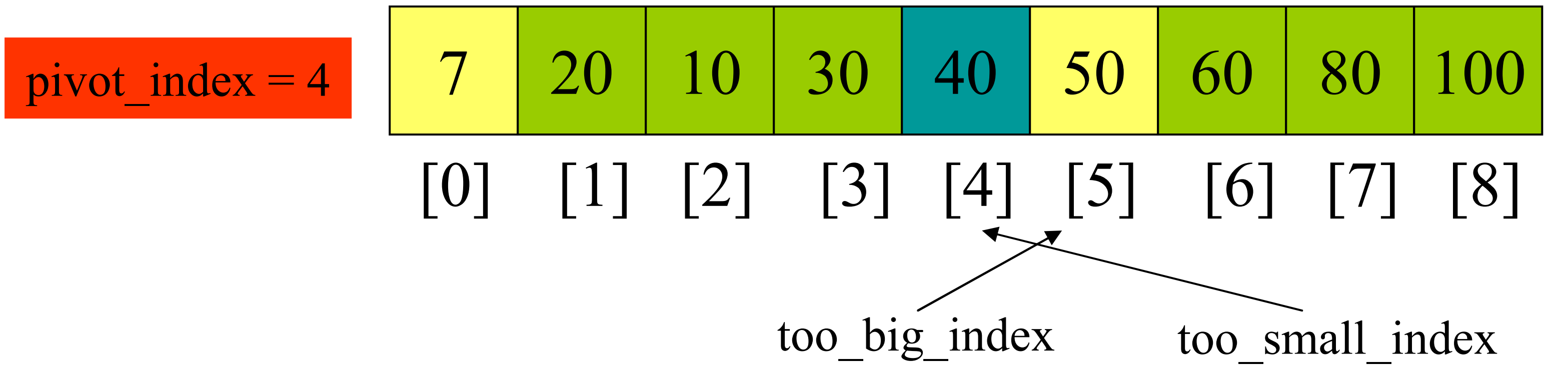
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



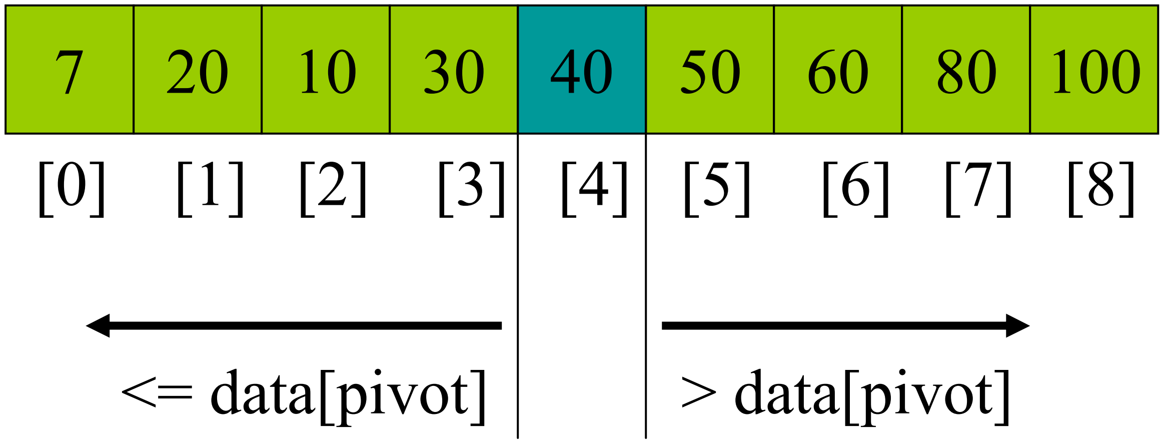
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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

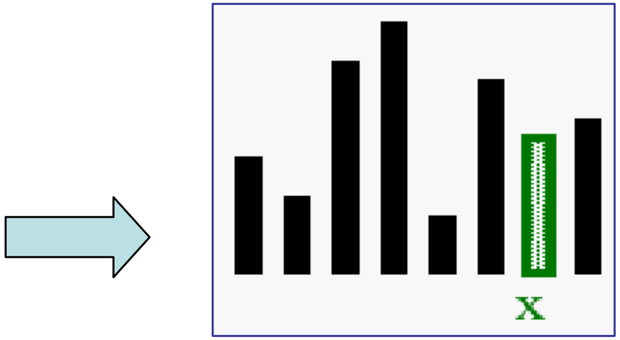


Partition Result

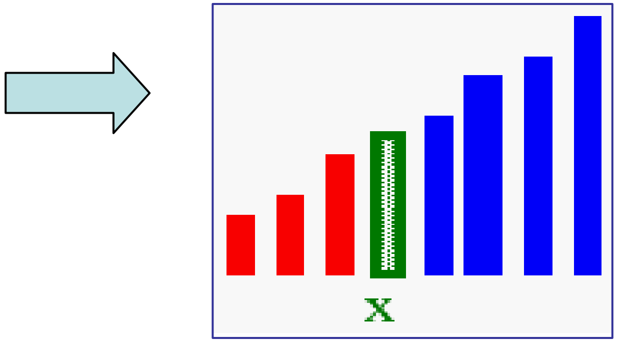


Idea of Quicksort

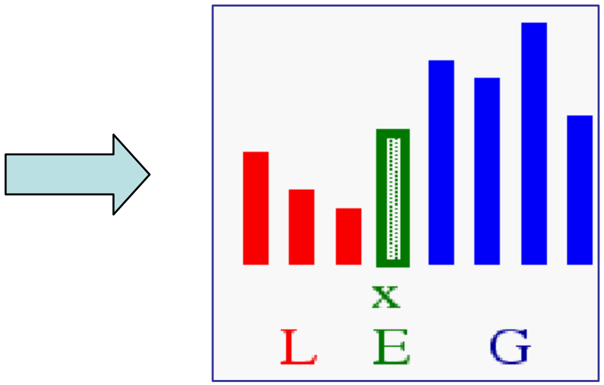
1. **Select**: pick an element



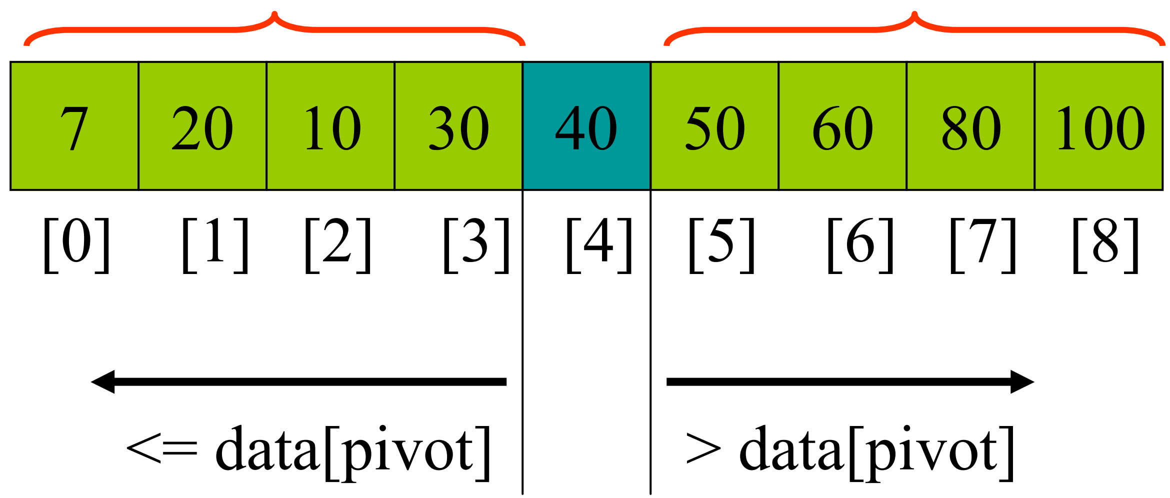
2. **Divide**: rearrange elements so that **x goes to its final position E**



