ICS 240
Introduction to Data Structures

Jessica Maistrovich

Metropolitan State University

Sorting

Recall selection sort and insertion sort

- Selection sort:
  - After `m` iterations of **`selection`** sort, we are sure that the elements in the segment `data[0]..data[m-1]` are sorted and guaranteed to be in their final spots (never to be moved again)
  - Selection sort is always **$O(n^2)$**

- Insertion sort:
  - After `m` iterations of **`insertion`** sort, we are sure that the elements in the segment `data[0]..data[m-1]` are sorted (but the elements may change their positions later)
  - Insertion sort is **$O(n^2)$** on average but **$O(n)$** in the best case (when the array is already sorted)

Quadratic Sorts

- Selection sort and insertion sort are called the quadratic sorts because they operate in O (n^2) time
- They aren’t very good sorts

So lets try Recursive Sorts

- Divide the data to be sorted into two groups of (almost) equal size
- Sort each of these smaller groups of elements (by recursive calls)
- Combine the two sorted groups into one large sorted list
- Divide and conquer sorting algorithms:
  - Merge sort
  - Quick sort

Merge Sort

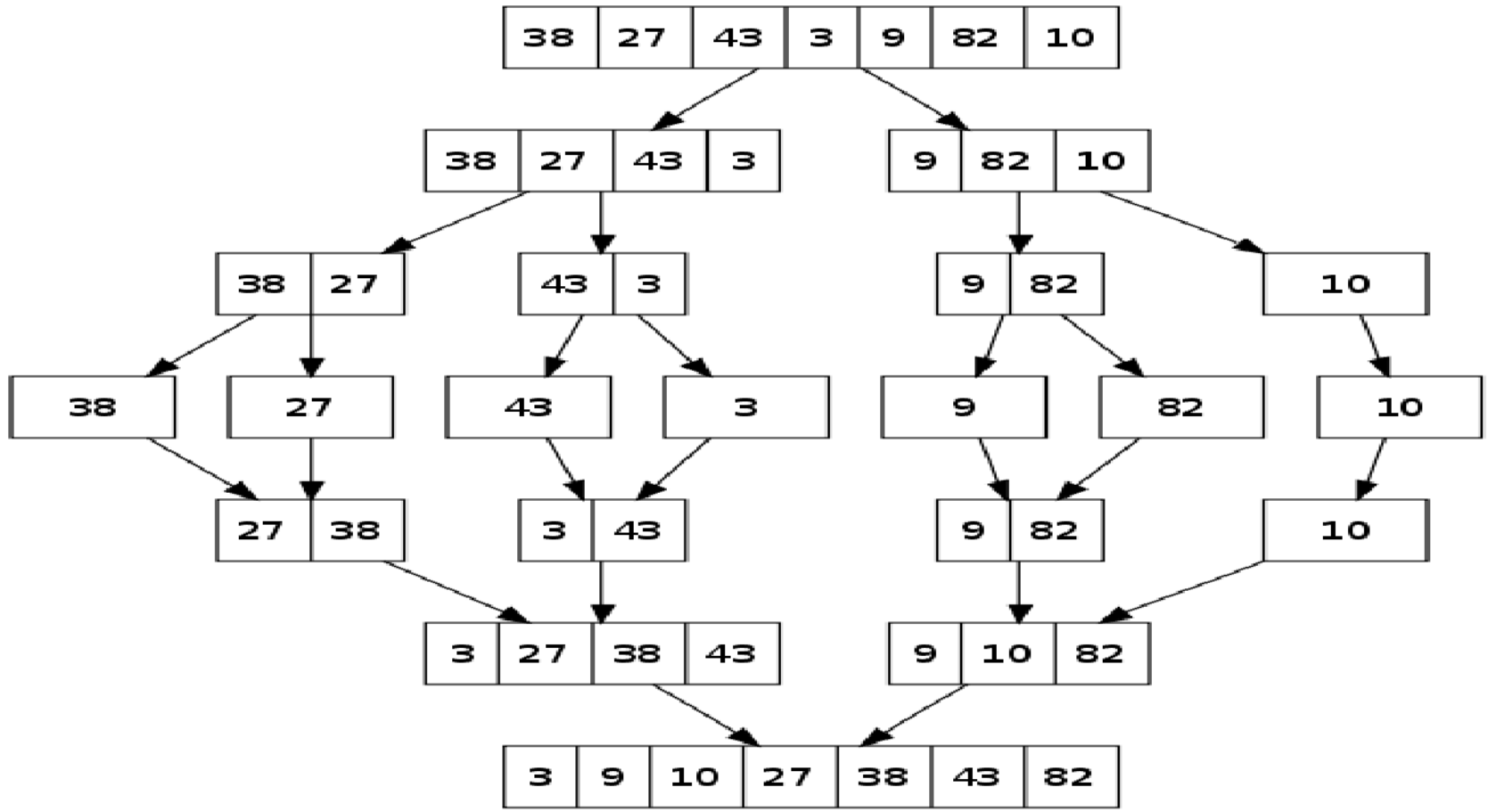
`mergeSort()` Algorithm

- Divide the given input array into two halves
- Sort each half recursively
- Merge the sorted arrays

mergeSort() Example

Divide

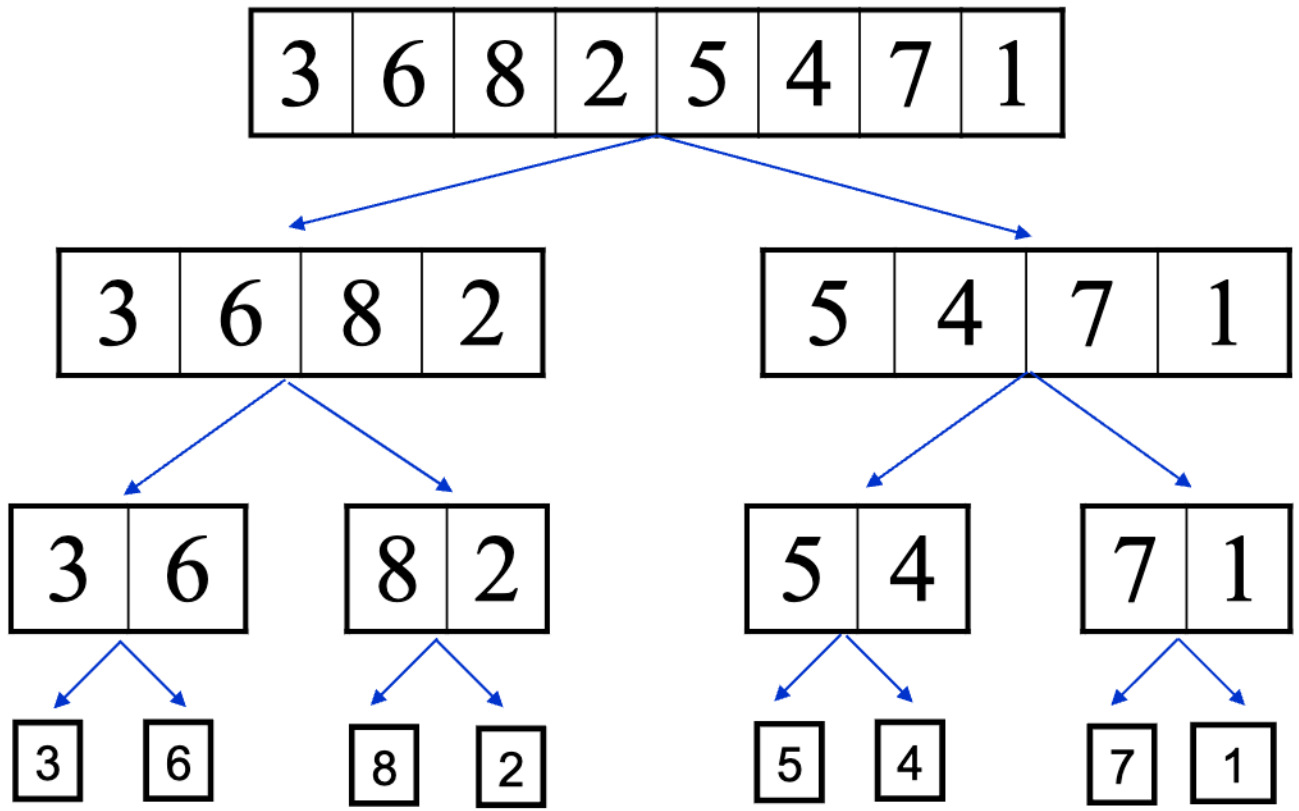
Merge



mergeSort() Implementation

```
public void mergeSort(int[] data, int leftIndex, int rightIndex){
      if (leftIndex < rightIndex){
            int middleIndex = (rightIndex + leftIndex) / 2;
            //sort first half
            mergeSort(data, leftIndex, middleIndex);
            //sort second half
            mergeSort(data, middleIndex + 1, rightIndex);
            //merge the two sorted halves
            merge(data, leftIndex, middleIndex, rightIndex);
      }
}
```

