Sorting Algorithms

Sorting

- ***Sorting*** : is a process that organizes a collection of data into either ascending or descending order.

- **There are two type of sorting:**

1- internal sort requires that the collection of data fit entirely in the computer's main memory.

2- external sort when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.

- **There are many sorting algorithms, such as:**
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort

Selection Sort

- The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of *n* elements requires *n-1* passes to completely rearrange the data.

Sorted **Unsorted**

23	78	45	8	32	56	Original List
8	78	45	23	32	56	After pass 1
8	23	45	78	32	56	After pass 2
8	23	32	78	45	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

Selection Sort (cont.)

```
template <class Item>
void selectionSort( Item a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
       if (a[j] < a[min]) min = j;
    swap(a[i], a[min]);
  }
}

template < class Object>
void swap( Object &lhs, Object &rhs )
{
   Object tmp = lhs;
   lhs = rhs;
   rhs = tmp;
}
```

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.

- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

Sorted **Unsorted**

23	78	45	8	32	56	Original List
23	78	45	8	32	56	After pass 1
23	45	78	8	32	56	After pass 2
8	23	45	78	32	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

Insertion Sort Algorithm

```
template <class Item>
void insertionSort(Item a[], int n)
{
   for (int i = 1; i < n; i++)
   {
      Item tmp = a[i];

      for (int j=i; j>0 && tmp < a[j-1]; j--)
         a[j] = a[j-1];
      a[j] = tmp;
   }
}
```

Bubble Sort

- The list is divided into two sublists: sorted and unsorted.
- The smallest element is bubbled from the unsorted list and moved to the sorted sublist.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

Bubble Sort

23	78	45	8	32	56	Original List
8	23	78	45	32	56	After pass 1
8	23	32	78	45	56	After pass 2
8	23	32	45	78	56	After pass 3
8	23	32	45	56	78	After pass 4

Bubble Sort Algorithm

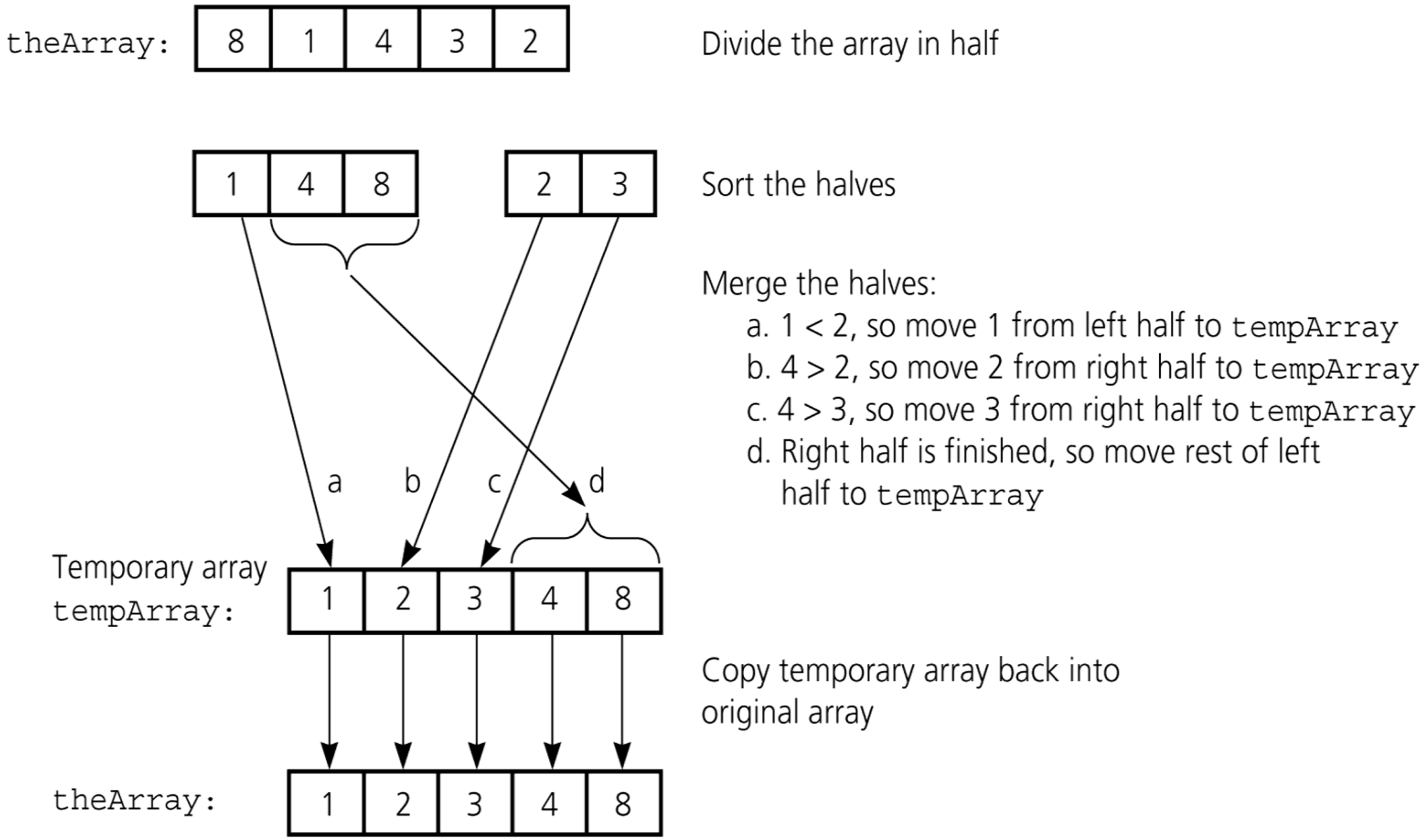
```
template <class Item>
void bubleSort(Item a[], int n)
{
   bool sorted = false;
   int last = n-1;

   for (int i = 0; (i < last) && !sorted; i++){
      sorted = true;
      for (int j=last; j > i; j--)
         if (a[j-1] > a[j]{
            swap(a[j],a[j-1]);
            sorted = false; // signal exchange
         }
    }
}
```