3. **Recur and Conquer**: recursively sort



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

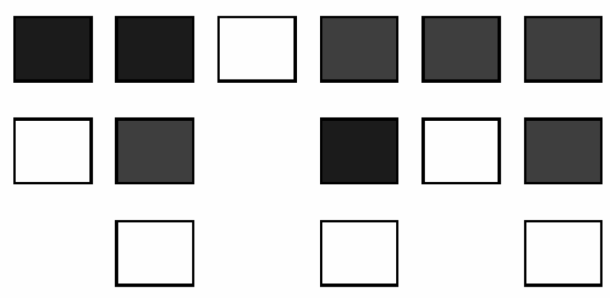
- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

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: $O(n \log_2 n)$**
  - the pivot is always the median element
  - at each recursive call the array is split into two equal parts, elements smaller than the pivot and elements larger than the pivot.



Quicksort Analysis

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

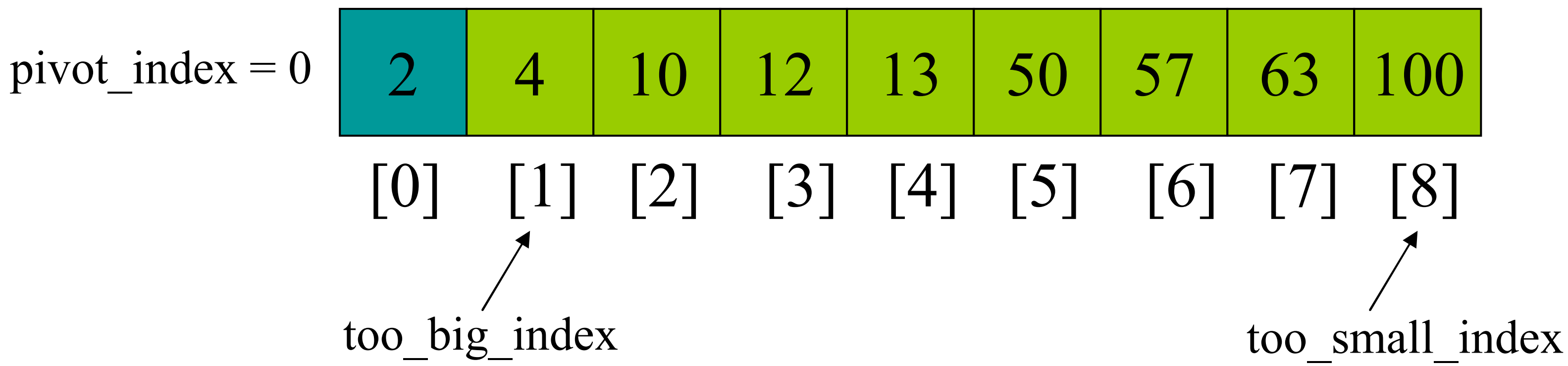
Quicksort Analysis

Worst case: $O(N^2)$

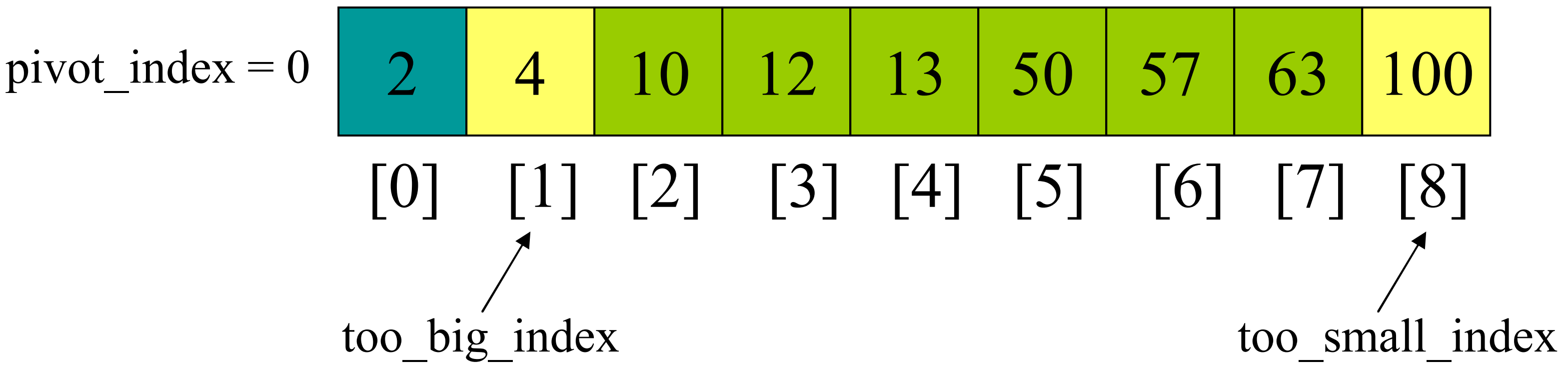
- The pivot is always the greatest (the least) element at each recursive call the array is split into a part where all elements smaller than the pivot are, the pivot, and an empty part.

Quicksort: Worst Case

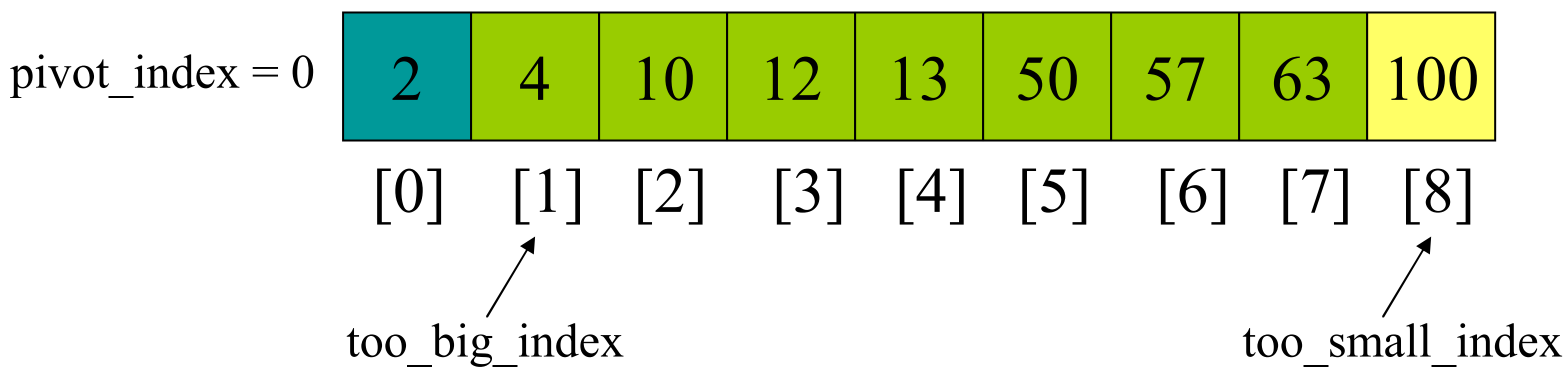
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



