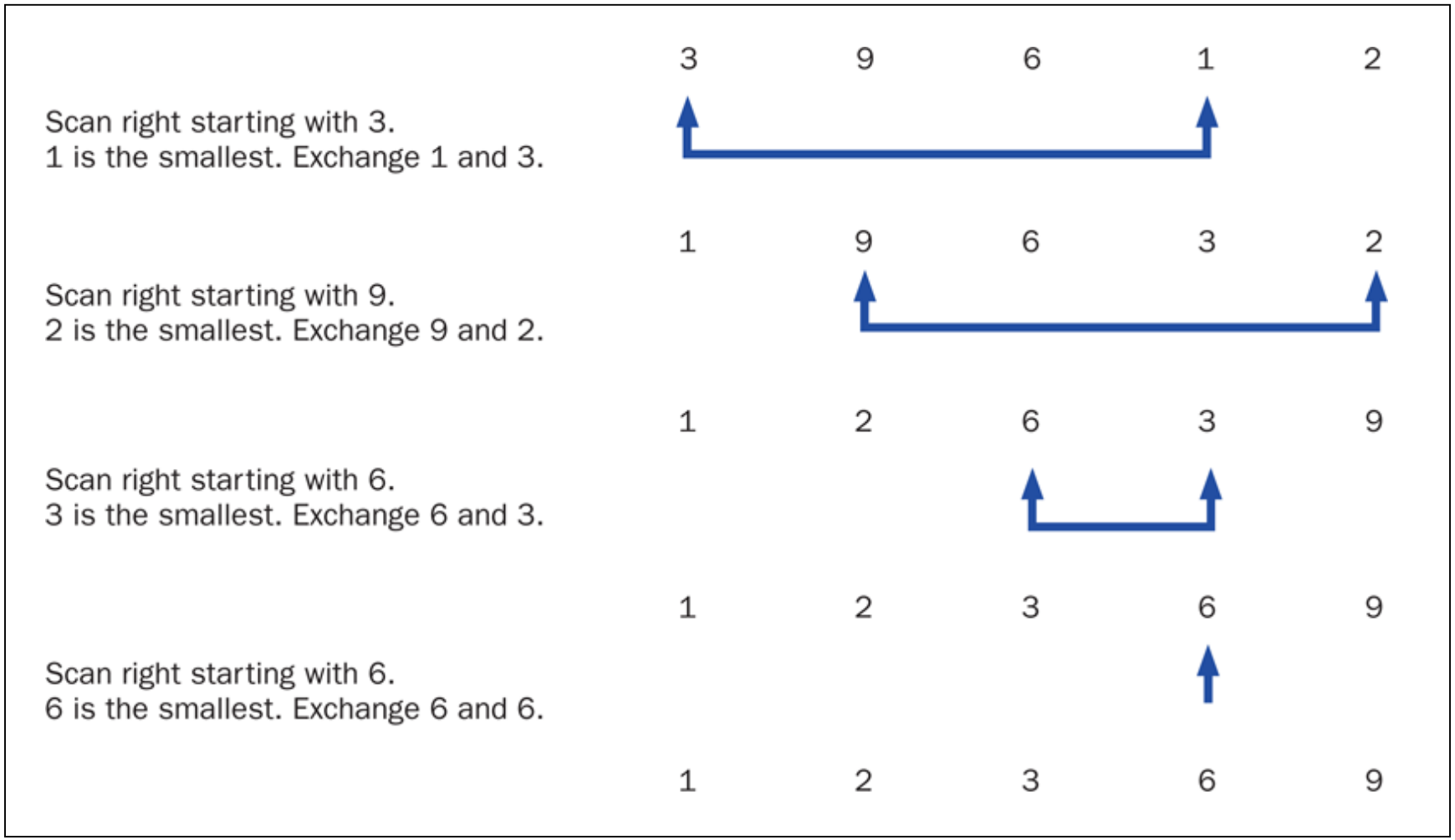
Sorting

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific value(s)
  - sorting a list of test scores in ascending numeric order
  - sorting a list of people alphabetically by last name
- There are many algorithms, which vary in efficiency, for sorting a list of items
- We will examine two specific algorithms:
  - **Selection Sort**
  - **Insertion Sort**