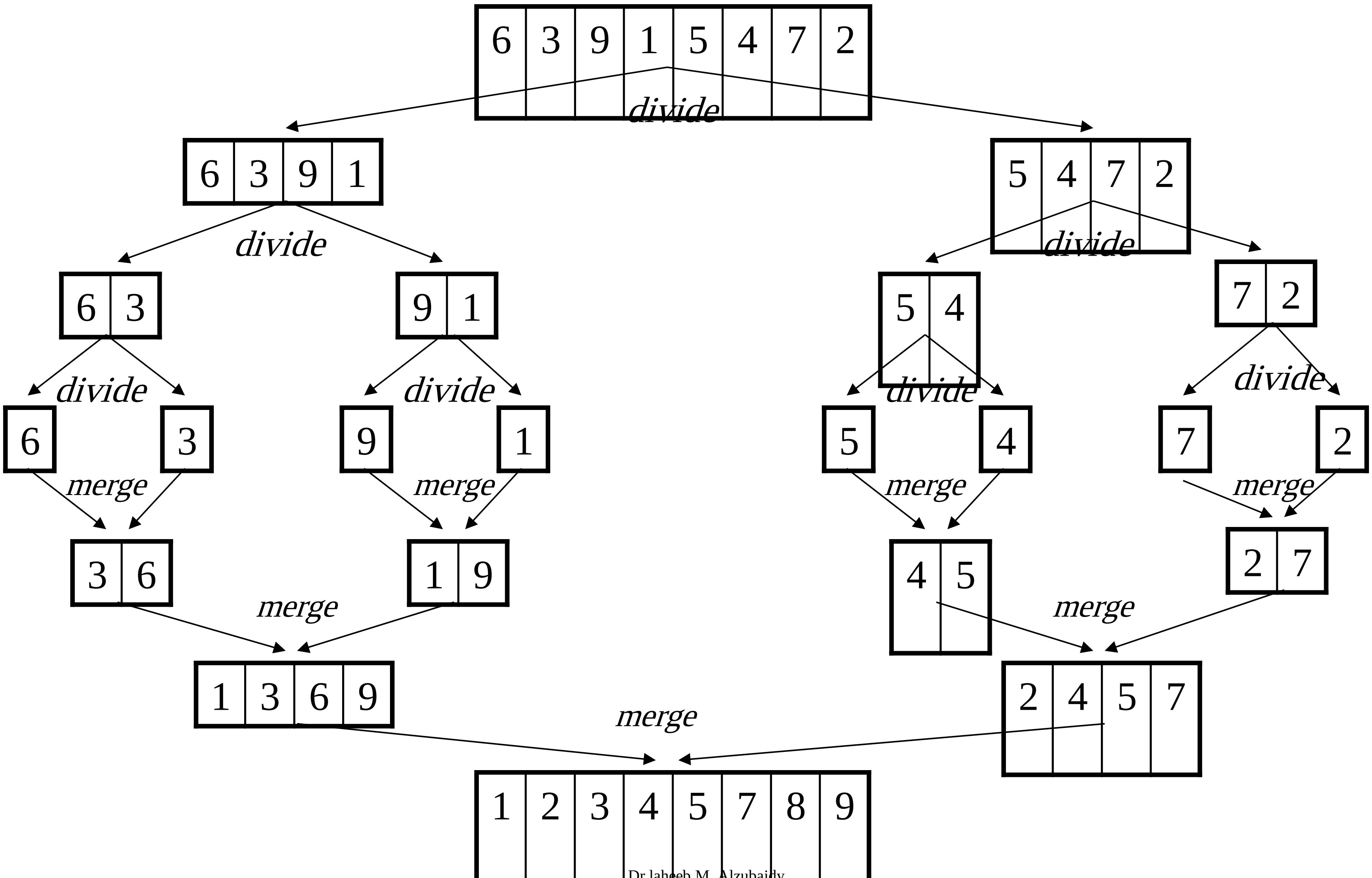
Mergesort

- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

Mergesort - Example



Mergesort - Example



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
   }
}
```

Quicksort Algorithm

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

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

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

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

40	20	10	80	60	50	7	30	100

Partitioning Array