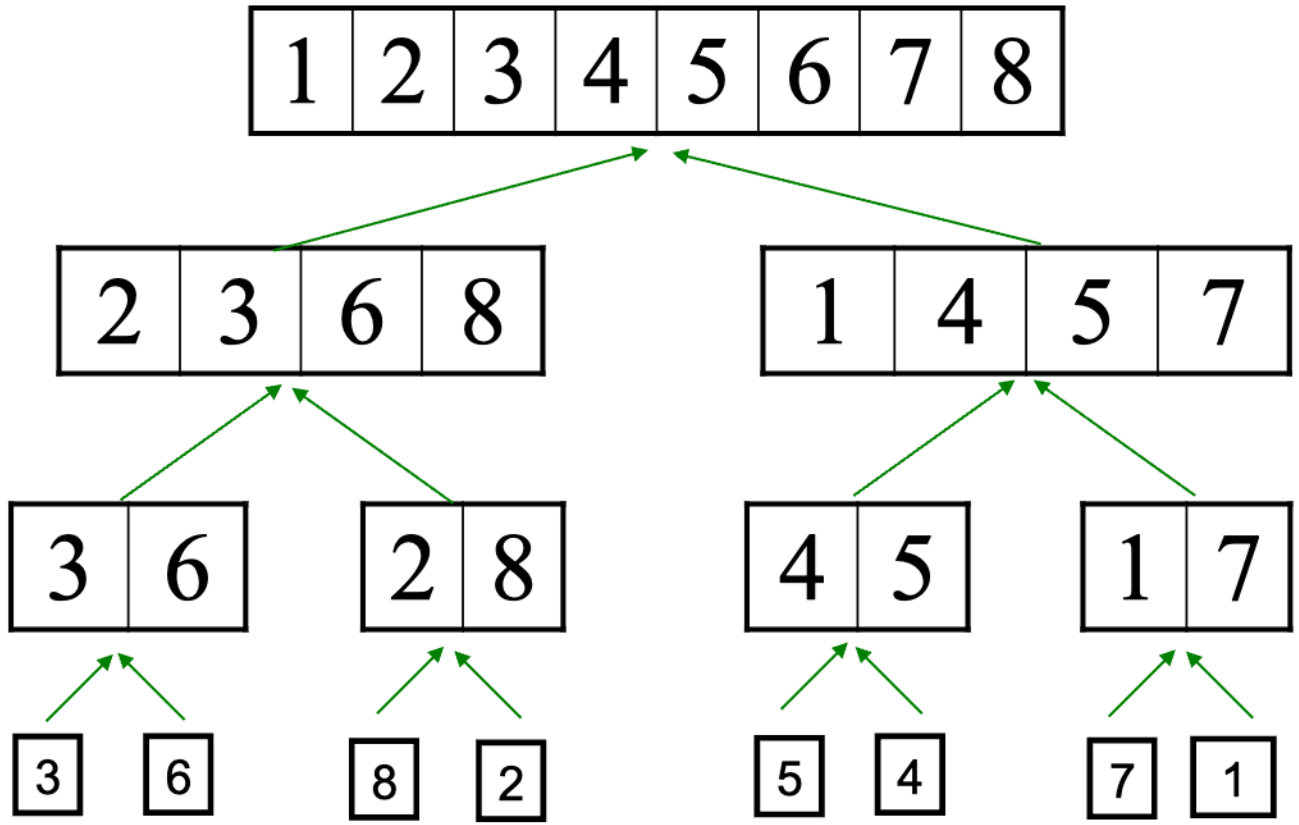
`mergeSort()` Decomposition



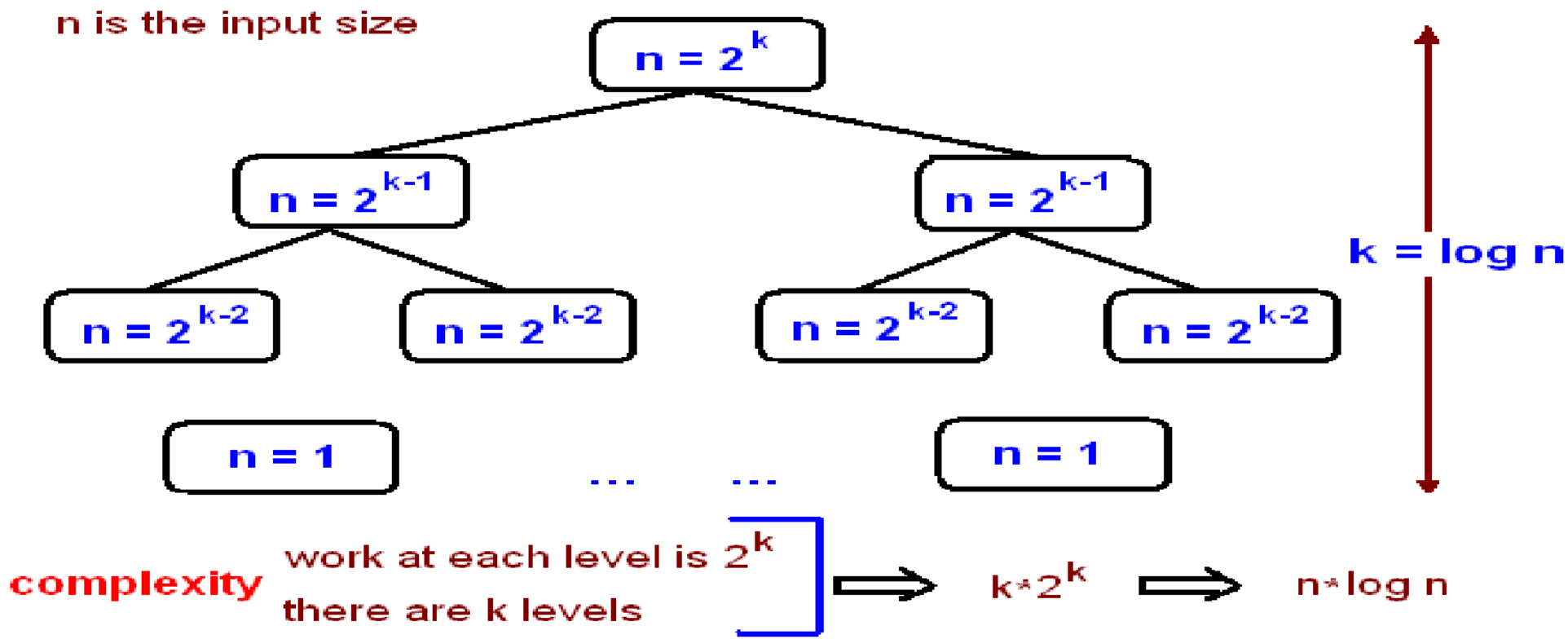
Implementation of the `merge()` Method

```
public void merge(int[] data, int leftIndex, int middleIndex, int rightIndex) {
          int leftSize = middleIndex – leftIndex + 1;
          int rightSize = rightIndex – middleIndex;

          int[] leftTemp = new int[leftSize + 1];                //create temp arrays to store data so it doesn't
          int[] rightTemp = new int[rightSize + 1];                                    get pushed out back

          leftTemp[leftSize] = Integer.Max_Value;                 //put huge value at the end of arrays
          rightTemp[rightSize] = Integer.Max_Value;

          for(int i = 0; i < leftSize; i++){                      //copy data to temp arrays
                    leftTemp[i] = data[leftIndex + i];
          }
          for(int i = 0; i < rightSize; i++){
                    rightTemp[i] = data[middleIndex + 1 + i];
          }

          int tempLeftIndex = 0;
          int tempRightIndex = 0;

          for(int i = 0; i < leftSize + rightSize; i++){
                    if(leftTemp[tempLeftIndex] < rightTemp[tempRightIndex]{
                              data[leftIndex + i] = leftTemp[tempLeftIndex++];
                    } else {
                              data[leftIndex + i] = rightTemp[tempRightIndex++];
                    }
          }

}
```