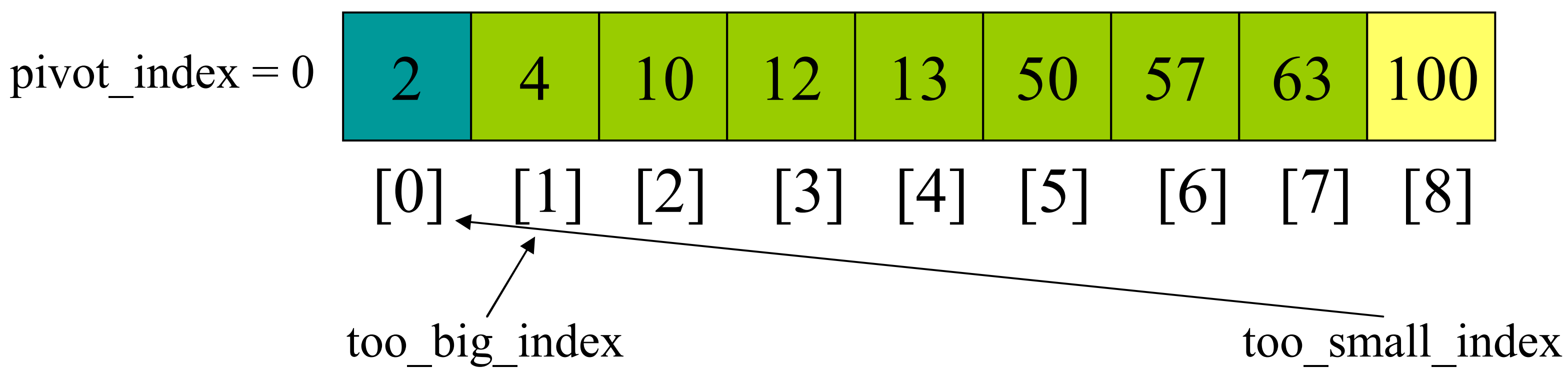
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



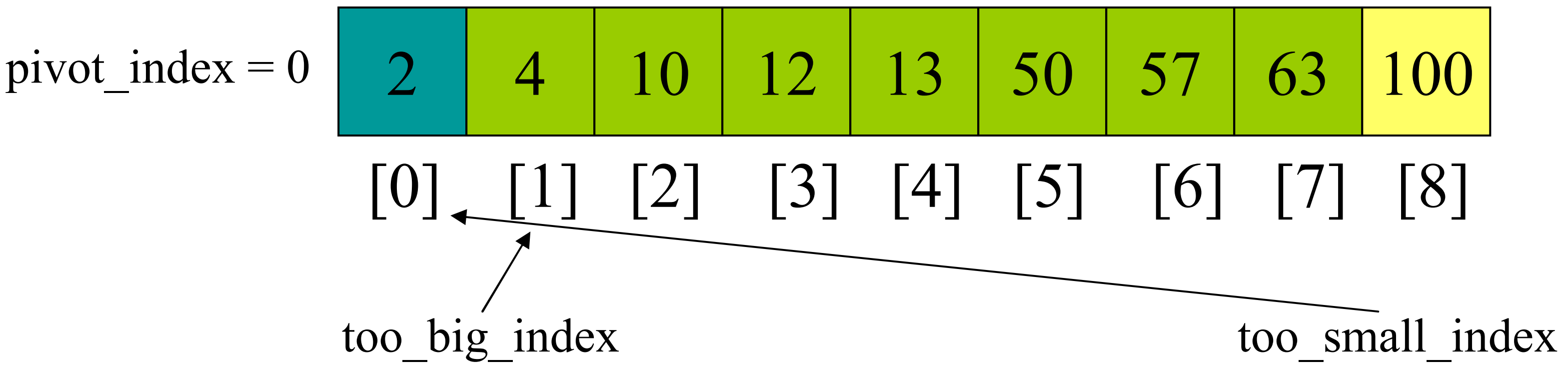
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



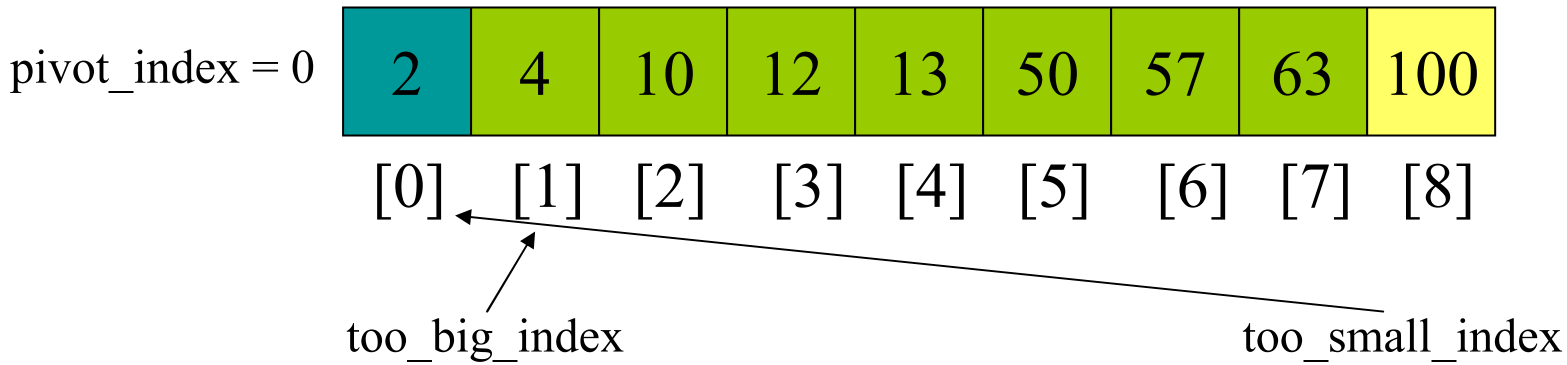
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



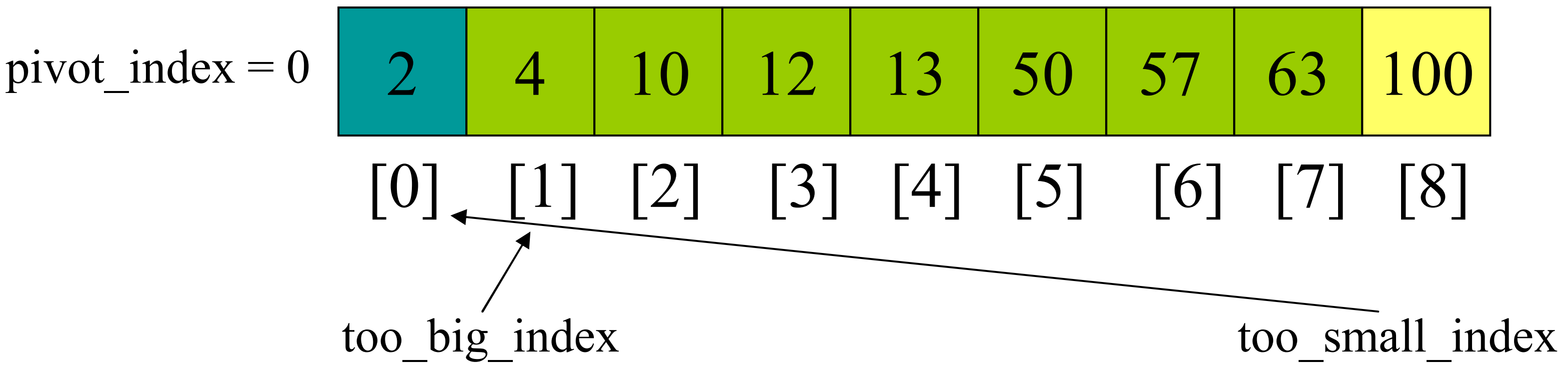
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