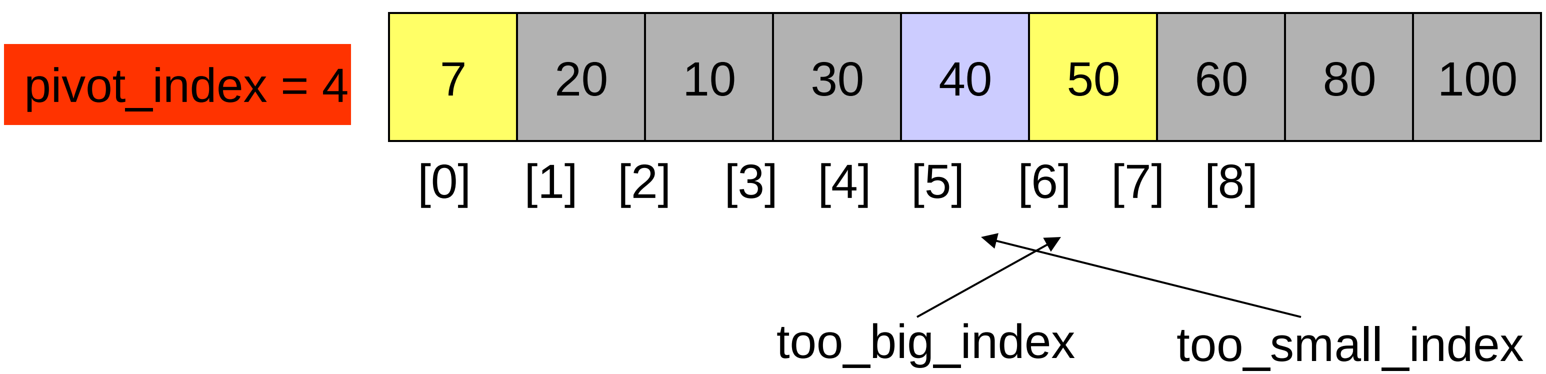
Given a pivot, partition the elements of the array such that the resulting array consists of:

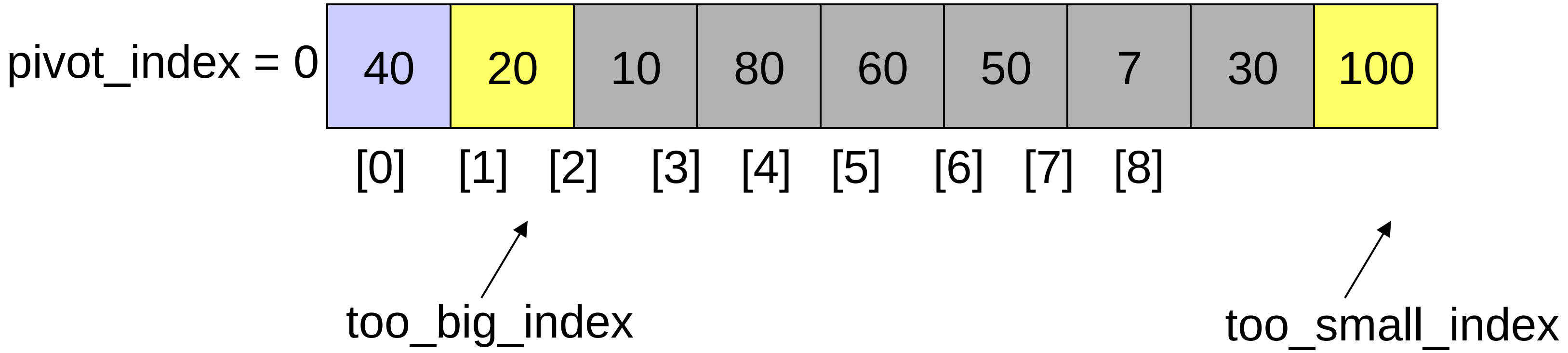
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

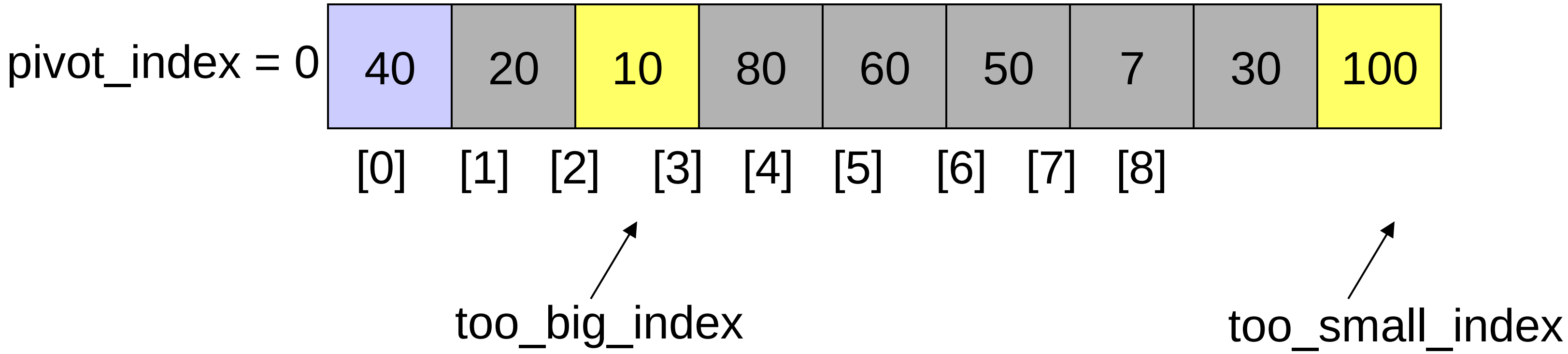
Partitioning loops through, swapping elements below/above pivot.

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

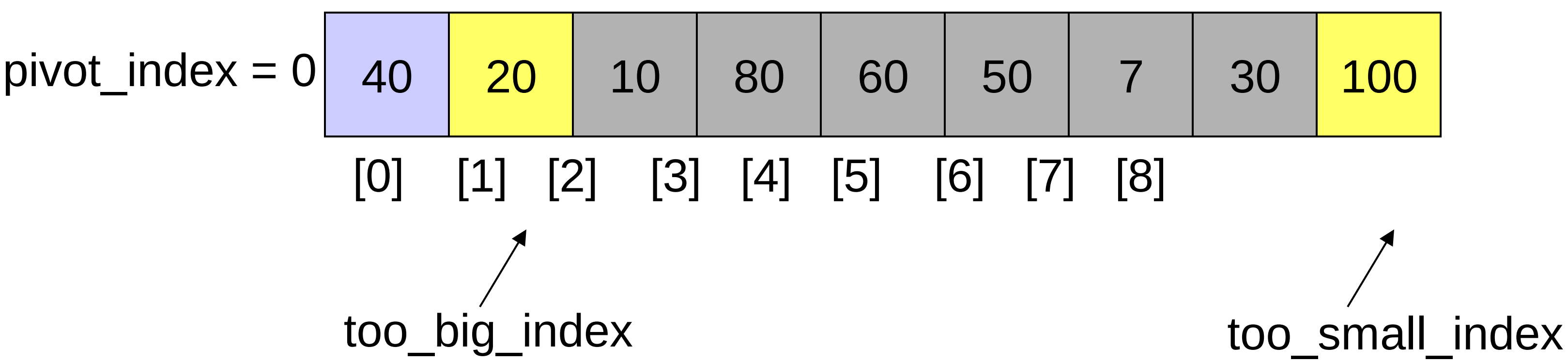



pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