`mergeSort()` Example 2: Composition



Runtime Analysis for `mergeSort()`



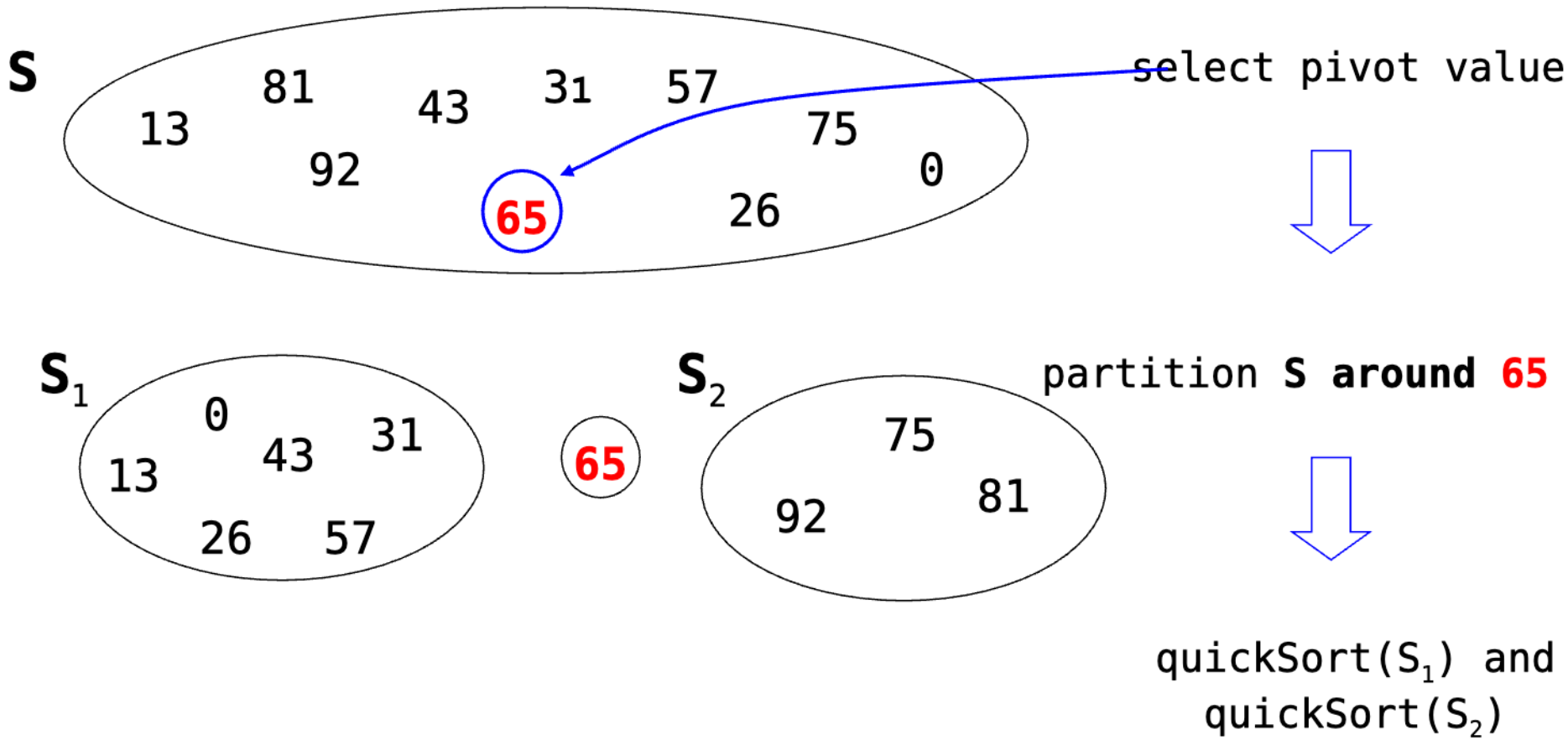
Both average and worst case time for mergeSort() are `O(n log n)`

Quick Sort

Quick Sort Algorithm

- Given an array of *n* elements:
  - If the array only contains one element
    - Return
  - Else
    - Pick one element to use as *pivot*
    - Partition elements into two sub-arrays around the pivot
      - Elements less than or equal to pivot
      - Elements greater than pivot
    - Quick sort the two sub-arrays using quick sort

Illustration of `quickSort()`



`quickSort()` Partitioning

- Partition the array into *left* and *right* sub-arrays
  - the elements in left sub-array are $\leq$ pivot
  - elements in right sub-array are > pivot

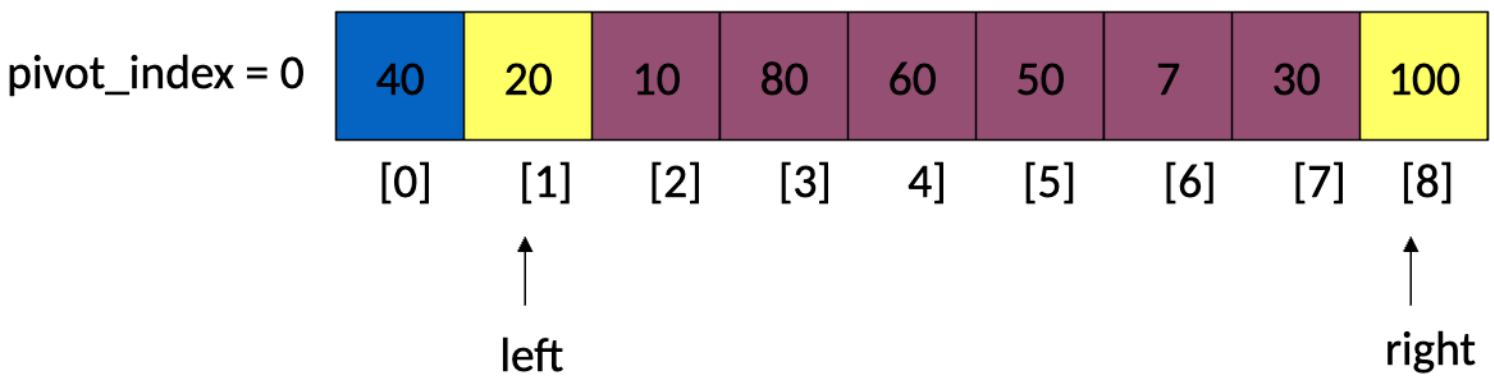
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in the correct partitions

`partition()` Algorithm

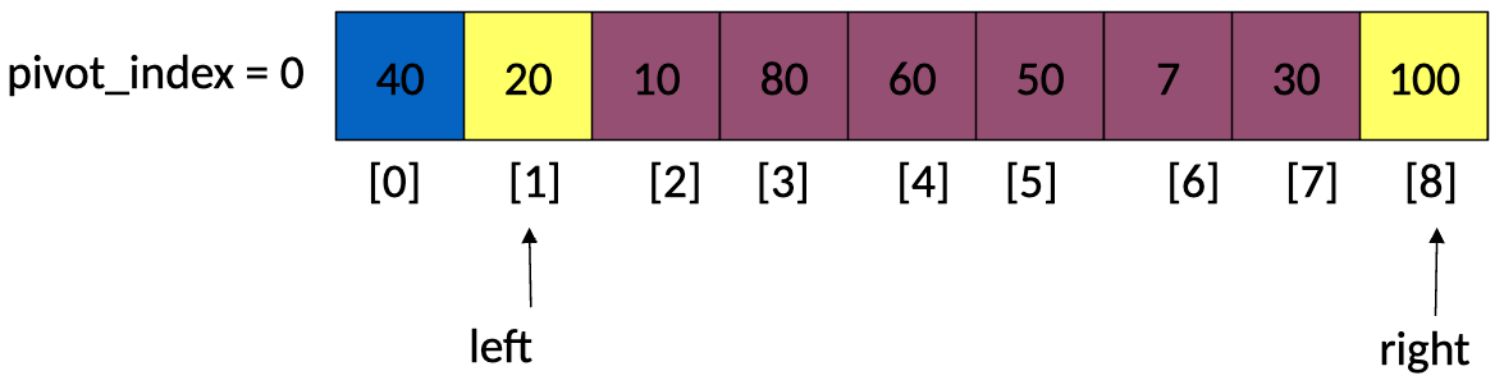
- Given a pivot, partition the elements of the array such that the resulting array consists of:
  1. One sub-array that contains elements <= pivot
  2. Another sub-array that contains elements > pivot

- The sub-arrays are stored in the original data array
- Partition algorithm loops through, swapping elements below/above pivot

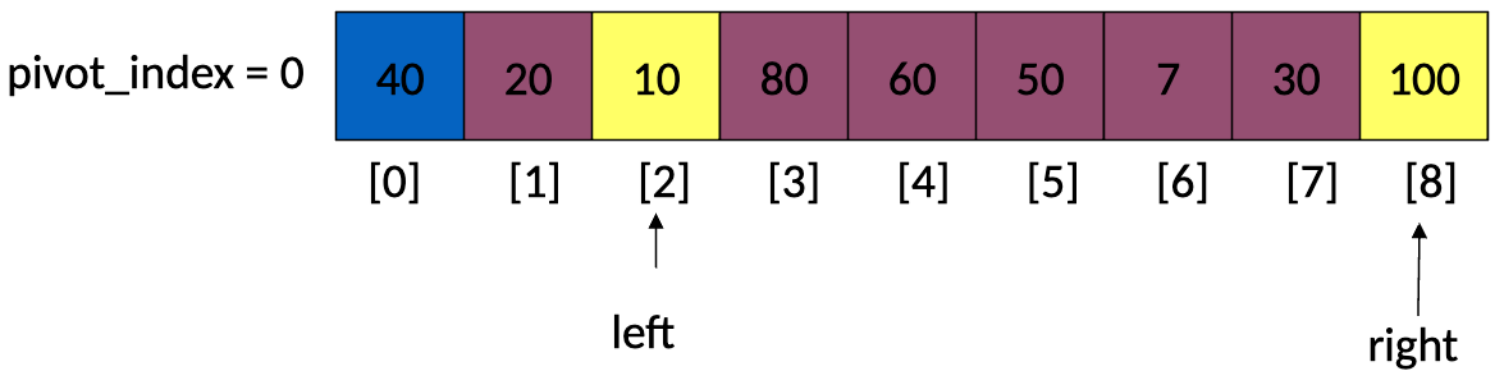
- Start with the following array of size 9
- Pick the first element (element at position) as the pivot
  - pivot = 40
  - pivot_index = 0
- left = 1
- right = data.length-1