Selection Sort

- The approach of Selection Sort:
  - select a value and put it in its final place into the list
  - repeat for all other values
- In more detail:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Selection Sort



Swapping

- The processing of the selection sort algorithm includes the *swapping* of two values
- Swapping requires three assignment statements and a temporary storage location
- To swap the values of `first` and `second`:

```
temp = first;
first = second;
second = temp;
```

Selection Sort

- This technique allows each class to decide for itself what it means for one object to be less than another
- Let's look at an example that sorts an array of Contact objects
- The selectionSort method is a static method in the Sorting class
- See PhoneList.java
- See Sorting.java
- See Contact.java

```
//********************************************************************
//  PhoneList.java       Author: Lewis/Loftus
//
//  Driver for testing a sorting algorithm.
//********************************************************************

public class PhoneList
{
   //-----------------------------------------------------------------
   //  Creates an array of Contact objects, sorts them, then prints
   //  them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Contact[] friends = new Contact[8];

      friends[0] = new Contact ("John", "Smith", "610-555-7384");
      friends[1] = new Contact ("Sarah", "Barnes", "215-555-3827");
      friends[2] = new Contact ("Mark", "Riley", "733-555-2969");
      friends[3] = new Contact ("Laura", "Getz", "663-555-3984");
      friends[4] = new Contact ("Larry", "Smith", "464-555-3489");
      friends[5] = new Contact ("Frank", "Phelps", "322-555-2284");
      friends[6] = new Contact ("Mario", "Guzman", "804-555-9066");
      friends[7] = new Contact ("Marsha", "Grant", "243-555-2837");
```

continue

continue

```
        Sorting.selectionSort(friends);

        for (Contact friend : friends)
           System.out.println (friend);
    }
}
```

continue

```
        Sorting.select
        for (Contact f
            System.out.
    }
}
```

Output

```
Barnes, Sarah     215-555-3827
Getz, Laura       663-555-3984
Grant, Marsha     243-555-2837
Guzman, Mario     804-555-9066
Phelps, Frank     322-555-2284
Riley, Mark       733-555-2969
Smith, John       610-555-7384
Smith, Larry      464-555-3489
```

```
//-----------------------------------------------------------------
//  Sorts the specified array of objects using the selection
//  sort algorithm.
//-----------------------------------------------------------------
public static void selectionSort (Comparable[] list)
{
   int min;
   Comparable temp;

   for (int index = 0; index < list.length-1; index++)
   {
      min = index;
      for (int scan = index+1; scan < list.length; scan++)
         if (list[scan].compareTo(list[min]) < 0)
            min = scan;

      // Swap the values
      temp = list[min];
      list[min] = list[index];
      list[index] = temp;
   }
}
```

```
//********************************************************************
//  Contact.java       Author: Lewis/Loftus
//
//  Represents a phone contact.
//********************************************************************

public class Contact implements Comparable
{
   private String firstName, lastName, phone;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this contact with the specified data.
   //-----------------------------------------------------------------
   public Contact (String first, String last, String telephone)
   {
      firstName = first;
      lastName = last;
      phone = telephone;
   }
```

continue

continue

```
//-----------------------------------------------------------------
//  Returns a description of this contact as a string.
//-----------------------------------------------------------------
public String toString ()
{
   return lastName + ", " + firstName + "\t" + phone;
}

//-----------------------------------------------------------------
//  Returns a description of this contact as a string.
//-----------------------------------------------------------------
public boolean equals (Object other)
{
   return (lastName.equals(((Contact)other).getLastName()) &&
           firstName.equals(((Contact)other).getFirstName()));
}
```

continue

continue

```
//-----------------------------------------------------------------
//  Uses both last and first names to determine ordering.
//-----------------------------------------------------------------
public int compareTo (Object other)
{
   int result;

   String otherFirst = ((Contact)other).getFirstName();
   String otherLast = ((Contact)other).getLastName();

   if (lastName.equals(otherLast))
      result = firstName.compareTo(otherFirst);
   else
      result = lastName.compareTo(otherLast);

   return result;
}
```