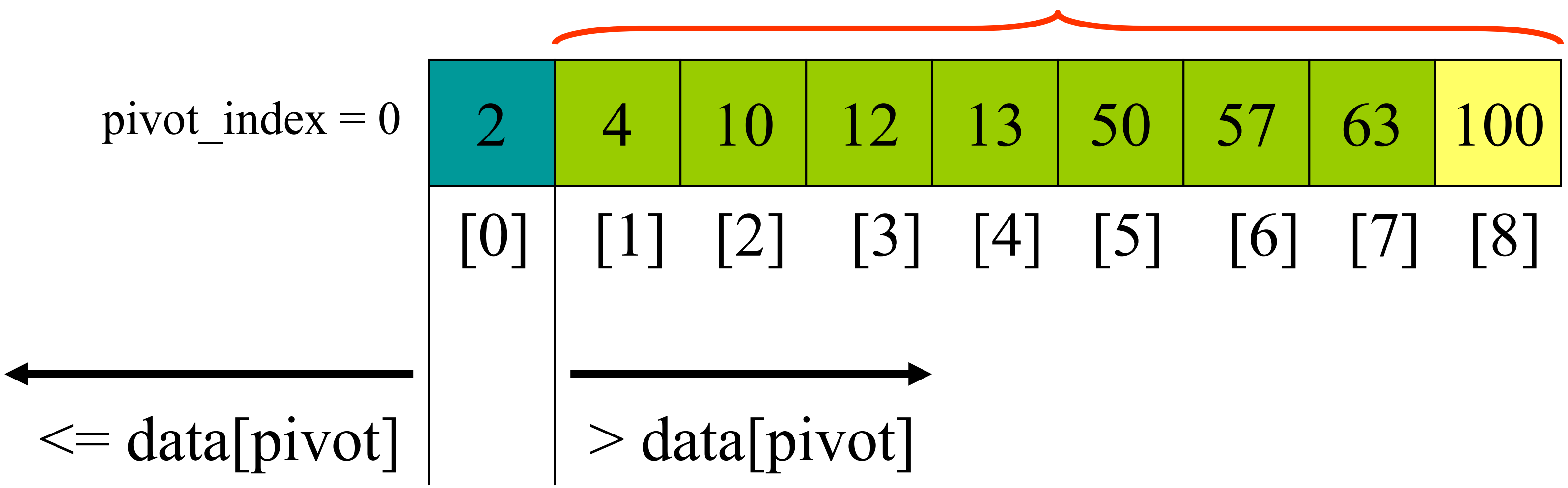
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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

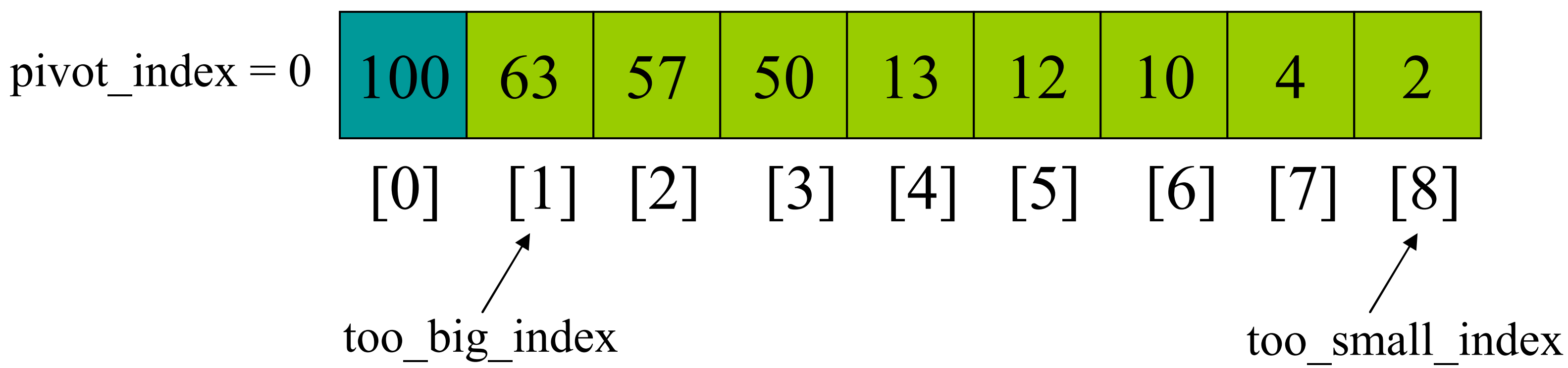


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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

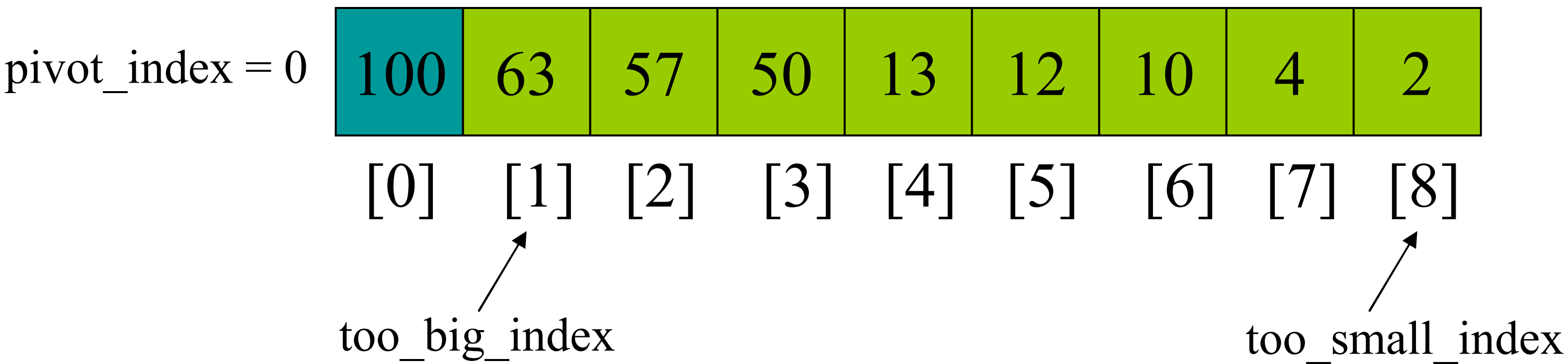


Quicksort:

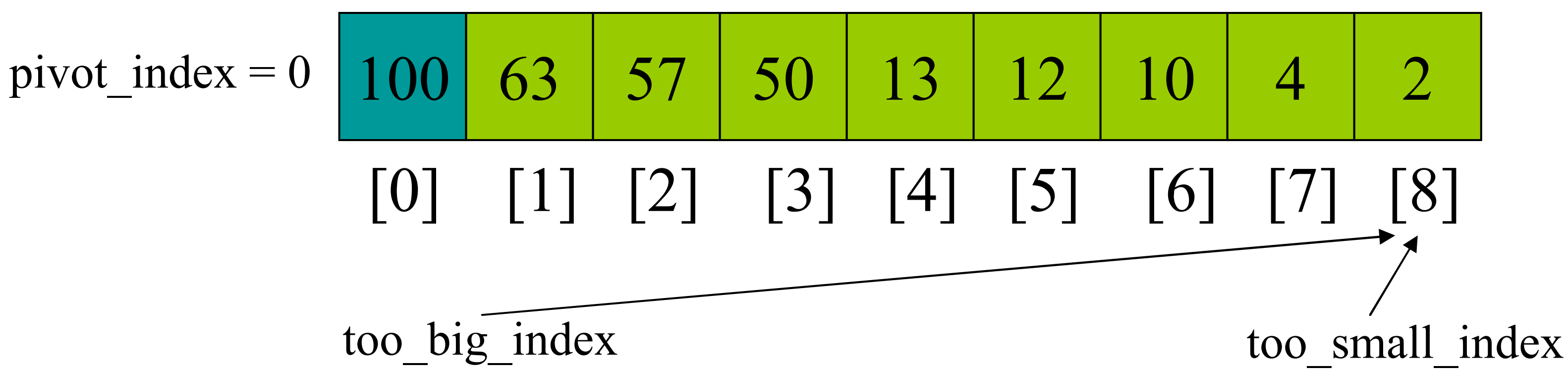
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



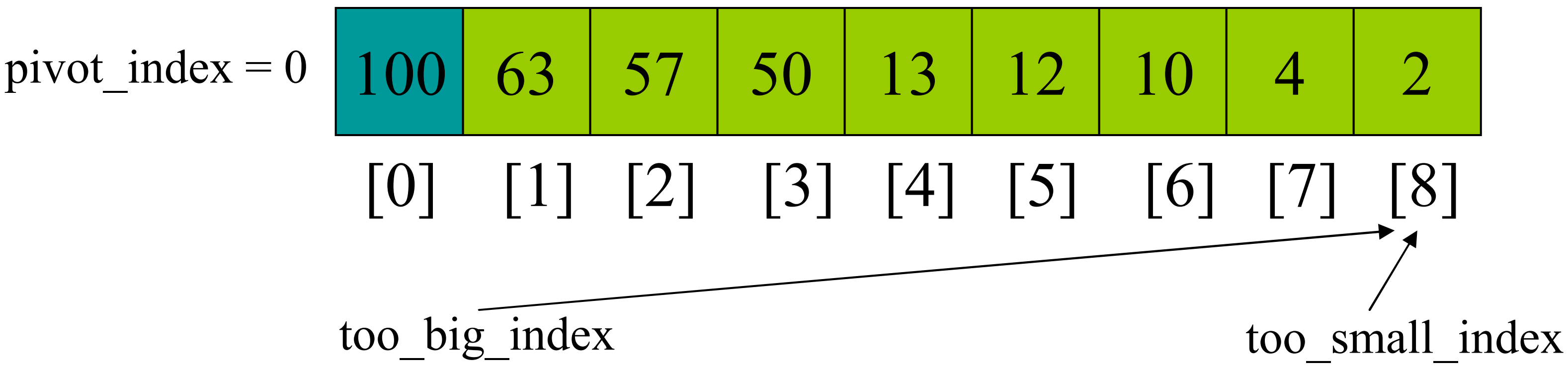
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



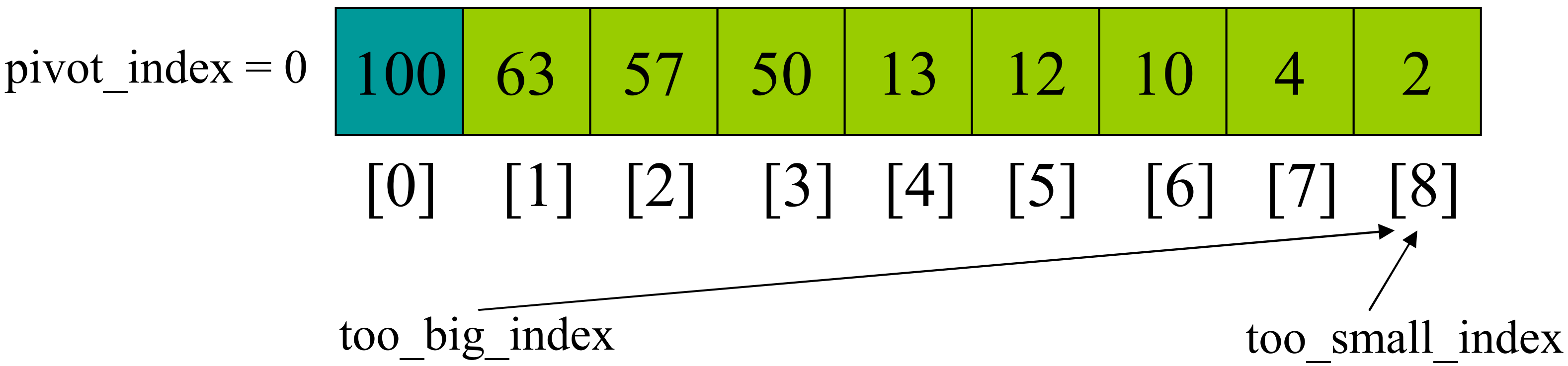
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



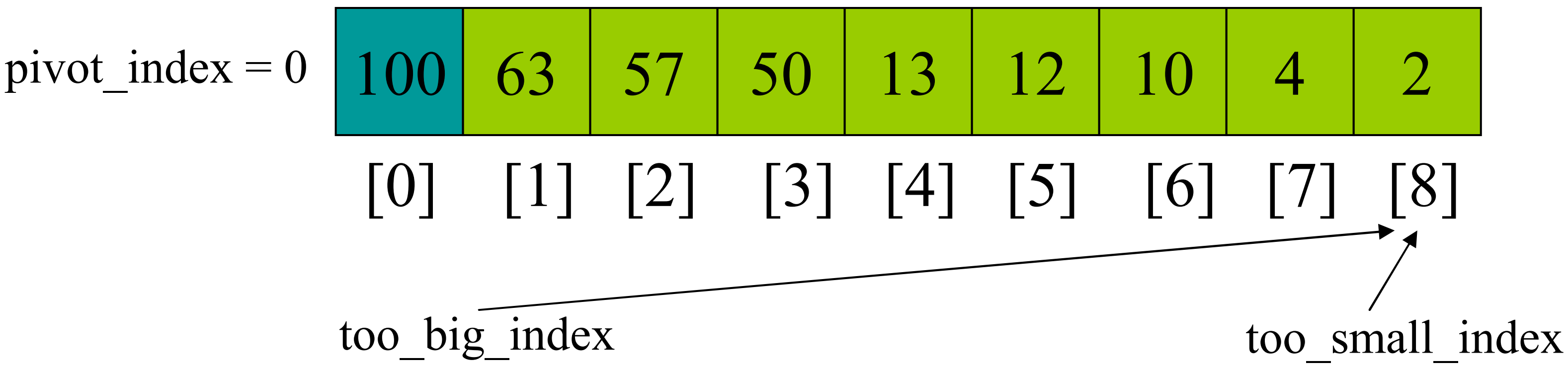
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