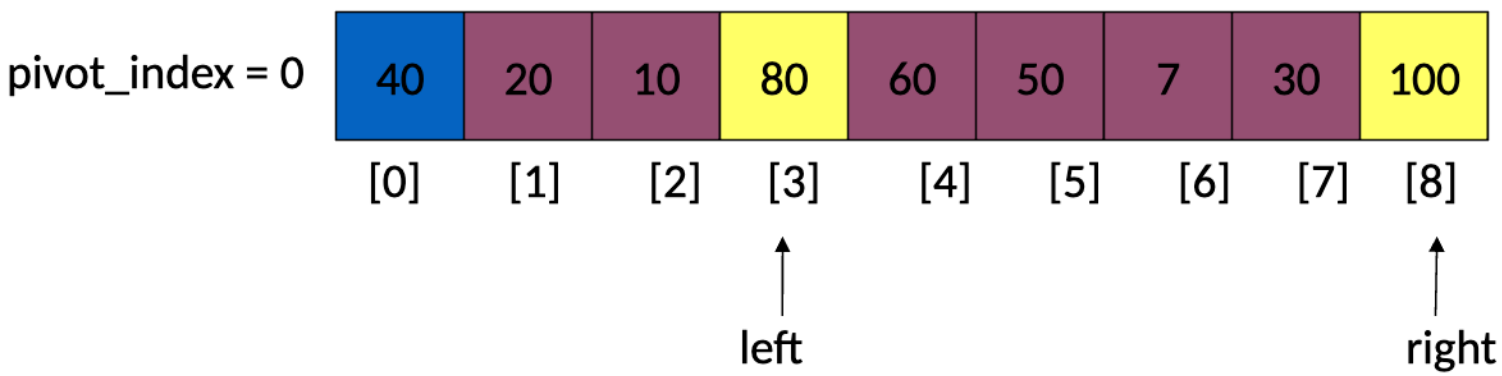
1. while data[left] <=
   data[pivot_index]
       left++



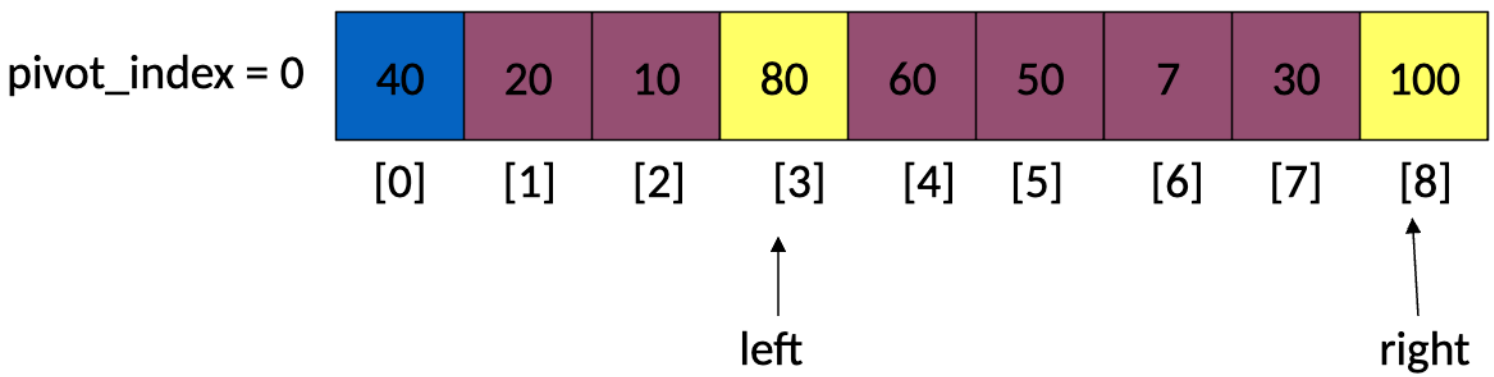
1. while data[left] <=
   data[pivot_index]
        left++



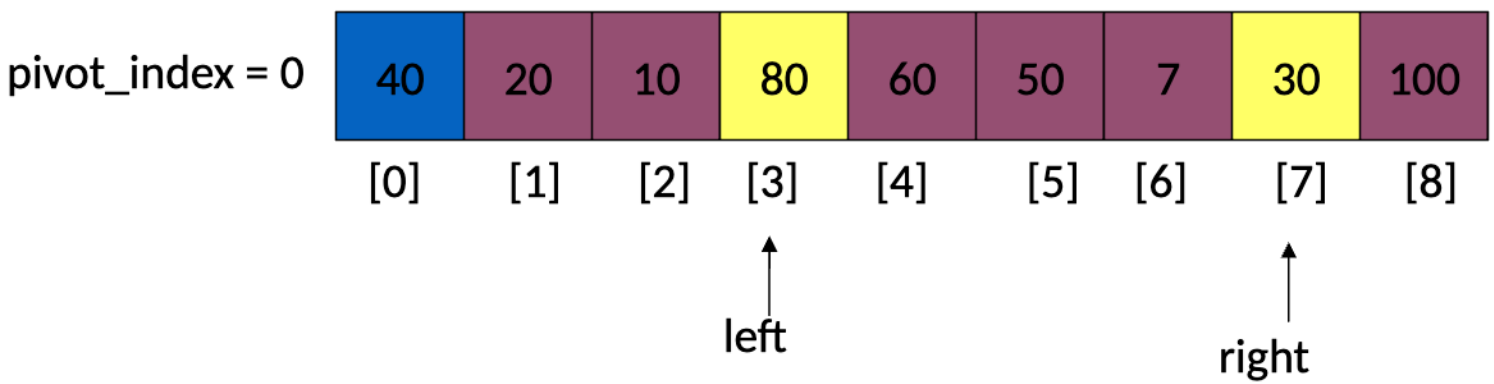
1. while data[left] <=
   data[pivot_index]
       left++



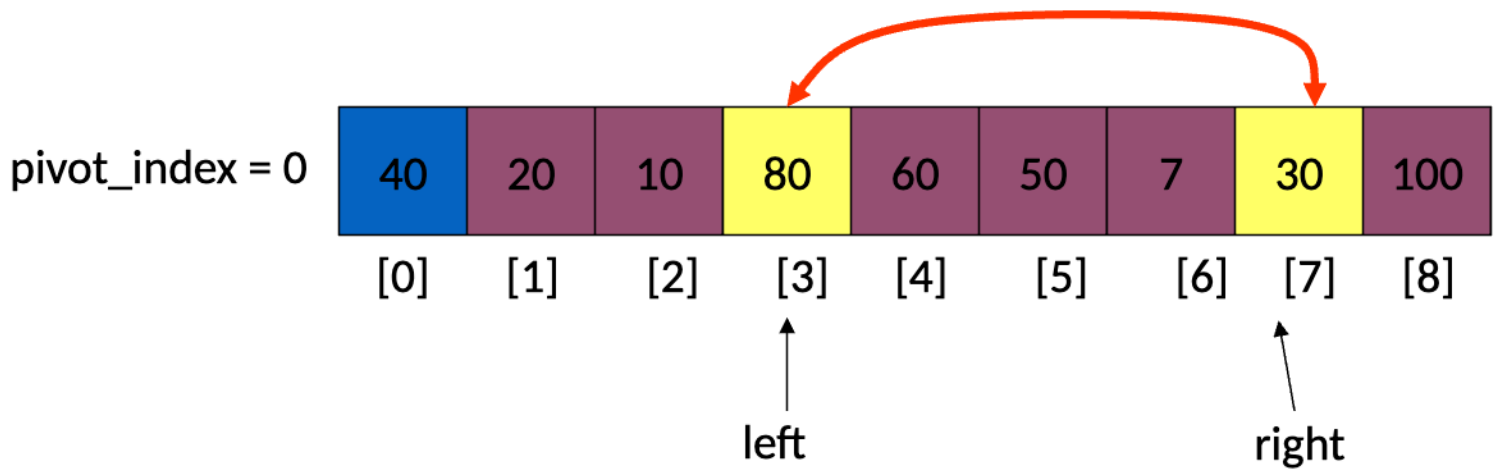
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
```



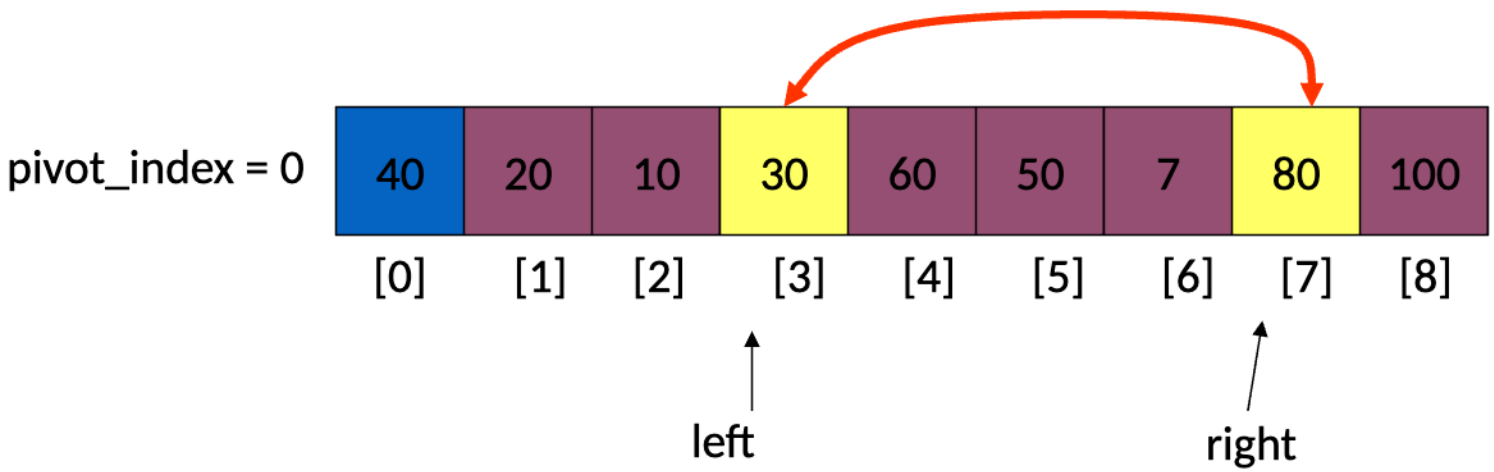
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
       right--



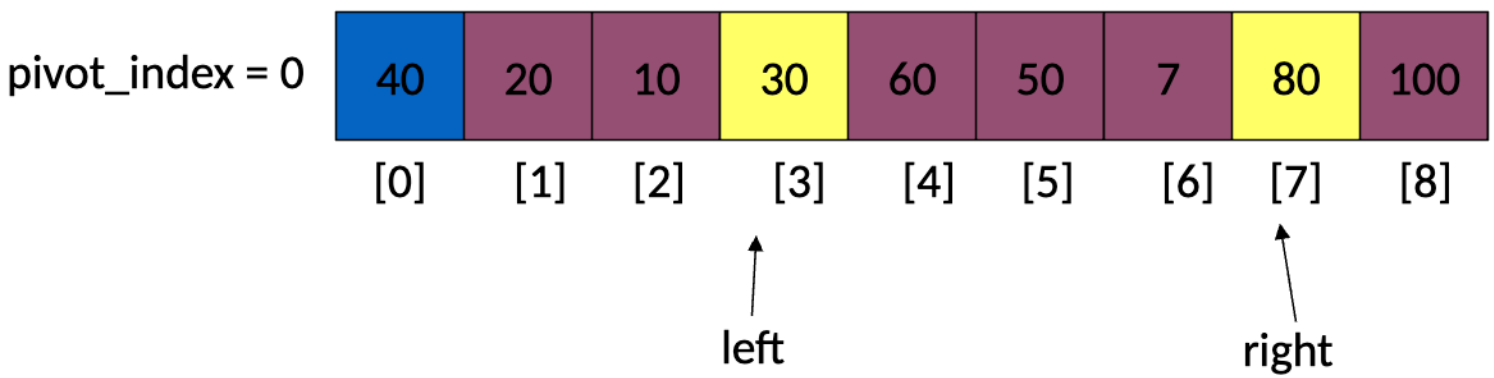
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
```