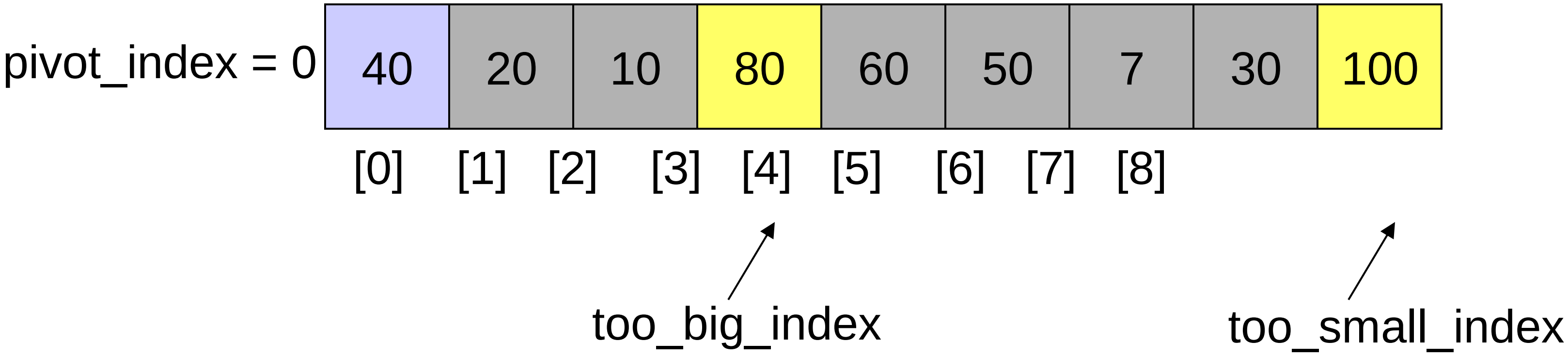
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



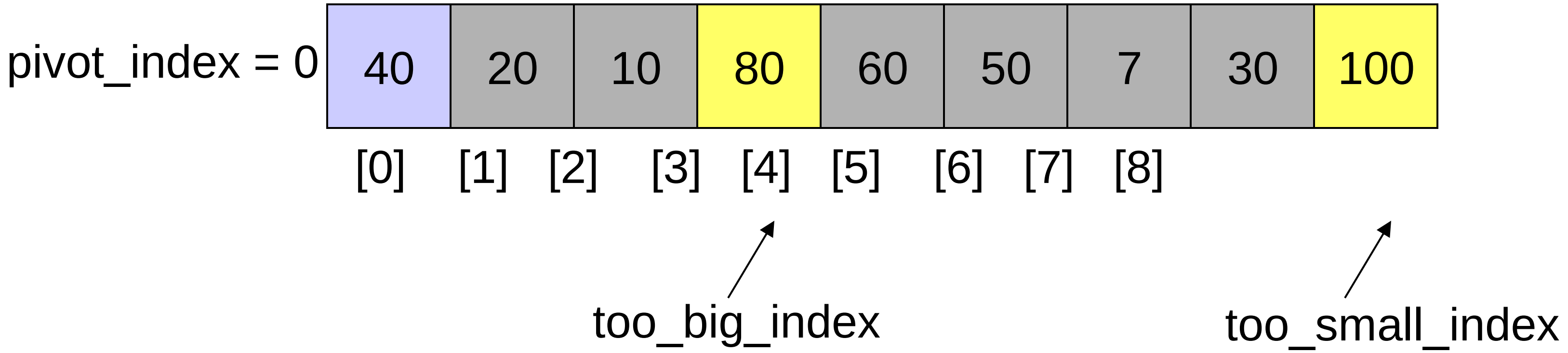
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



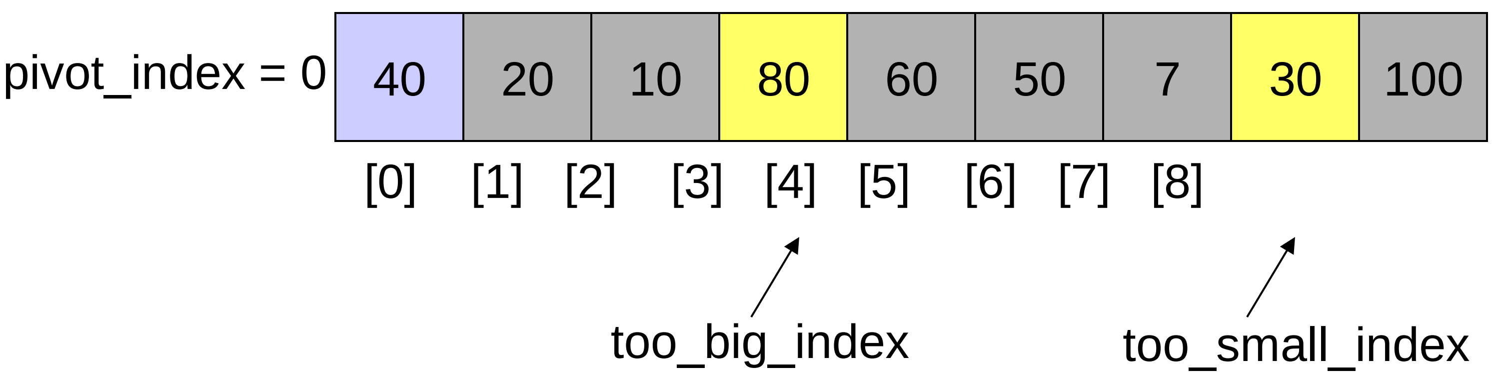
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



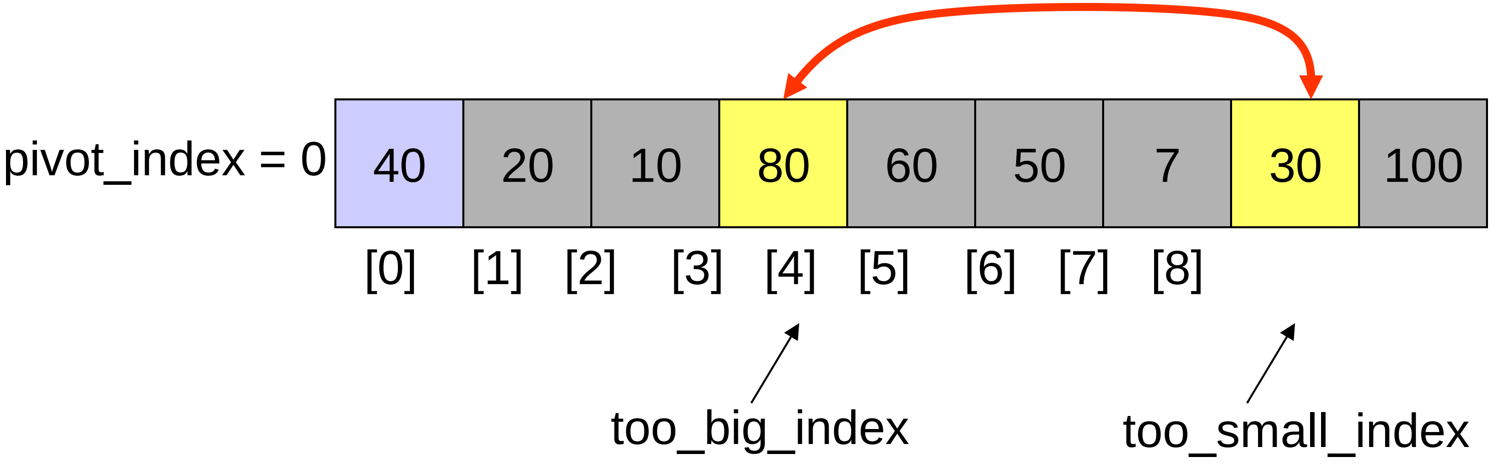
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



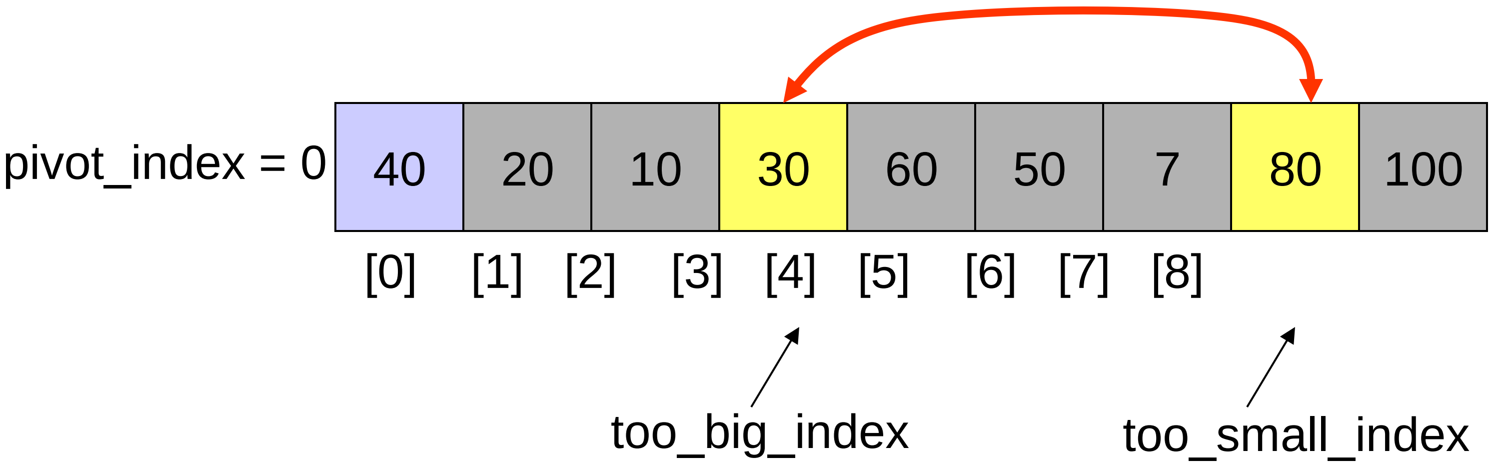
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