continue

continue

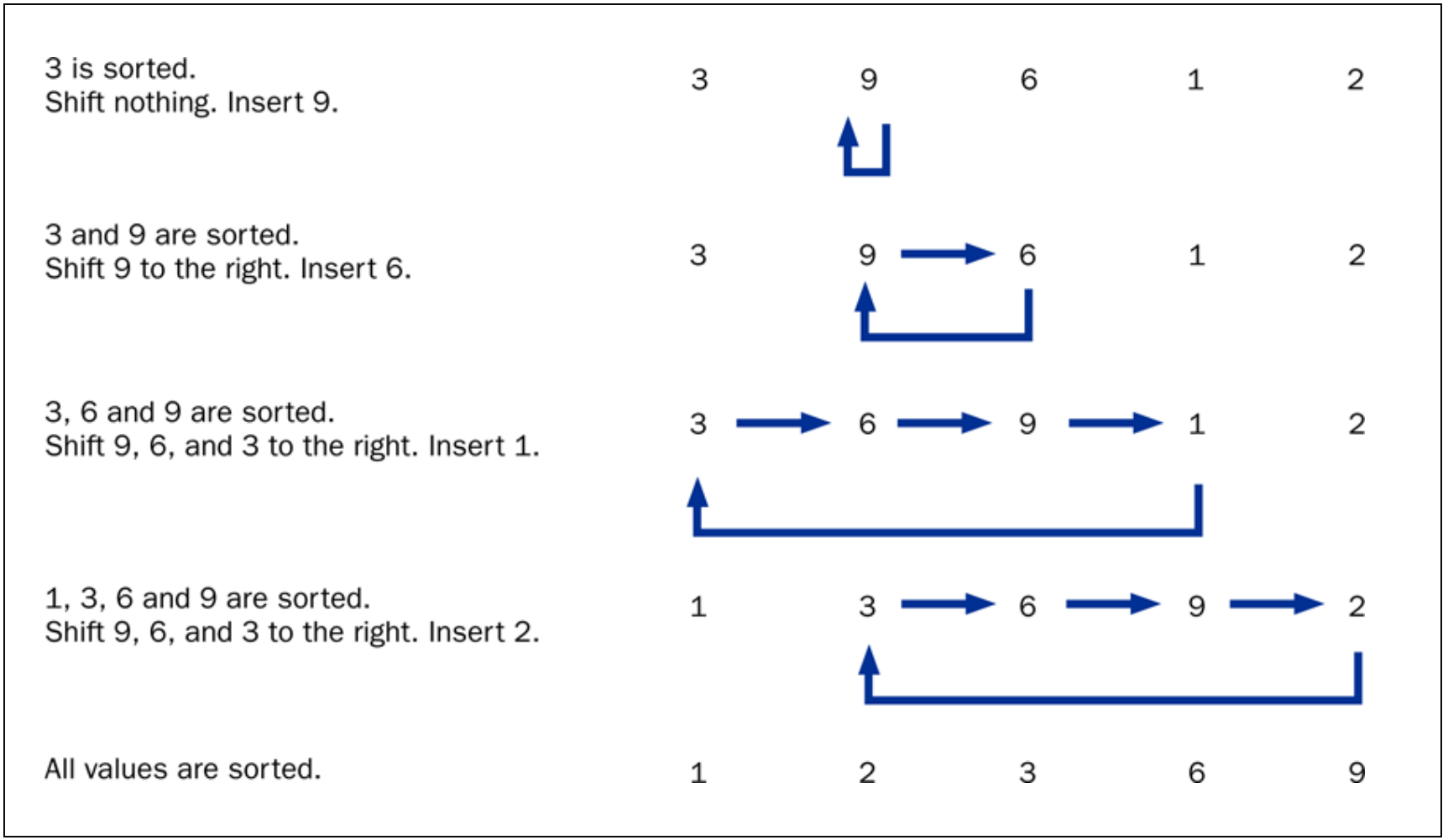
```
    //-------------------------------------------------------------------
    //  First name accessor.
    //-------------------------------------------------------------------
    public String getFirstName ()
    {
       return firstName;
    }

    //-------------------------------------------------------------------
    //  Last name accessor.
    //-------------------------------------------------------------------
    public String getLastName ()
    {
       return lastName;
    }
}
```

Insertion Sort

- The approach of Insertion Sort:
  - pick any item and insert it into its proper place in a sorted sublist
  - repeat until all items have been inserted
- In more detail:
  - consider the first item to be a sorted sublist (of one item)
  - insert the second item into the sorted sublist, shifting the first item as needed to make room to insert the new addition
  - insert the third item into the sorted sublist (of two items), shifting items as necessary
  - repeat until all values are inserted into their proper positions