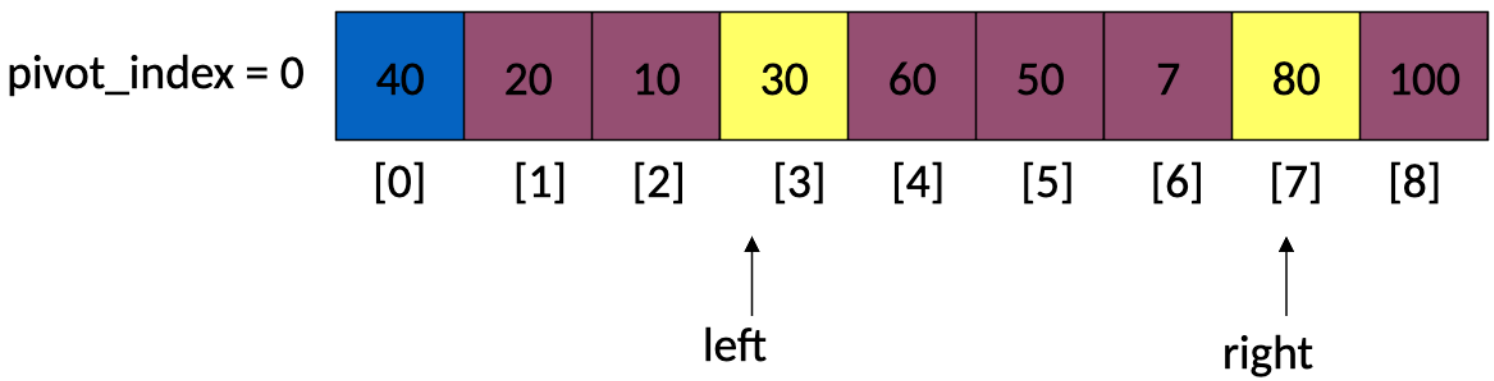
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. If left < right
     swap data[left] and data[right]
```



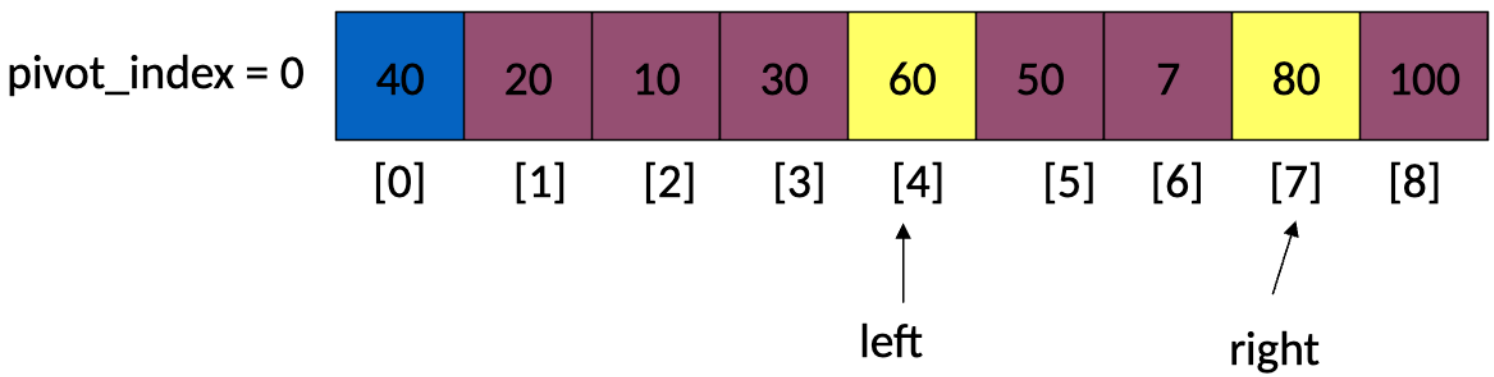
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. If left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



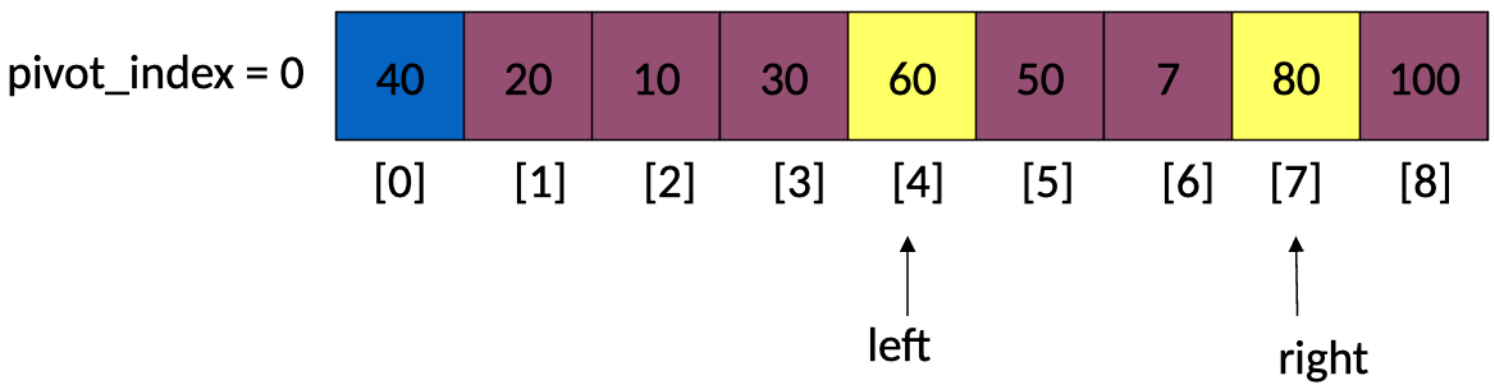
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. If left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



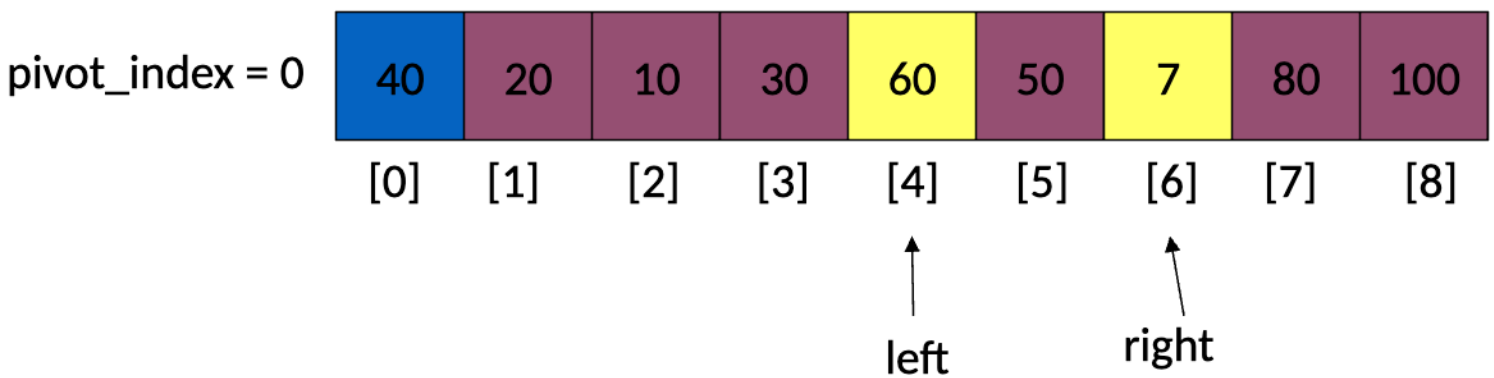
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. If left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



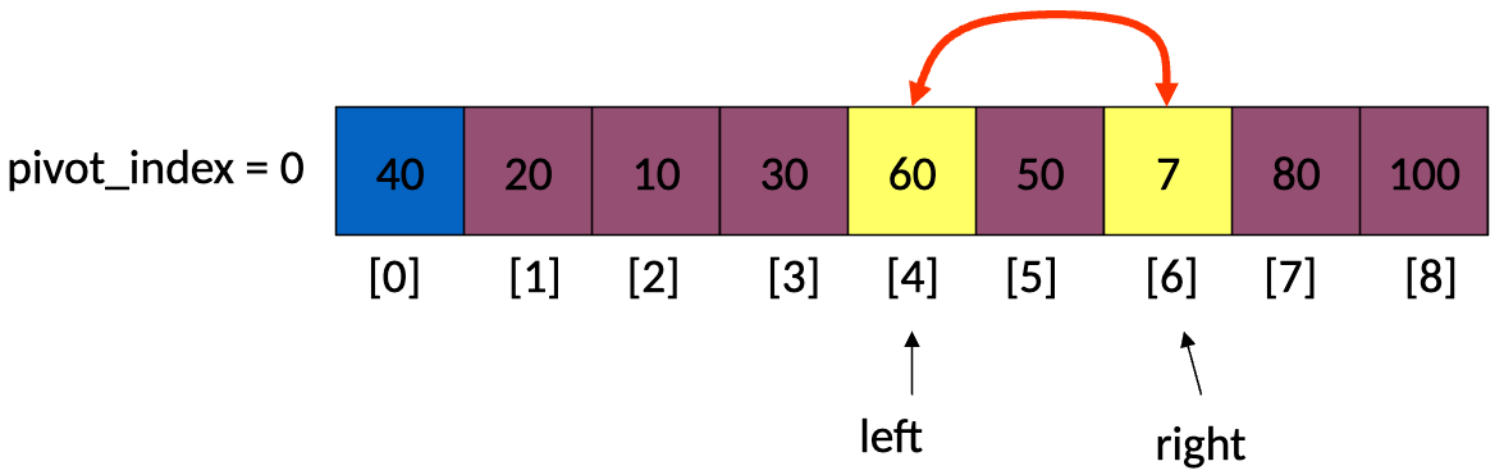
1. while data[left] <= data[pivot_index]
   left++
2. while data[right] > data[pivot_index]
   right--
3. if left < right
   swap data[left] and data[right]
4. while right > left, go to 1



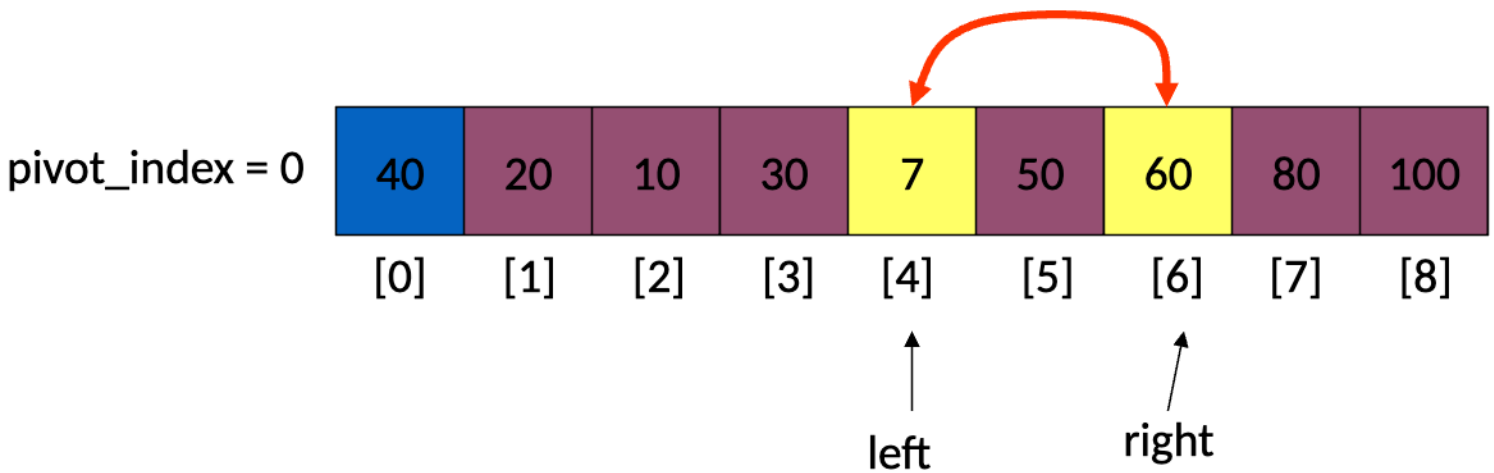
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



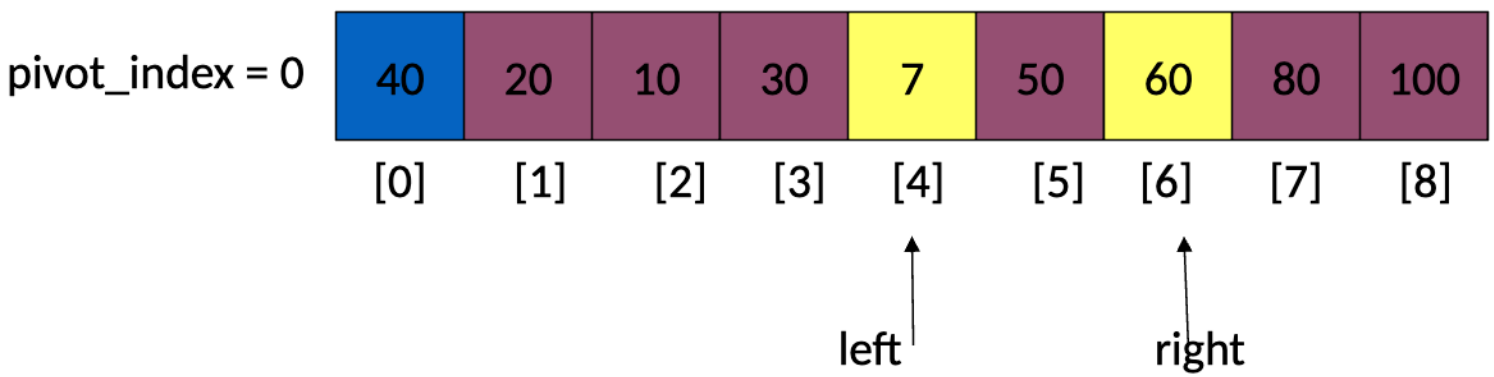
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