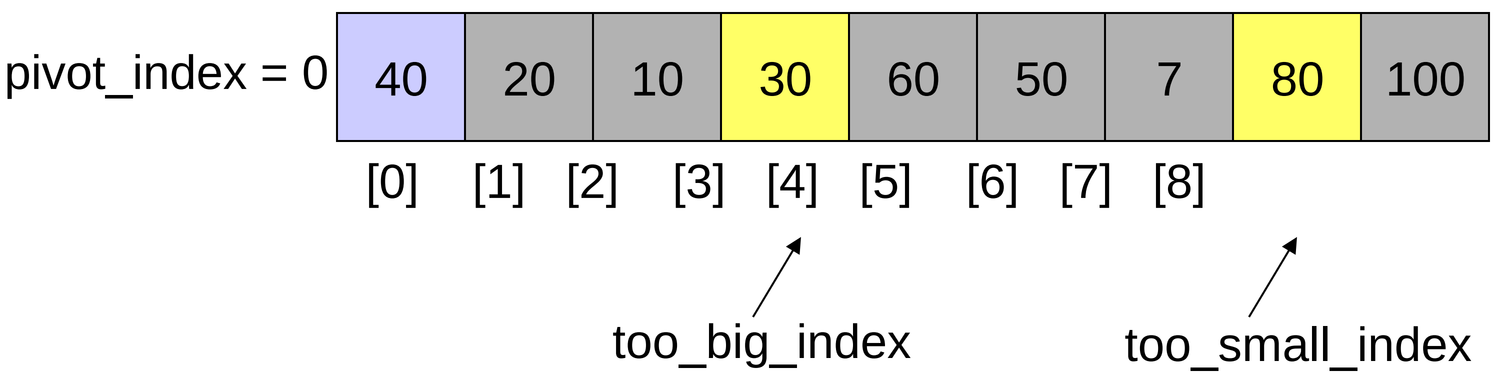
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



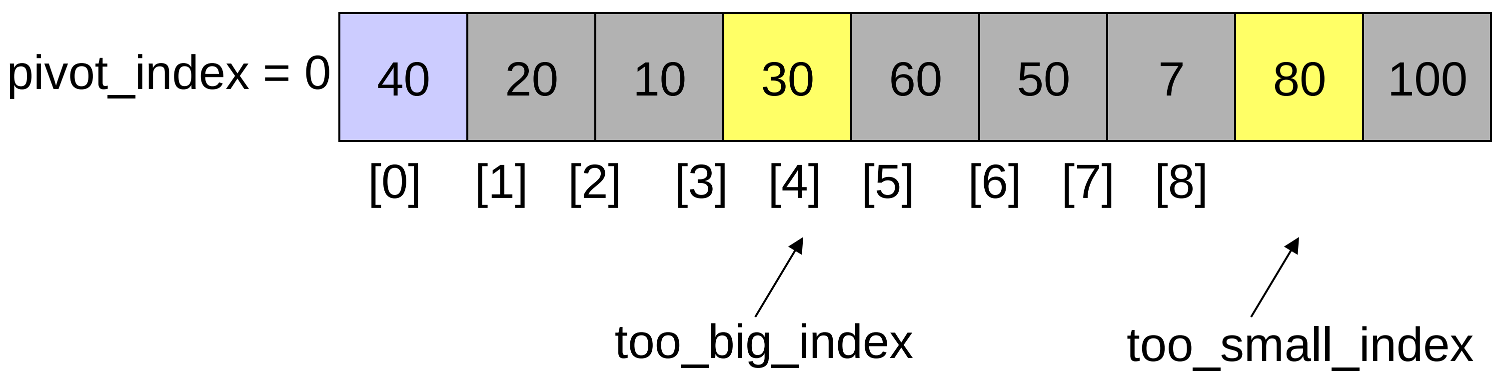
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



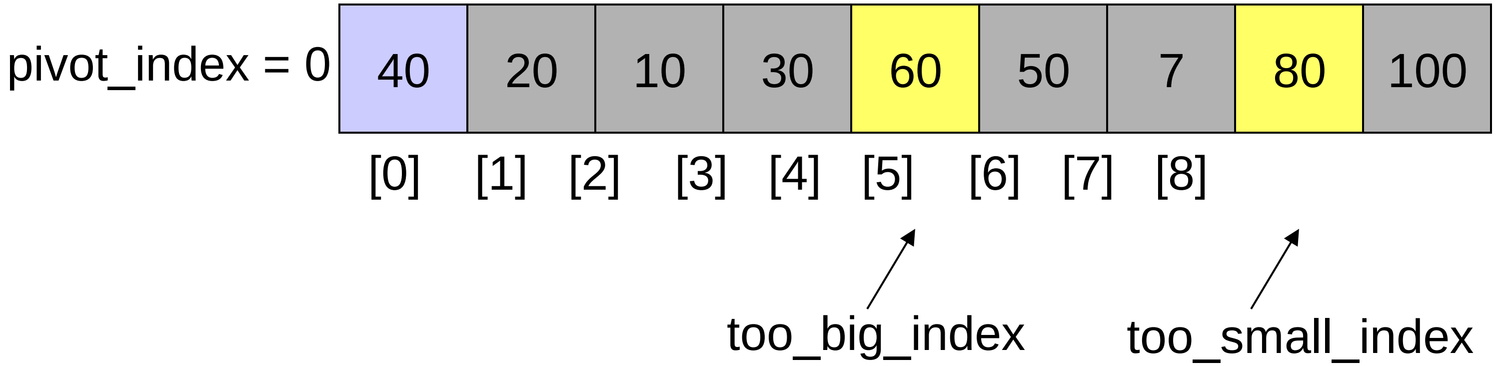
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



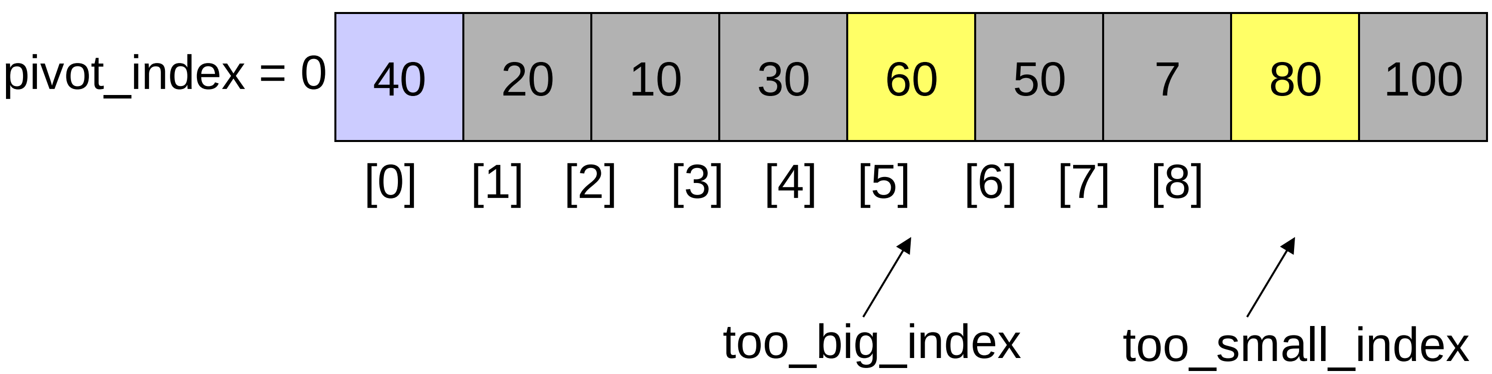
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



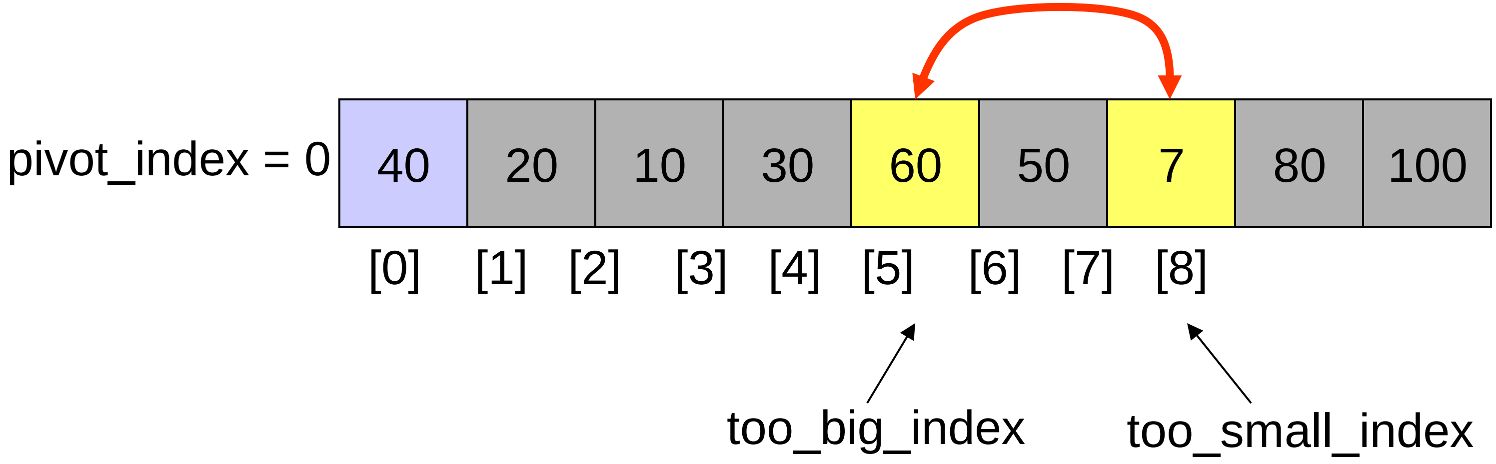
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