Insertion Sort



```
//-----------------------------------------------------------------
//  Sorts the specified array of objects using the insertion
//  sort algorithm.
//-----------------------------------------------------------------
public static void insertionSort (Comparable[] list)
{
   for (int index = 1; index < list.length; index++)
   {
      Comparable key = list[index];
      int position = index;

      //  Shift larger values to the right
      while (position > 0 && key.compareTo(list[position-1]) < 0)
      {
         list[position] = list[position-1];
         position--;
      }

      list[position] = key;
   }
}
```

Comparing Sorts

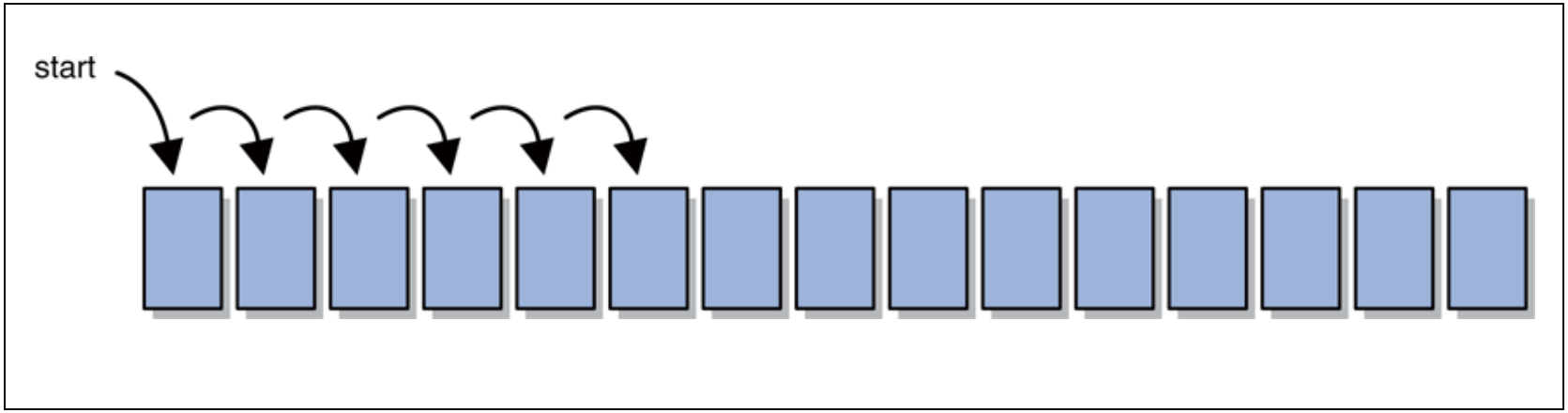
- The Selection and Insertion sort algorithms are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts are of order n^2
- Other sorts are more efficient: order nlogn

Searching

- Searching is the process of finding a target element within a group of items called the search pool
- The target may or may not be in the search pool
- We want to perform the search efficiently, minimizing the number of comparisons
- Let's look at two classic searching approaches: linear search and binary search
- As we did with sorting, we'll implement the searches with polymorphic Comparable parameters

Linear Search

- A linear search begins at one end of a list and examines each element in turn
- Eventually, either the item is found or the end of the list is encountered