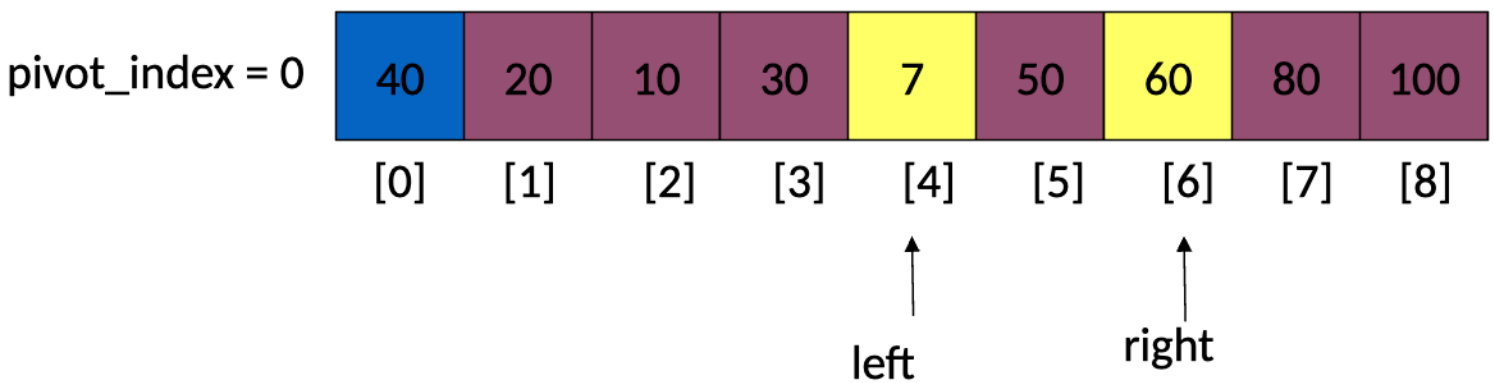
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



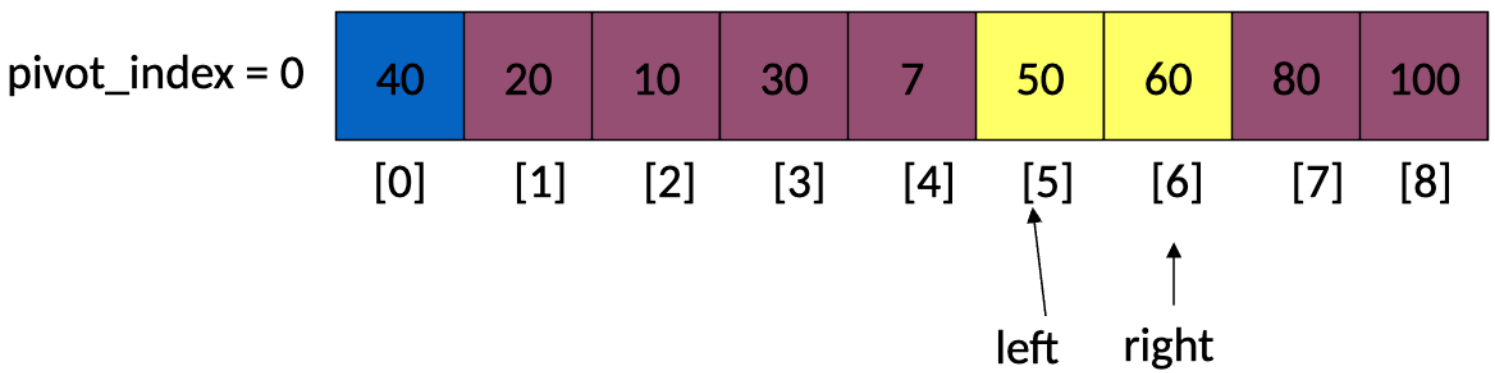
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



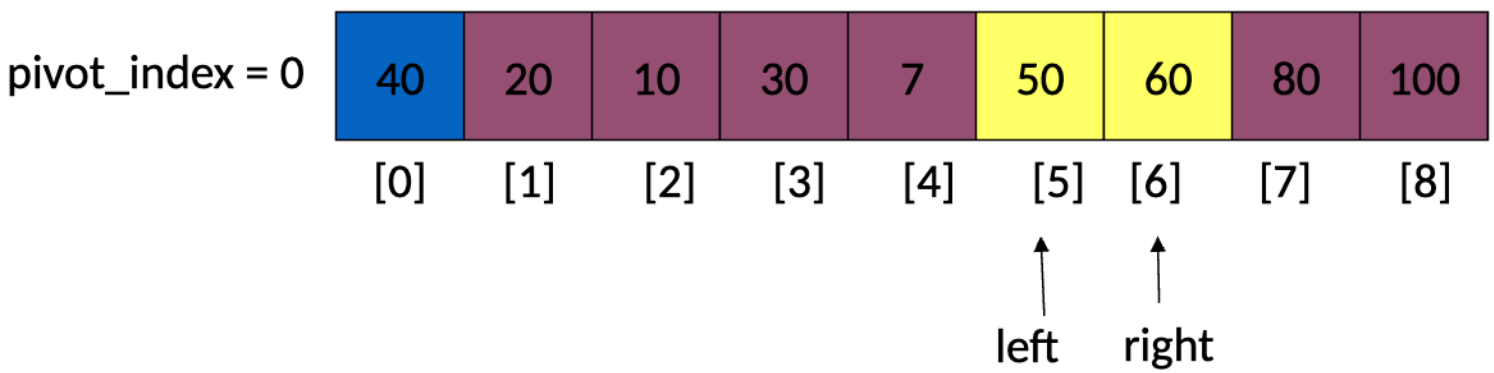
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



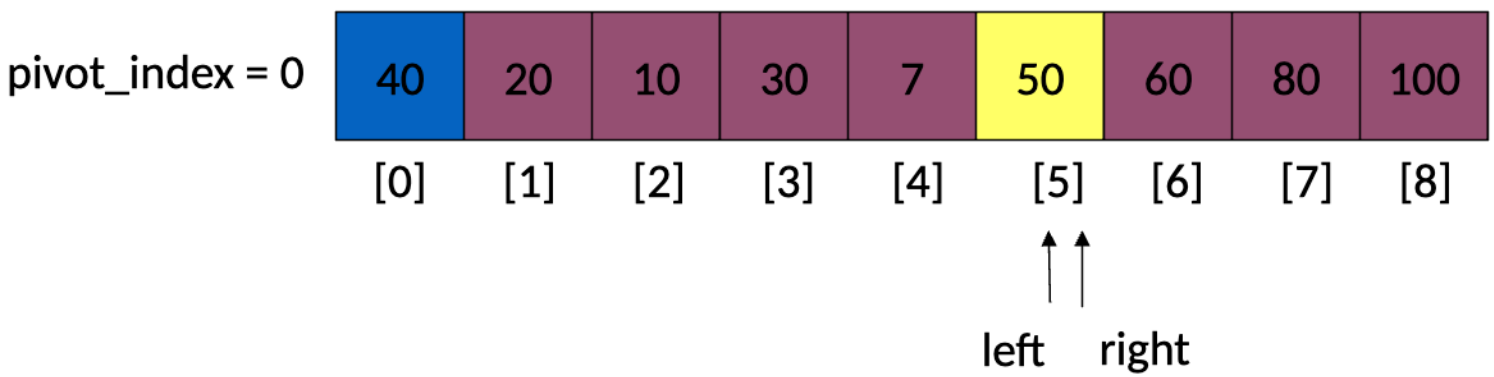
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



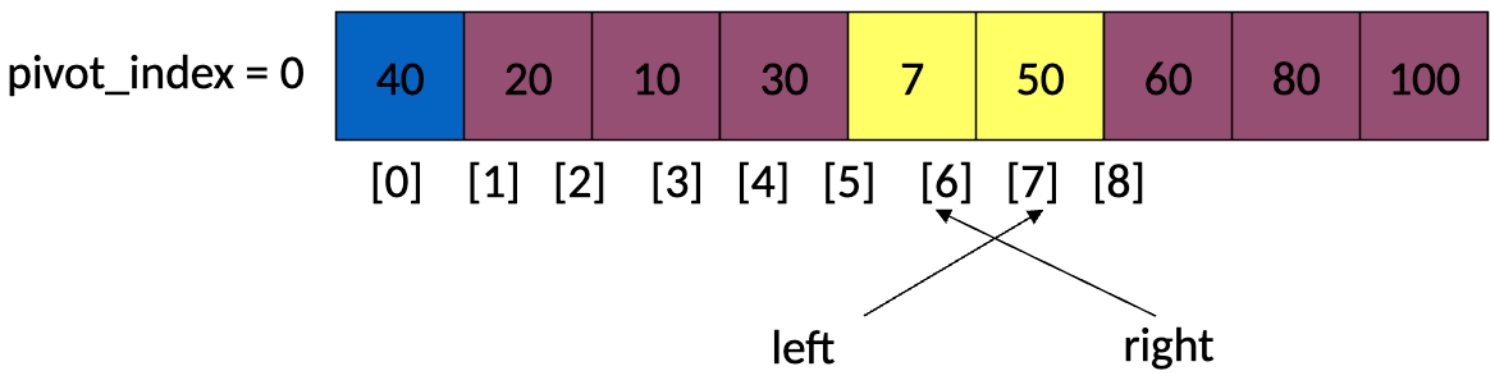
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



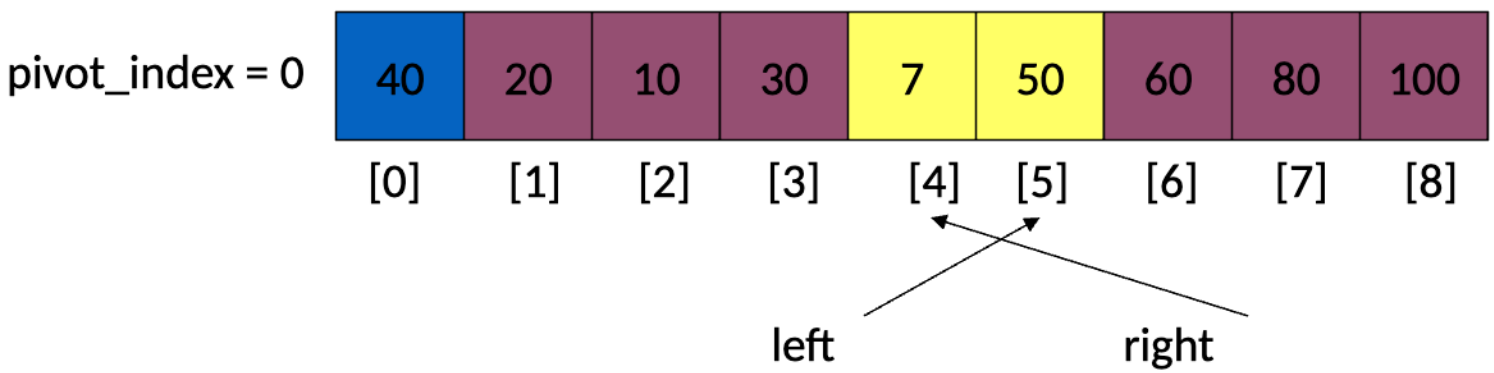
1. while data[left] <= data[pivot_index]
   left++
2. while data[right] > data[pivot_index]
   right--
3. if left < right
   swap data[left] and data[right]
4. while right > left, go to 1



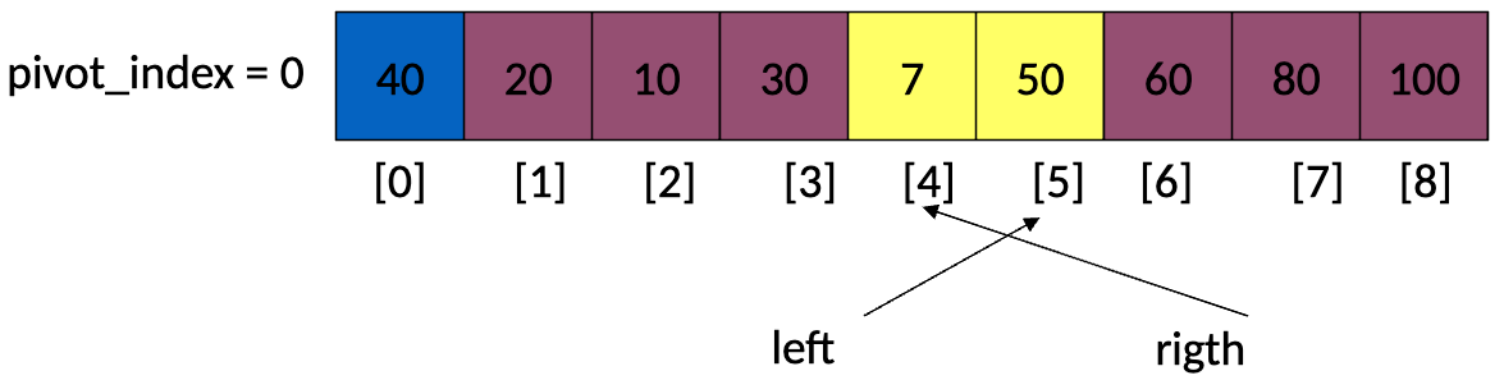
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



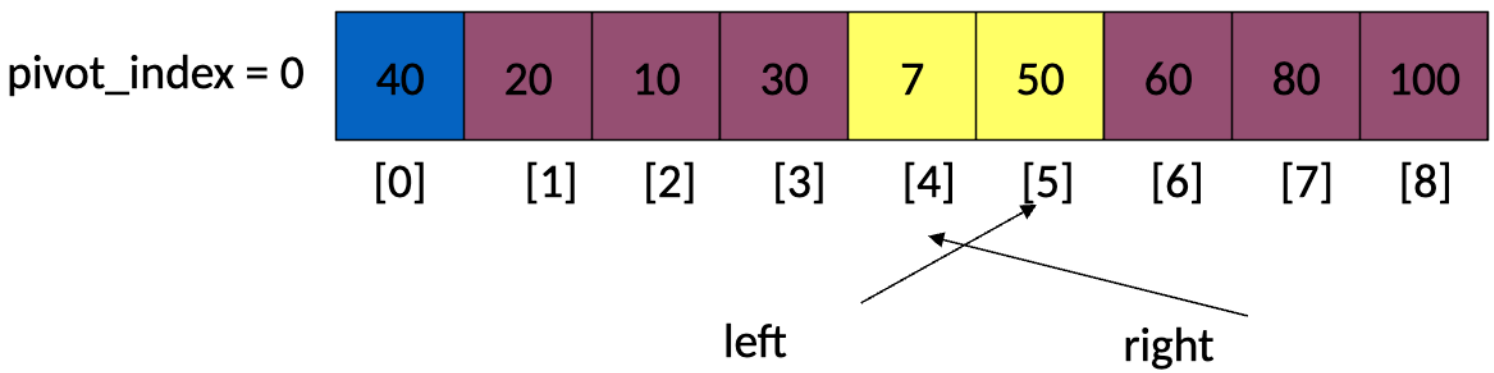
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



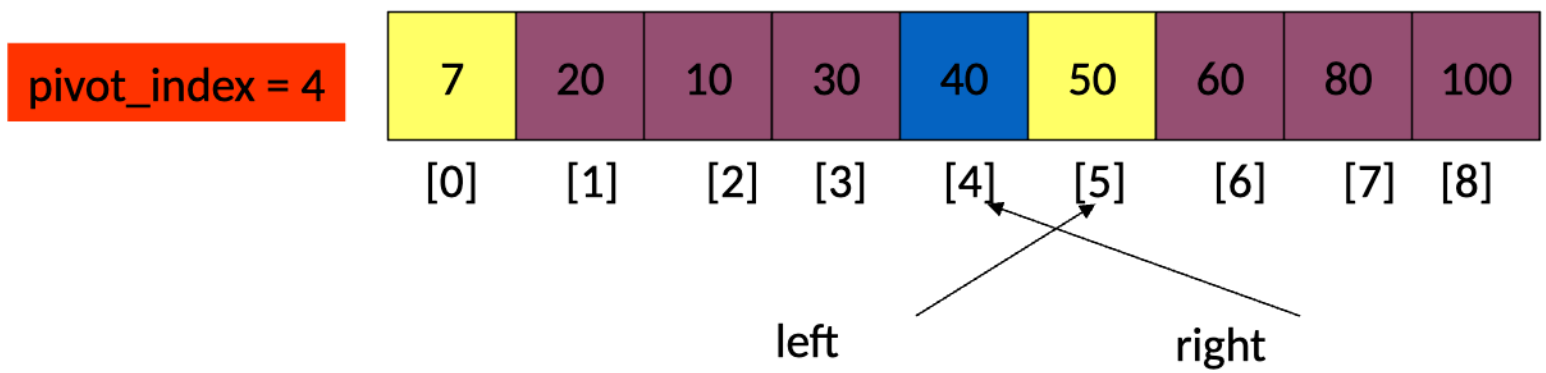
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
```