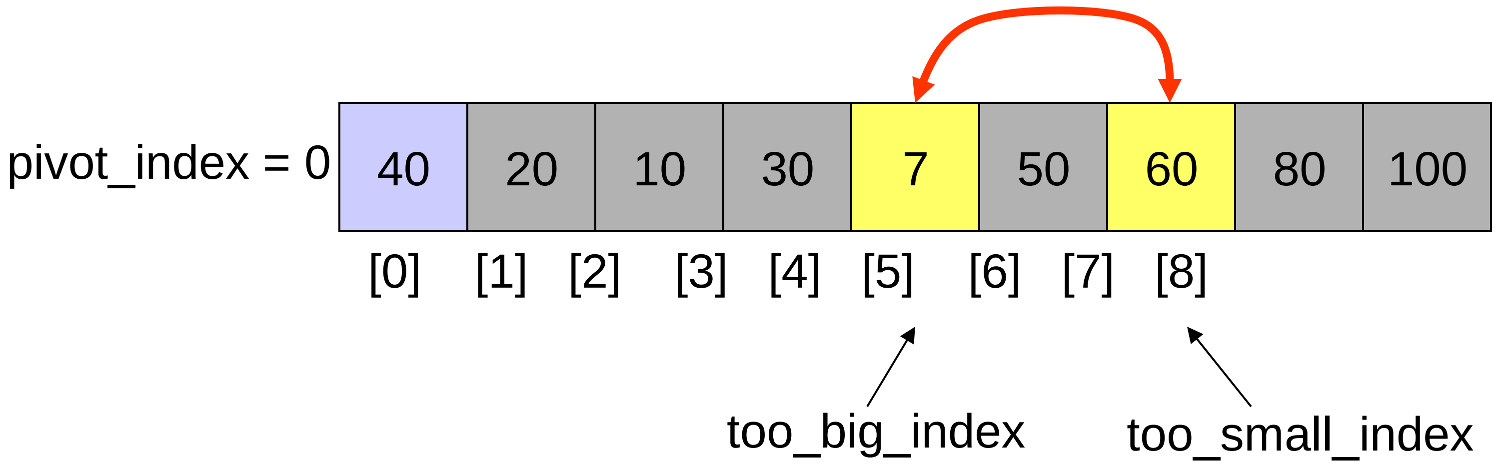
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



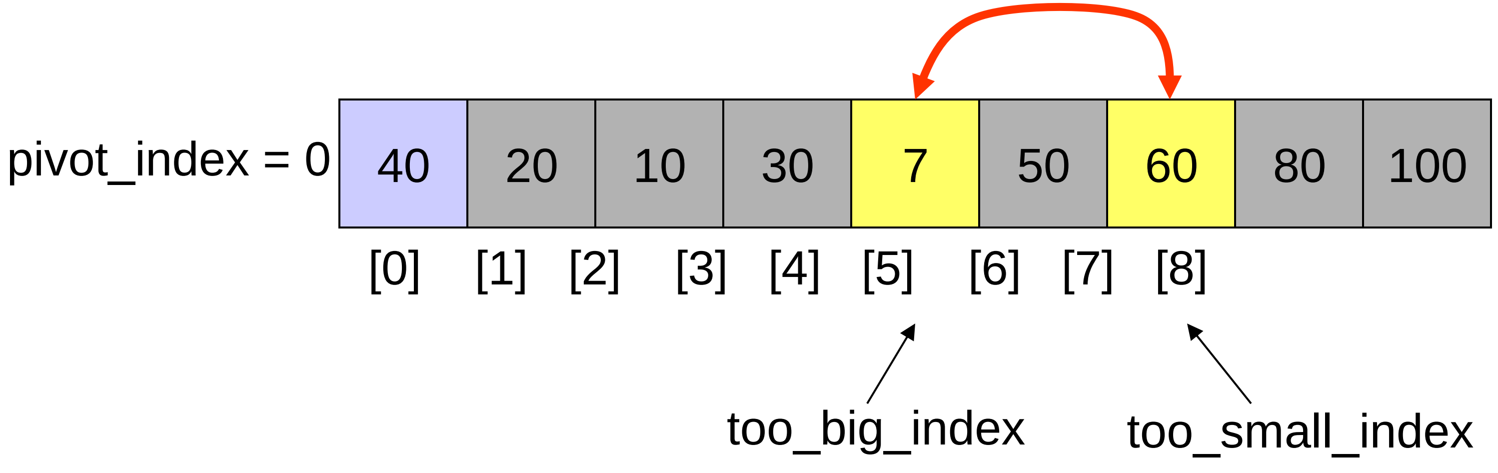
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



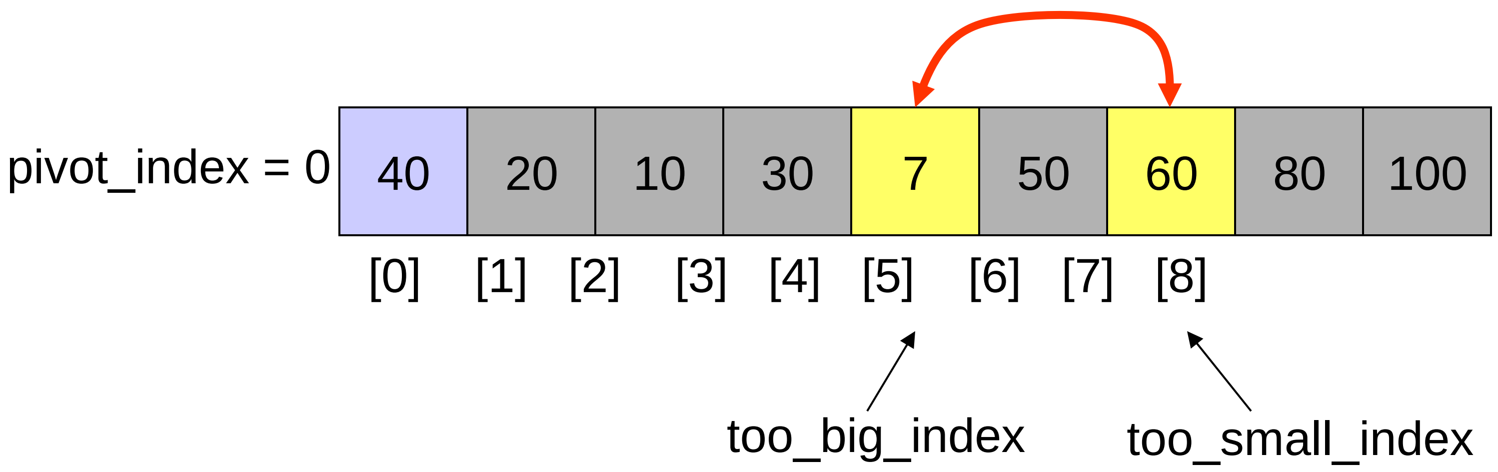
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



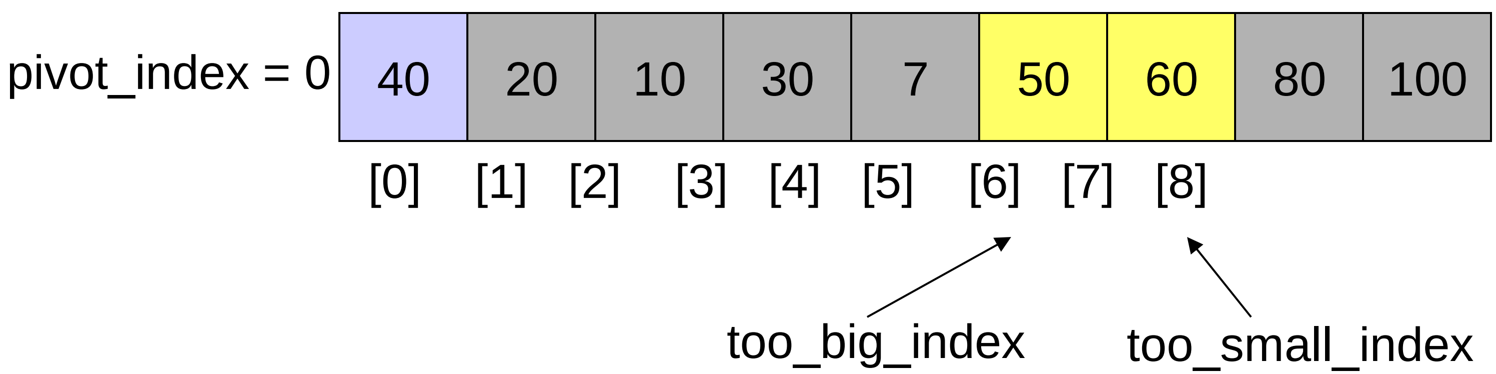
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



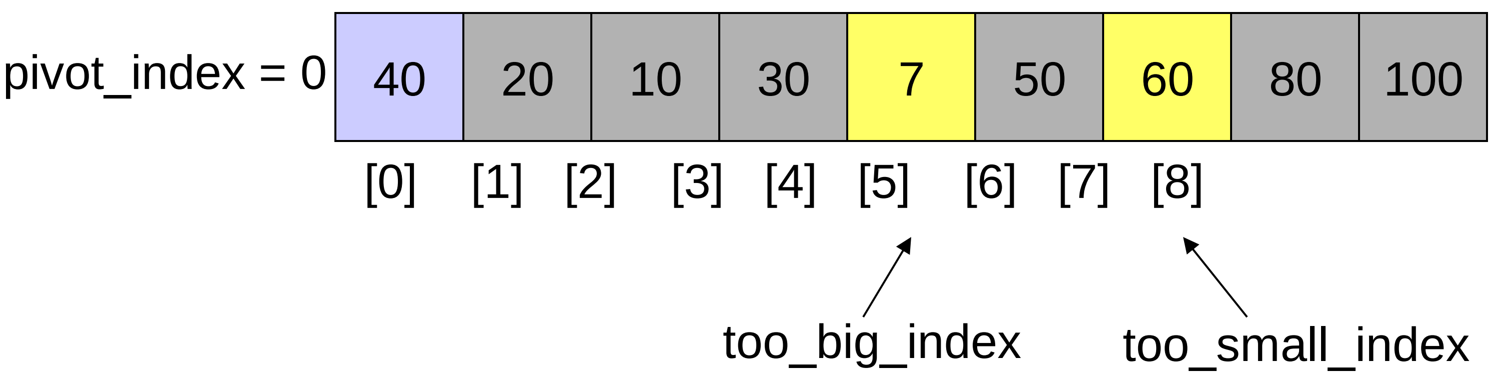
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



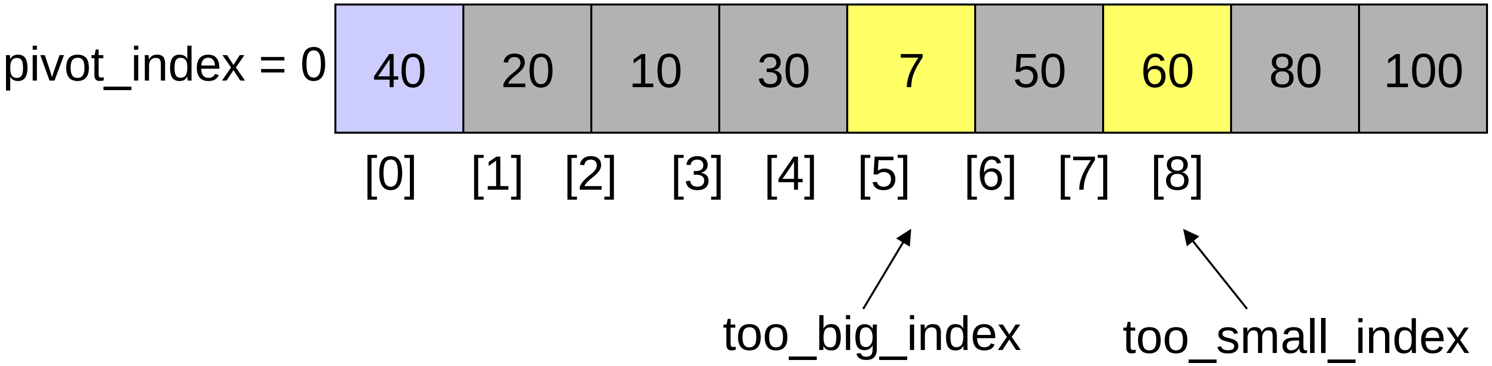