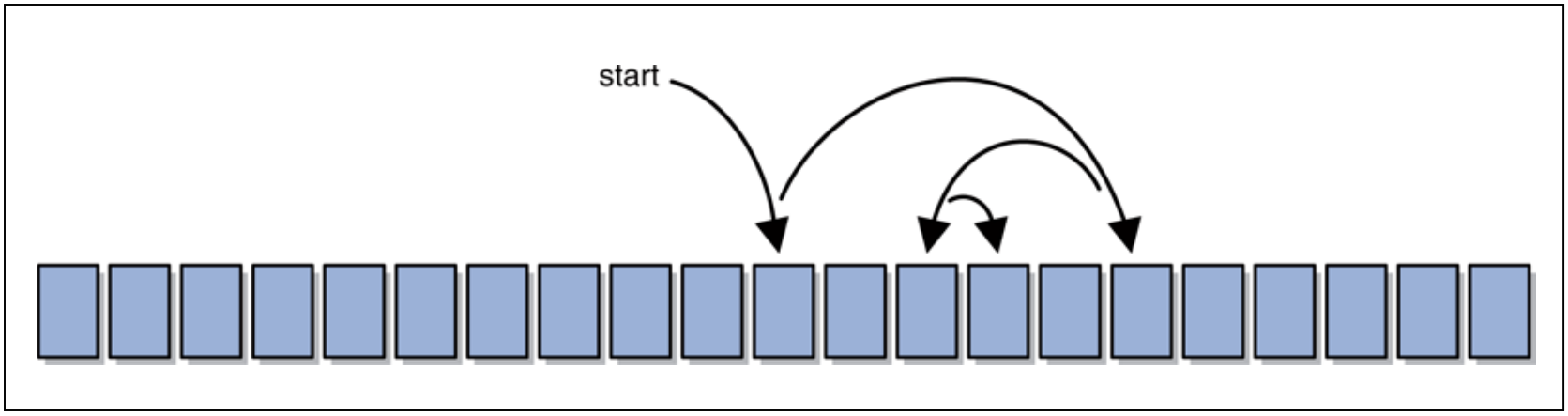


Binary Search

- **A binary search assumes the list of items in the search pool is sorted**
- It eliminates a large part of the search pool with a single comparison
- A binary search first examines the middle element of the list -- if it matches the target, the search is over
- If it doesn't, only one half of the remaining elements need be searched
- Since they are sorted, the target can only be in one half of the other

Binary Search

- The process continues by comparing the middle element of the remaining viable candidates
- Each comparison eliminates approximately half of the remaining data
- Eventually, the target is found or the data is exhausted



Linear Search

- A linear search begins at one end of a list and examines each element in turn
- Eventually, either the item is found or the end of the list is encountered
- See PhoneList2.java
- See Searching.java, specifically the linearSearch method

```
//********************************************************************
//  PhoneList2.java       Author: Lewis/Loftus
//
//  Driver for testing searching algorithms.
//********************************************************************

public class PhoneList2
{
   //-----------------------------------------------------------------
   //  Creates an array of Contact objects, sorts them, then prints
   //  them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Contact test, found;
      Contact[] friends = new Contact[8];

      friends[0] = new Contact ("John", "Smith", "610-555-7384");
      friends[1] = new Contact ("Sarah", "Barnes", "215-555-3827");
      friends[2] = new Contact ("Mark", "Riley", "733-555-2969");
      friends[3] = new Contact ("Laura", "Getz", "663-555-3984");
      friends[4] = new Contact ("Larry", "Smith", "464-555-3489");
      friends[5] = new Contact ("Frank", "Phelps", "322-555-2284");
      friends[6] = new Contact ("Mario", "Guzman", "804-555-9066");
      friends[7] = new Contact ("Marsha", "Grant", "243-555-2837");
```

continue

continue

```
        test = new Contact ("Frank", "Phelps", "");
        found = (Contact) Searching.linearSearch(friends, test);
        if (found != null)
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
        System.out.println ();

        Sorting.selectionSort(friends);

        test = new Contact ("Mario", "Guzman", "");
        found = (Contact) Searching.binarySearch(friends, test);
        if (found != null)
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
    }
}
```

continue

```
        test = n
        found =                                          test);
        if (foun
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
        System.out.println ();

        Sorting.selectionSort(friends);

        test = new Contact ("Mario", "Guzman", "");
        found = (Contact) Searching.binarySearch(friends, test);
        if (found != null)
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
    }
}
```

Output

```
Found: Phelps, Frank      322-555-2284

Found: Guzman, Mario      804-555-9066
```

```
//-----------------------------------------------------------------
//  Searches the specified array of objects for the target using
//  a linear search. Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)
{
   int index = 0;
   boolean found = false;

   while (!found && index < list.length)
   {
      if (list[index].equals(target))
         found = true;
      else
         index++;
   }

   if (found)
      return list[index];
   else
      return null;
}
```

```
//-----------------------------------------------------------------
//  Searches the specified array of objects for the target using
//  a binary search. Assumes the array is already sorted in
//  ascending order when it is passed in. Returns a reference to
//  the target object from the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable binarySearch (Comparable[] list,
                                       Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   while (!found && min <= max)
   {
      mid = (min+max) / 2;
      if (list[mid].equals(target))
         found = true;
      else
         if (target.compareTo(list[mid]) < 0)
            max = mid-1;
         else
            min = mid+1;
   }
```

continue

continue

```
        if (found)
            return list[mid];
        else
            return null;
    }
```