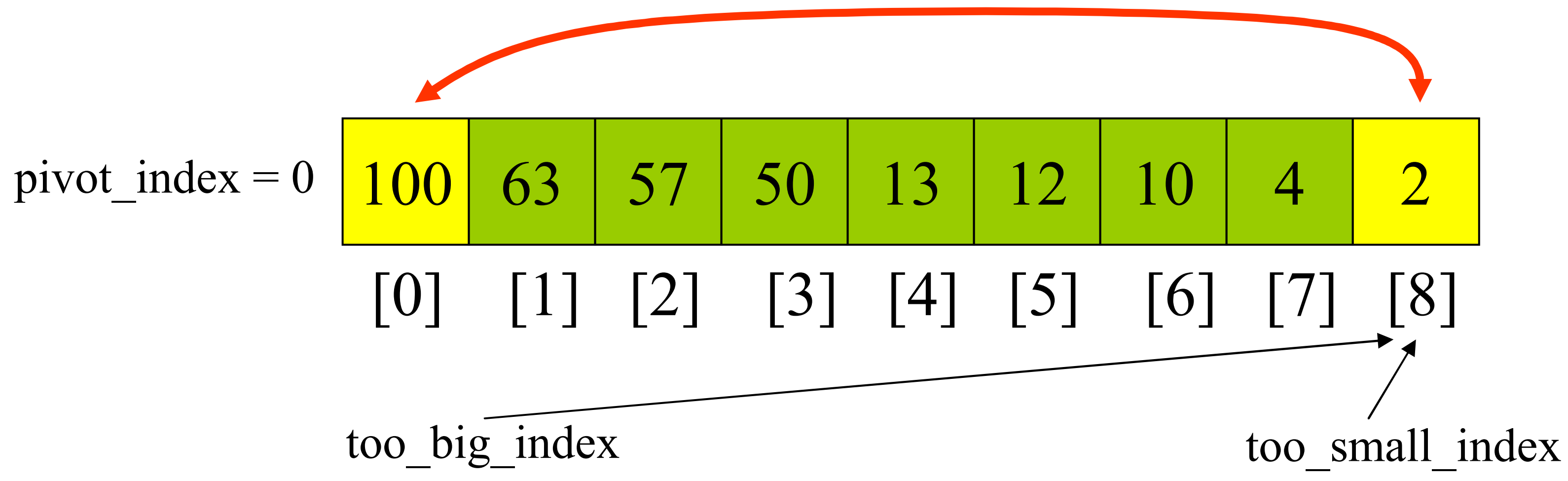
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



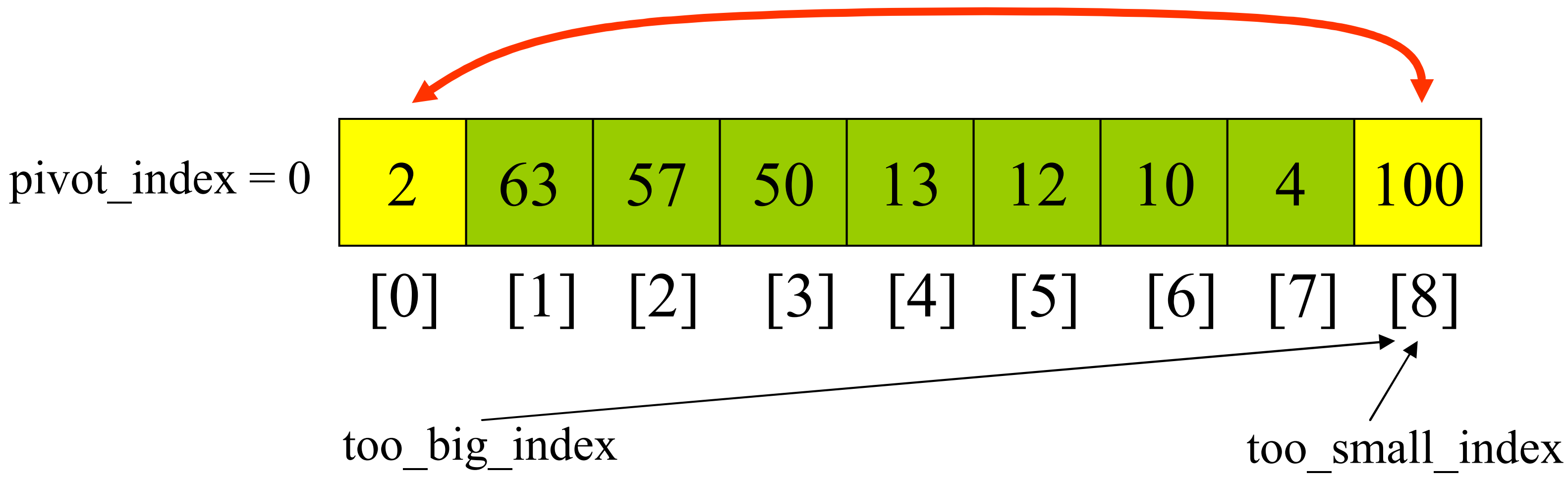
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



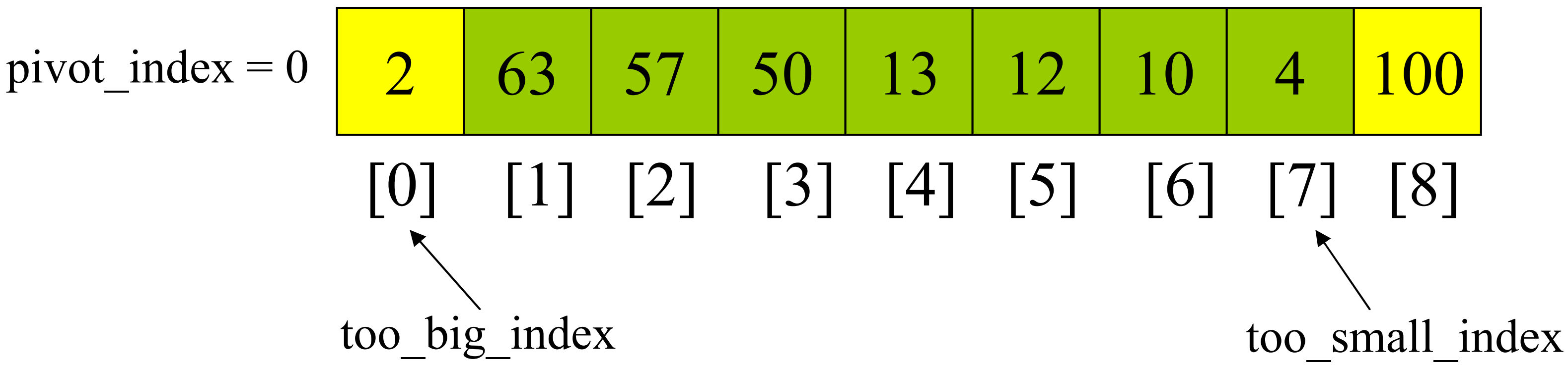
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