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



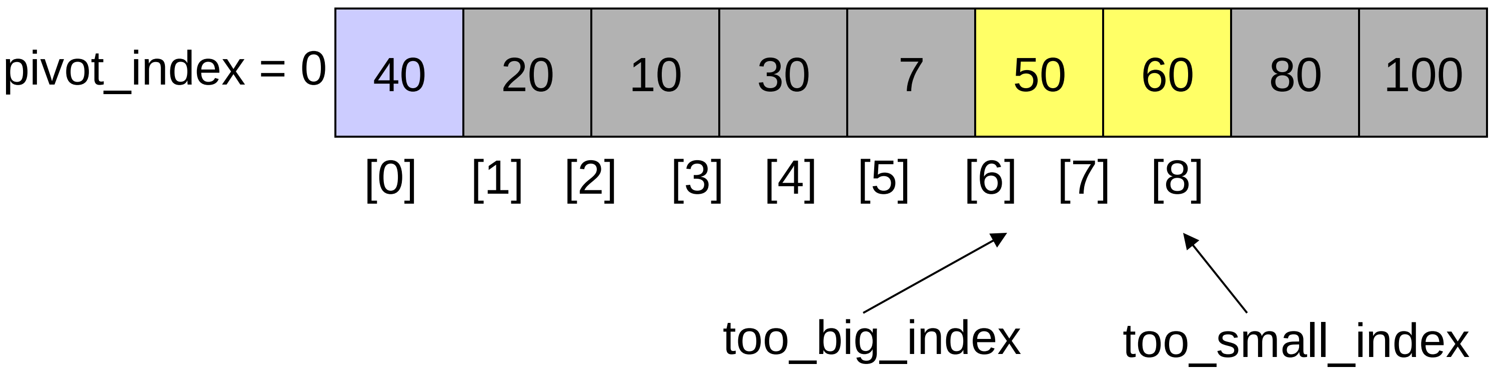
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



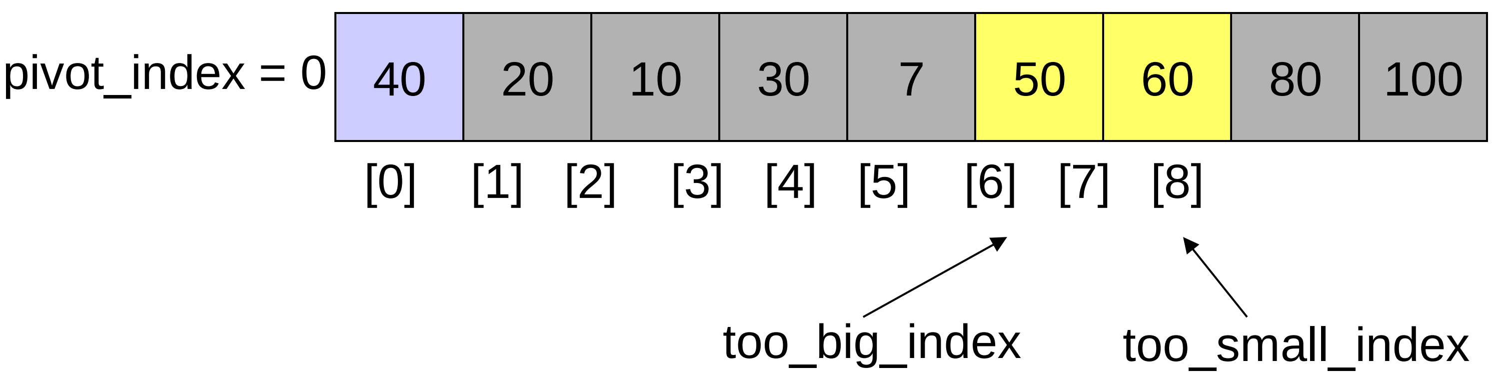
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



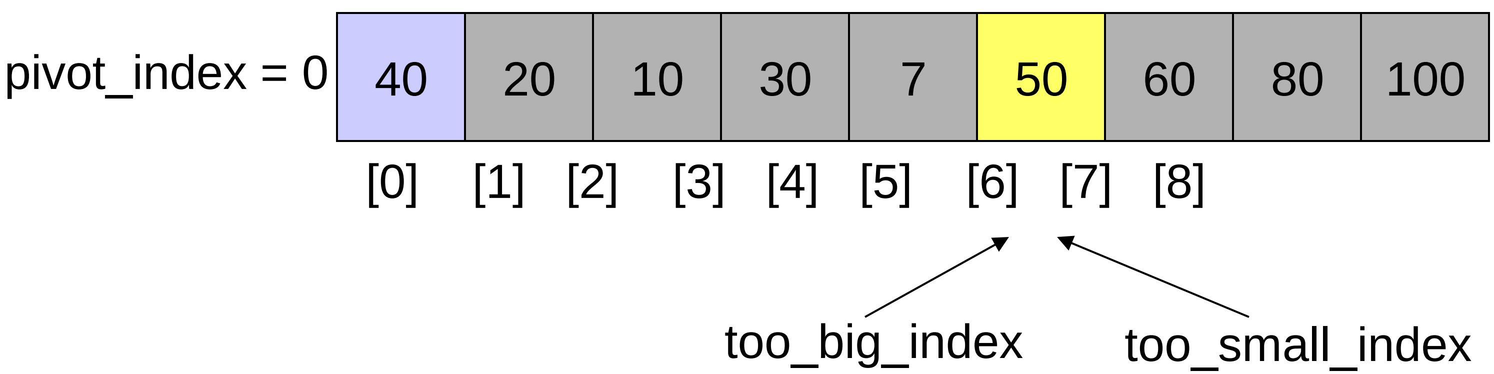
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



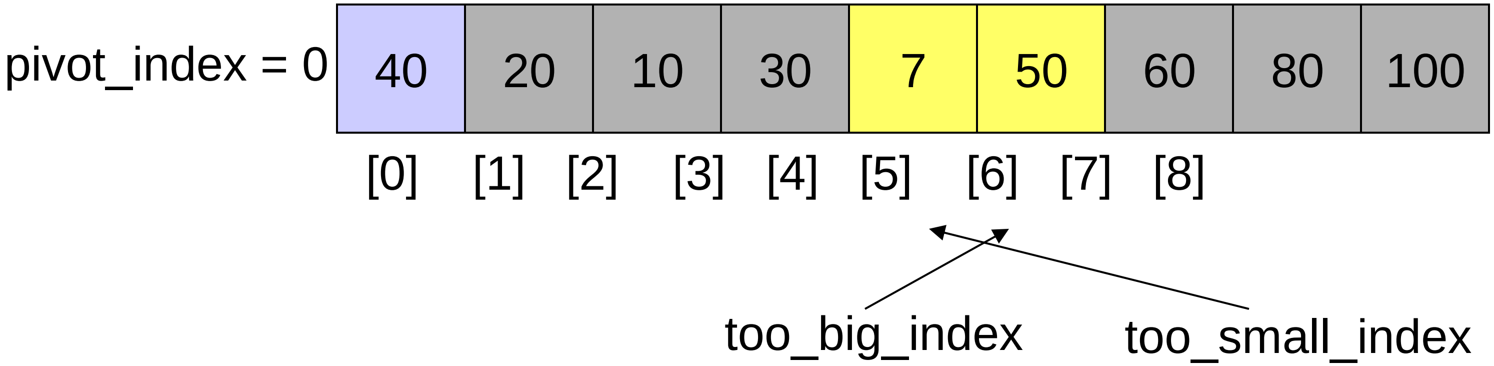
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



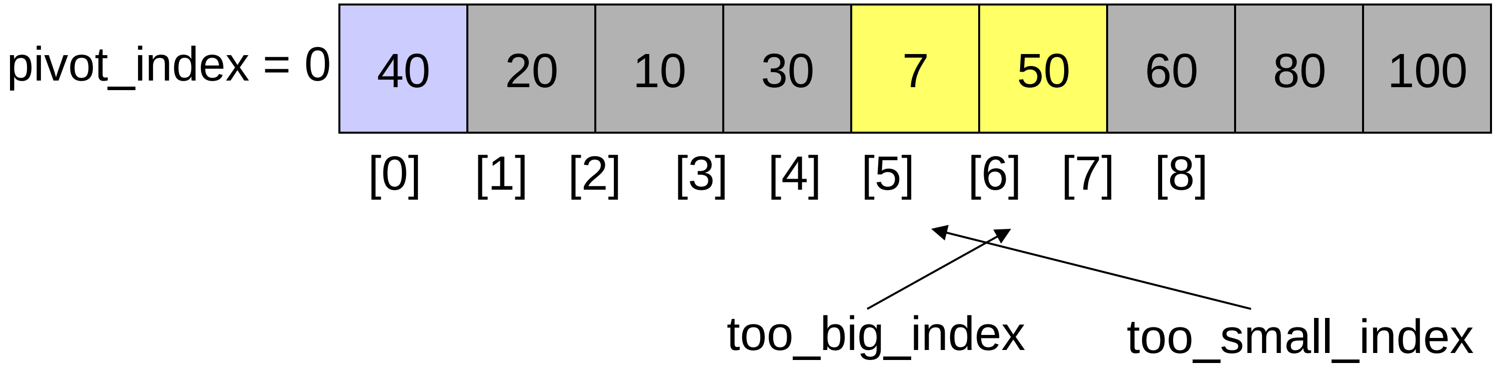