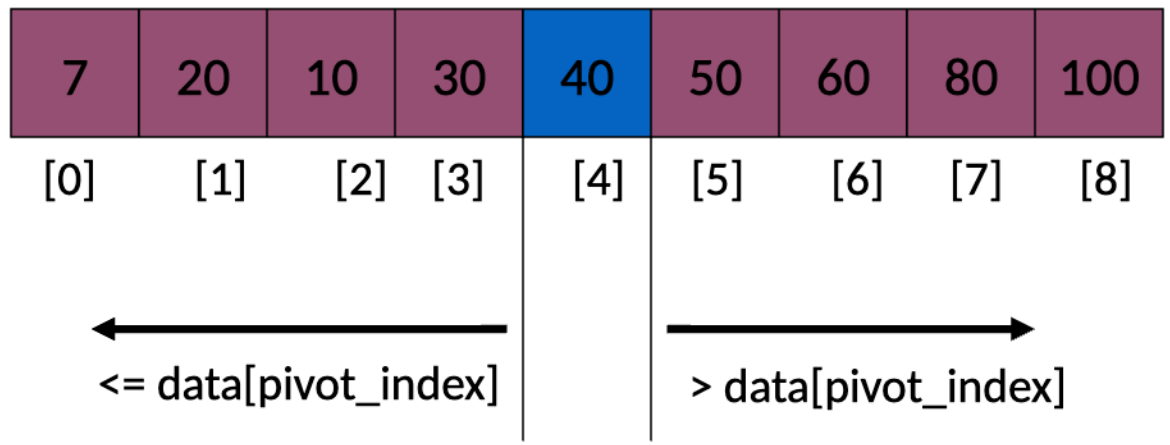
```
1. while data[left] <= data[pivot_index]
       left++
2. while data[right] > data[pivot_index]
     right--
3. if left < right
     swap data[left] and data[right]
4. while right > left, go to 1
5. swap data[right] and data[pivot_index]
```



```
1.  while data[left] <= data[pivot_index]
        left++
2. while data[right] > data[pivot_index]
    right--
3. if left < right
    swap data[left] and data[right]
4. while right > left, go to 1
5. swap data[right] and data[pivot_index]
```