Interview question on binary search

- Two similar eggs
- n Stairs
- Breaks at and after a certain stair k
- Find k with min number of trials

More Efficient Sorting Algorithms

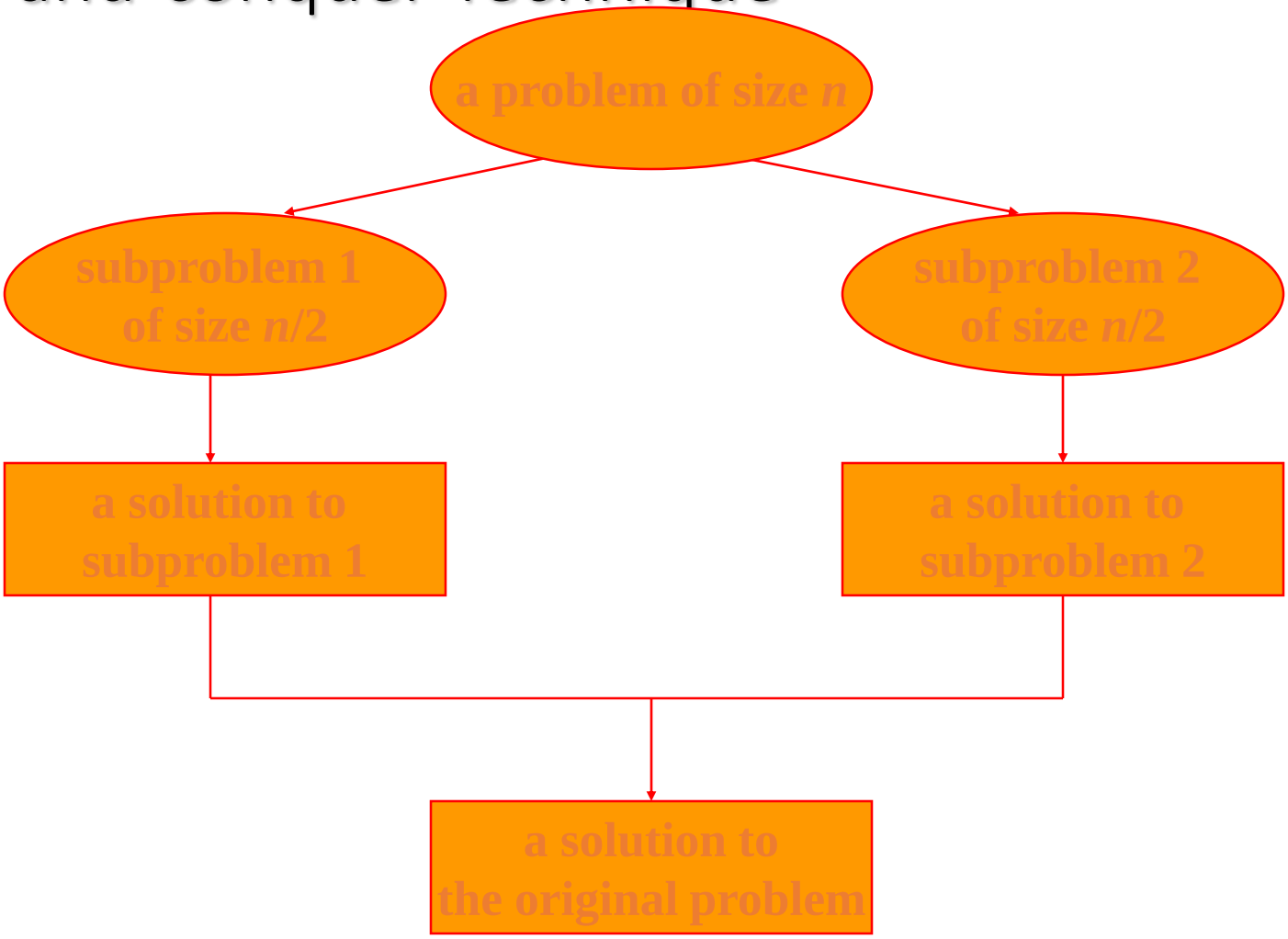
- http://math.hws.edu/TMCM/java/xSortLab/