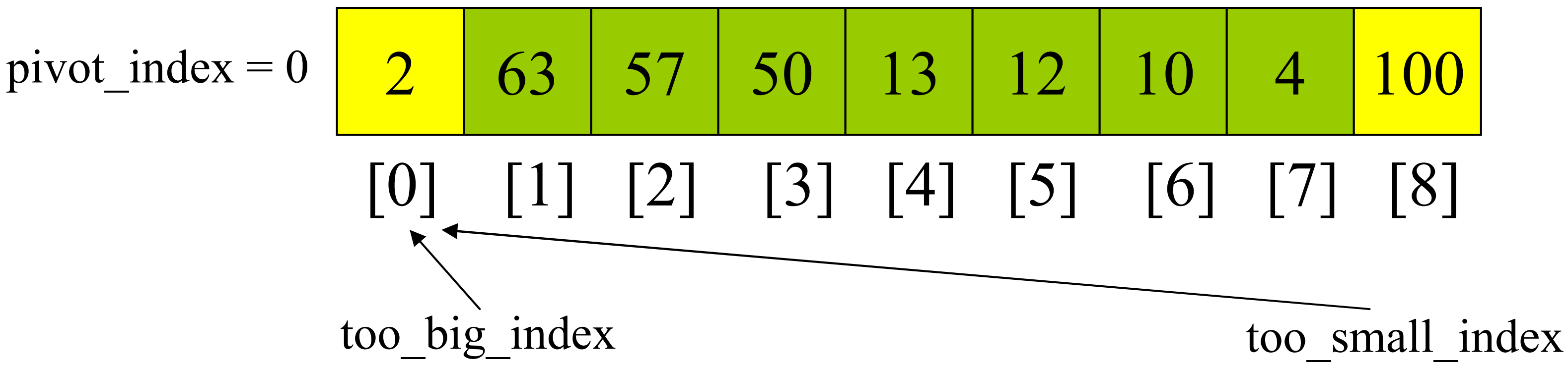
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



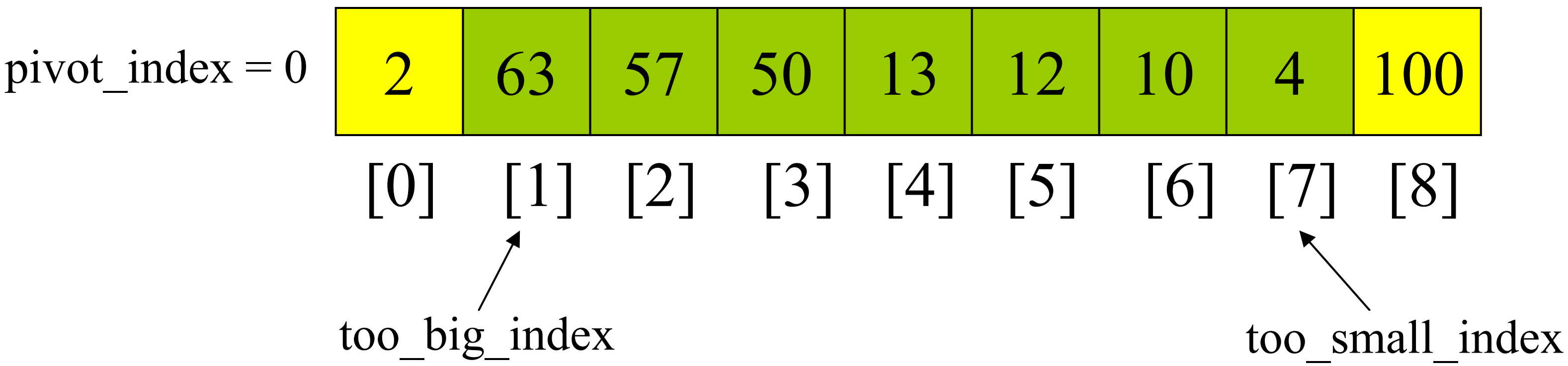
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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

- Complicated proof using recurrences
- If the pivot is chosen at random, what is its average expected value?
- About the median of all values in that part of the array
- Hence on average the array is split in about equal parts
- Hence quicksort in the average case behaves as in the best case: $O(N \log N)$.

Quicksort Analysis

- **Best** case running time: $\mathbf{O(n \log_2 n)}$
- **Worst** case running time: $\mathbf{O(n^2)}$
- **Average** case running time: $\mathbf{O(n \log_2 n)}$

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	• in-place • slow (good for small inputs)
insertion-sort	$O(n^2)$	• in-place • slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	• in-place, randomized • fastest (good for large inputs)
merge-sort	$O(n \log n)$	• sequential data access • fast (good for huge inputs)

Mergesort X Quicksort

- **Mergesort**
  - Splits partitions in half
  - Merges smaller lists back into larger list
  - Requires overhead when sorting arrays

- **Quicksort**
  - Relies on a pivot point for sorting
  - Smaller sets are sorted based on pivot point
  - Can perform slowly if a bad pivot point is used

Mergesort X Quicksort

- **Mergesort**
  - Use extra space
  - Is guaranteed to have O(n log n) performance in the worst case
- **Quicksort**
  - Does **not** use extra space
  - Is **not** guaranteed to have O(n log n) performance in the worst case, unlike merge sort

Mergesort X Quicksort

- **Advantages of quicksort over mergesort**
  - Quicksort doesn't require an extra array
  - the hidden constant in the average case for quicksort is smaller than the hidden constant for merge sort.

- **Advantage of mergesort over quicksort:**
  - better worst case behavior

- **Quicksort and mergesort are optimal, in the sense that a general sorting algorithm cannot do better than average case O(nlog n).**

Mergesort X Quicksort

- Both QuickSort and MergeSort are O(n log n) for their average cases.

- However, the characteristic of 'O' notation is that you drop all constant factors.

- Therefore, one O(n) algorithm could take 1000 times as long as another O(n) algorithm, and as long as the complexity isn't dependant on the size on 'n', they're both O(n) algorithms.

Mergesort X Quicksort

- The advantage of QSort over MergeSort is that it's constant factor is smaller than MergeSort's, and so therefore, on the average case, it is faster (but not less complex).
- Empirically, QSort is best, especially if the 'pivot' value has been properly selected

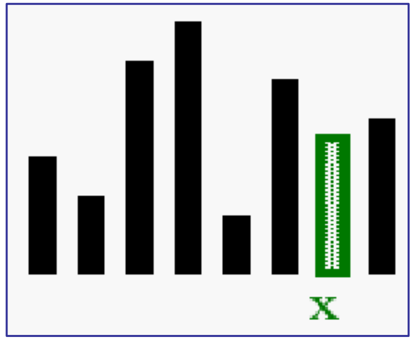
Quicksort Analysis

- **Best** case running time: $\mathbf{O(n \log_2 n)}$
- **Worst** case running time: $\mathbf{O(n^2)}$
- **Average** case running time: $\mathbf{O(n \log_2 n)}$

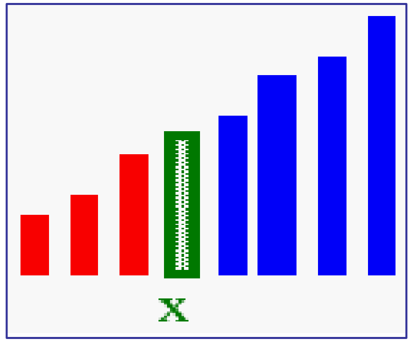
What can we do to avoid worst case?