Array After Partitioning



quickSort() Algorithm

```
public static void quickSort(int[] data, int firstIndex, int
lastIndex) {

   if (firstIndex < lastIndex) {
      int pivotIndex = partition(data, firstIndex, lastIndex);

      //recursive calls to sort the two pieces
      quickSort(data, firstIndex, pivotIndex - 1);
      quickSort(data, pivotIndex + 1, lastIndex);
   }
}
```

Runtime Analysis for `quickSort()`

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

  1. Depth of recursion tree? **$O(\log_2 n)$**
  2. Number of accesses in partition? **O(n)**

`quickSort()`'s <u>best case</u> running time is $O(n \log_2 n)$ when the array elements are randomly distributed (and hence the pivot will divide the array into two (almost) equal-sized partitions)

`quicksort()`: **Worst Case** running time

- If the input array is already sorted.

- Then the pivot partitions the array in two sub-arrays such that
  - one sub-array is of size 0
  - the other sub-array is of size n-1

- Depth of recursion tree? **O(n)**
- Number of accesses per partition? **O(n)**

`quickSort()`'s worst case running time $O(n^2)$ when the input array is already sorted

Runtime Comparisons

`mergeSort()` vs. `quickSort()`

- Both `quickSort()` and `mergeSort()` have an expected time of O(n log n) to sort a list containing n items
- However, in the worst case, `quickSort()` can require $O(n^2)$ time, while `mergeSort()`'s worst case running time is the same as its expected time O(n log n).
- Given that `mergeSort()`'s worst case time is better than `quickSort()`'s, why bother ever to use `quickSort()` at all? Why is `quickSort()` so commonly used in practice?
  - Because it does not require the additional space that is needed by `mergeSort()`

Runtime Analysis of Sorting Algorithms

Algorithm	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	**$O(n)$** when the array is already sorted	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n \log n)$	**$O(n^2)$** When the array is already sorted
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$