Divide and Conquer

The most well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-conquer Technique



Divide and Conquer Examples

- Sorting: mergesort and quicksort
- Tree traversals
- Binary search

Mergesort

Algorithm:

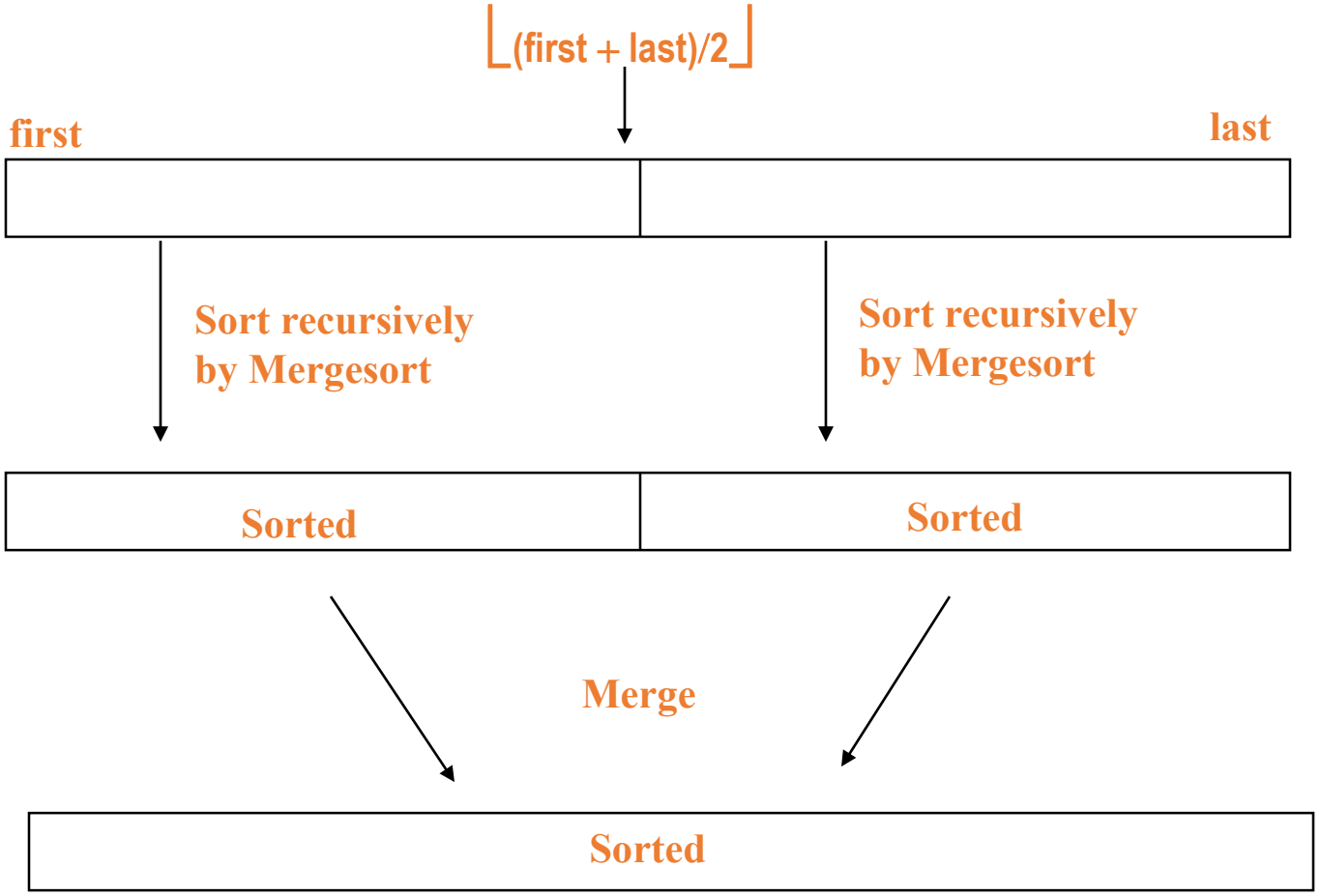
- Split array A[1..*n*] in two
- Make copies of each half
  - In arrays B[1.. *n*/2] and C[1.. *n*/2]
- Sort arrays B and C

Mergesort

- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Using Divide and Conquer: Mergesort

- Mergesort Strategy



Algorithm: Mergesort

Input: Array E and indices first and last, such that the elements E[i] are defined for first <= i <= last.