Idea of Quicksort

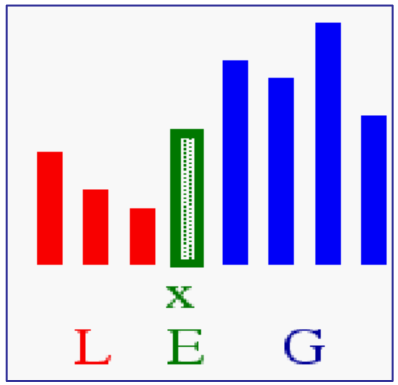
1. **Select**: pick an element



2. **Divide**: rearrange elements so that **x goes to its final position E**



3. **Recur and Conquer**: recursively sort



Choosing the Pivot

- **If we happen to pick the pivot which is always the largest or the smallest element, quicksort works just like selection sort**

- **If we always pick the element with the median value, it splits the array in half at every recursion level**

- **Hence: best case is O(N log N), worst case quadratic**

Choosing the Pivot

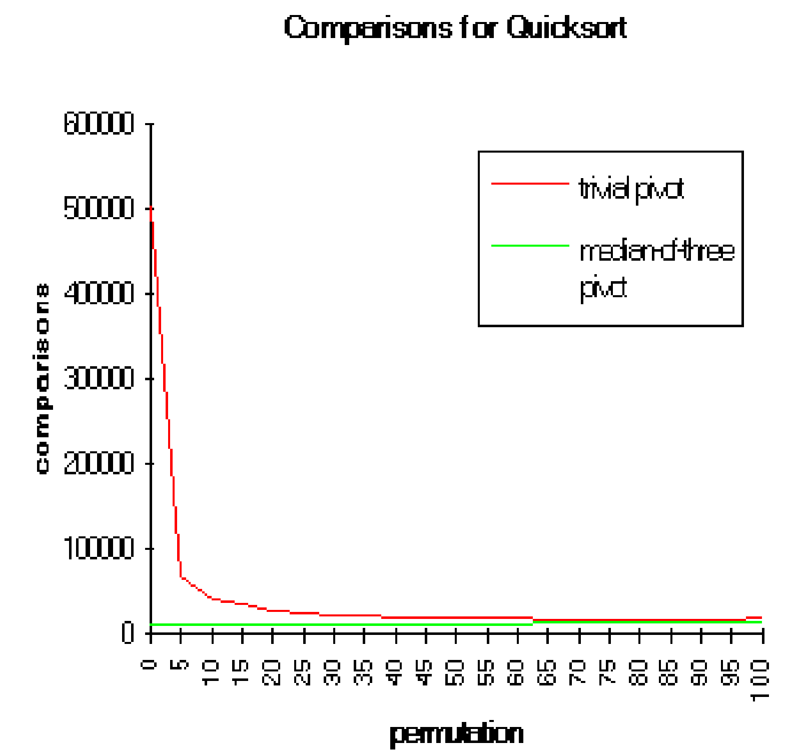
- Pick first, last, or middle element: works fine with random arrays
- Generate a random index
- Pick median value of three elements from data array: data[0], data[n/2], and data[n-1].
- ...

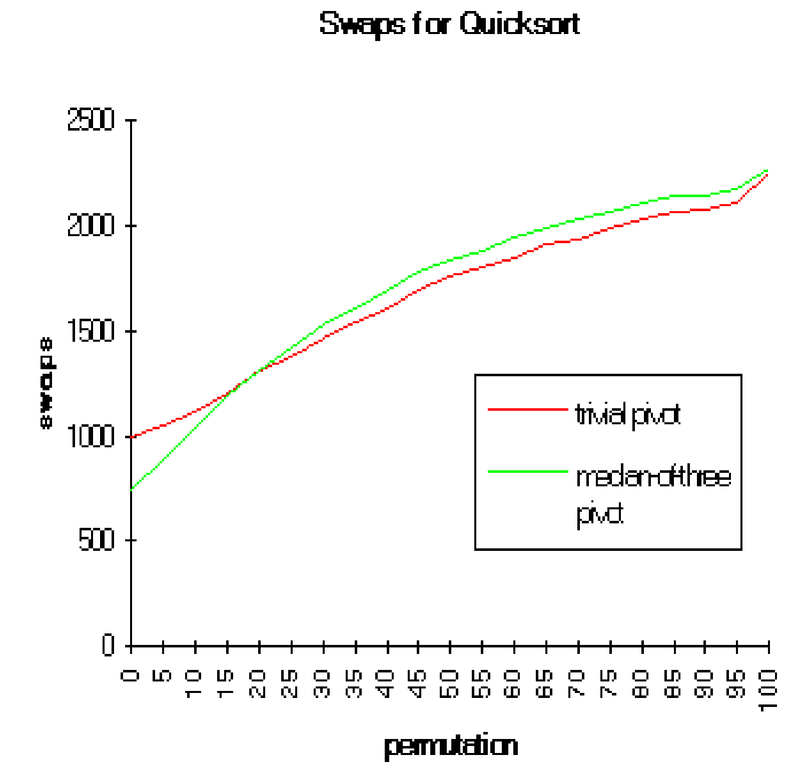
This algorithm works with two counters, i, j, that count inwards from the left and right, that swap across elements that are the wrong size compared to the pivot, stopping when the pointers cross.

```
void quicksort(int a[], int left, int right)
{
        int i,j,v;

        if(right >  left) {
                v = a[right]; i = left - 1; j = right;
                for(;;) {
                        while(a[++i] <  v) ;
                        while(a[--j] >  v) ;
                        if(i >= j) break;
                        swap(a, i, j);
                }
                swap(a, i, right);
                quicksort(a, left, i - 1);
                quicksort(a, i + 1, right);
        }
}
```

This algorithm works with two counters, i, j, that count inwards from the left and right, that swap across elements that are the wrong size compared to the pivot, stopping when the pointers cross.

```
void quicksort(int a[], int left, int right)
{
        int i,j,v,m, mid;

        if(right >  left) {
                if((right - left) >  3) {
                        m = get_median(a, left, right);
                        if(m != right)
                                swap(a, m, right);
                }
                v = a[right]; i = left - 1; j = right;
                for(;;) {
                        while(a[++i] <  v) ;
                        while(a[--j] >  v) ;
                        if(i >= j) break;
                        swap(a, i, j);
                }
                swap(a, i, right);
                quicksort(a, left, i - 1);
                quicksort(a, i + 1, right);
        }
}
```


Comparisons for Quicksort
comparisons
600000
500000
400000
300000
200000
100000
0
permutation
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
trivial pivot
median-of-three pivot


Swaps for Quicksort
swaps
2500
2000
1500
1000
500
0
permutation
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
trivial pivot
median-of-three 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.