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



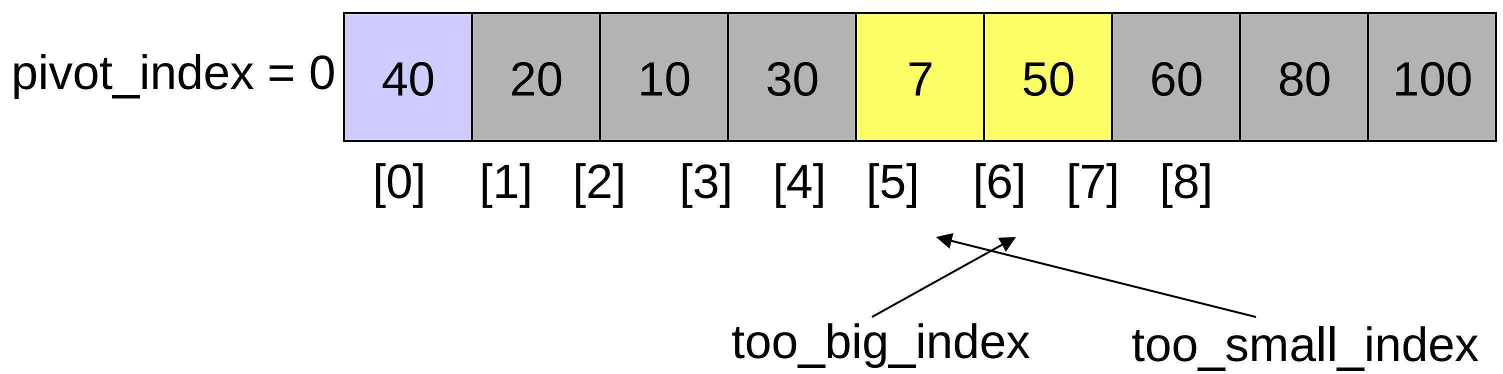
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



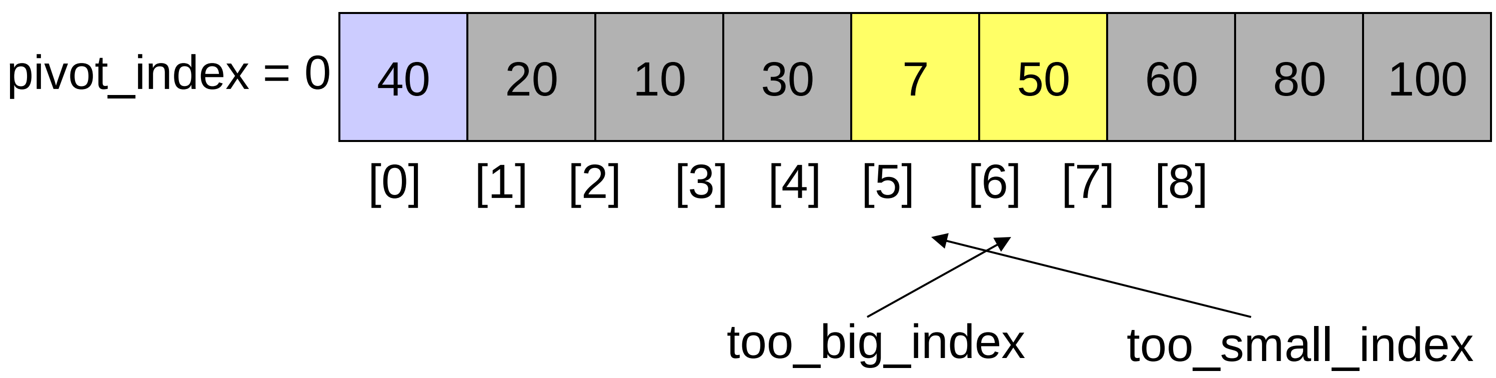
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



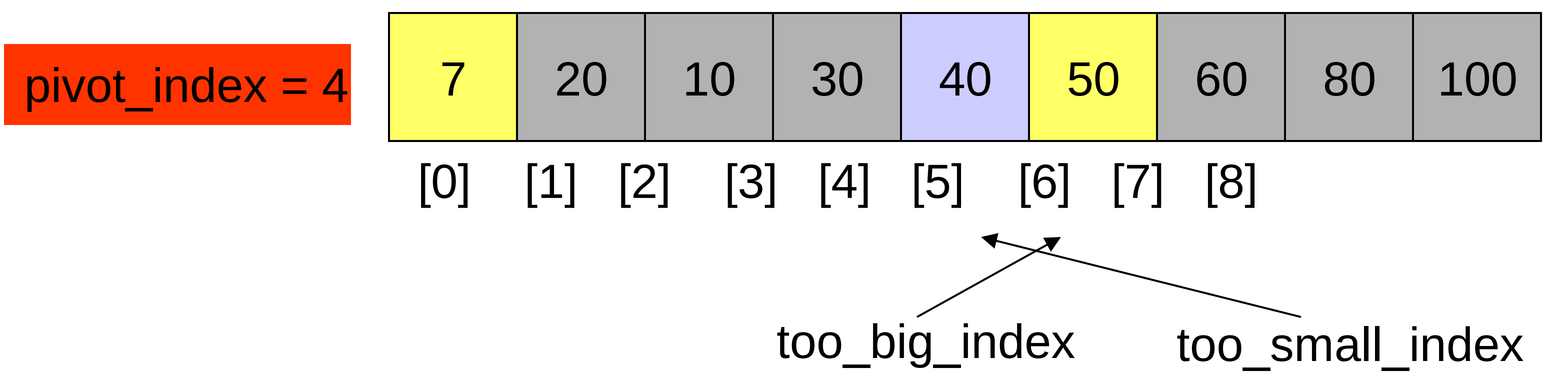
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



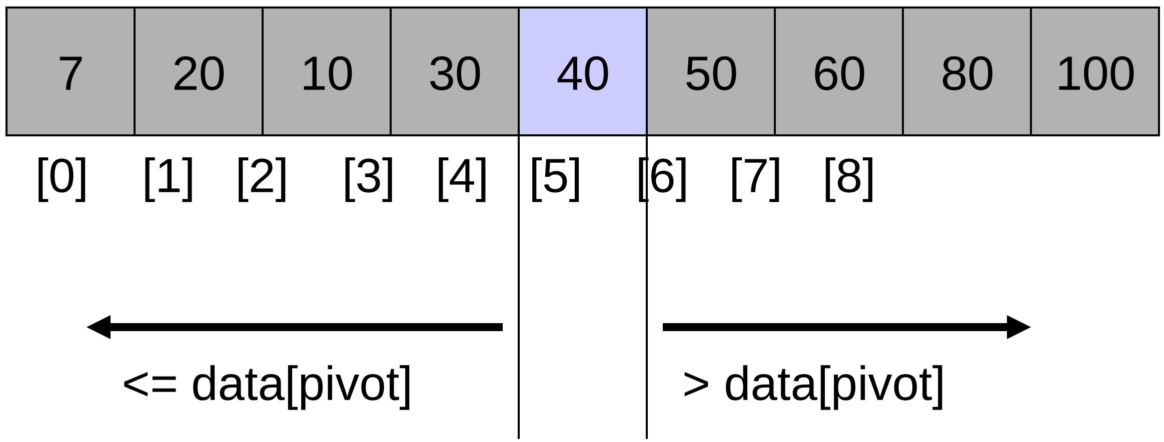
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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