Output: E[first], …, E[last] is a sorted rearrangement of the same elements

```
void mergeSort(Element[] E, int first, int last)
    if (first < last)
        int mid = (first+last)/2;
        mergeSort(E, first, mid);
        mergeSort(E, mid+1, last);
        merge(E, first, mid, last);
    return;
```

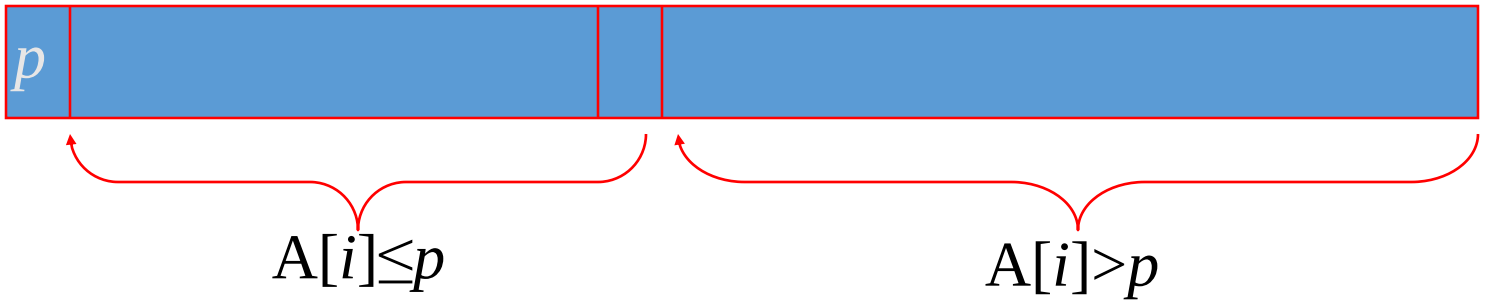
Animation

Mergesort complexity

- Mergesort always partitions the array equally.
- Thus, the recursive depth is always $O(\lg n)$
- The amount of work done at each level is $O(n)$
- Intuitively, the complexity is $O(n \lg n)$
- The amount of extra memory used is $O(n)$

Quicksort by Hoare (1962)

- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot
- Exchange the pivot with the last element in the first (i.e., ≤) sublist – the pivot is now in its final position
- Sort the two sublists recursively



Quicksort

http://math.hws.edu/TMCM/java/xSortLab/

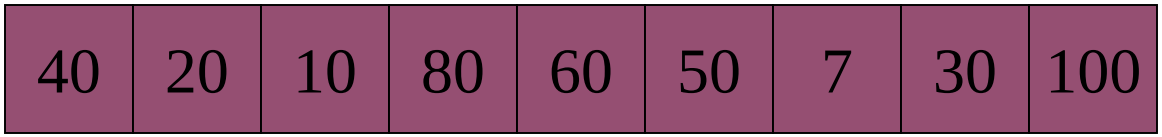
Quicksort Algorithm

- Input: Array E and indices first, and last, s.t. elements E[i] are defined for first $\leq$ i $\leq$ last
- Ouput: E[first], …, E[last] is a sorted rearrangement of the array
- Void quickSort(Element[] E, int first, int last)

```
if (first < last)
    Element pivotElement = E[first];
    int splitPoint = partition(E, pivotElement, first, last);
    quickSort (E, first, splitPoint –1 );
    quickSort (E, splitPoint +1, last );
return;
```

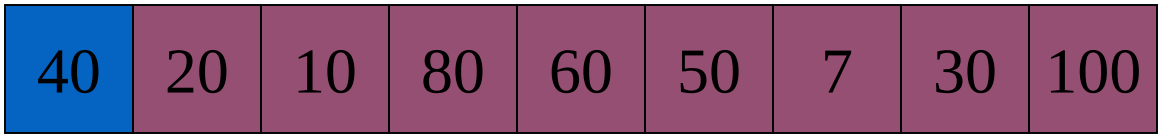
Example

We are given array of n integers to sort:



Pick Pivot Element

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



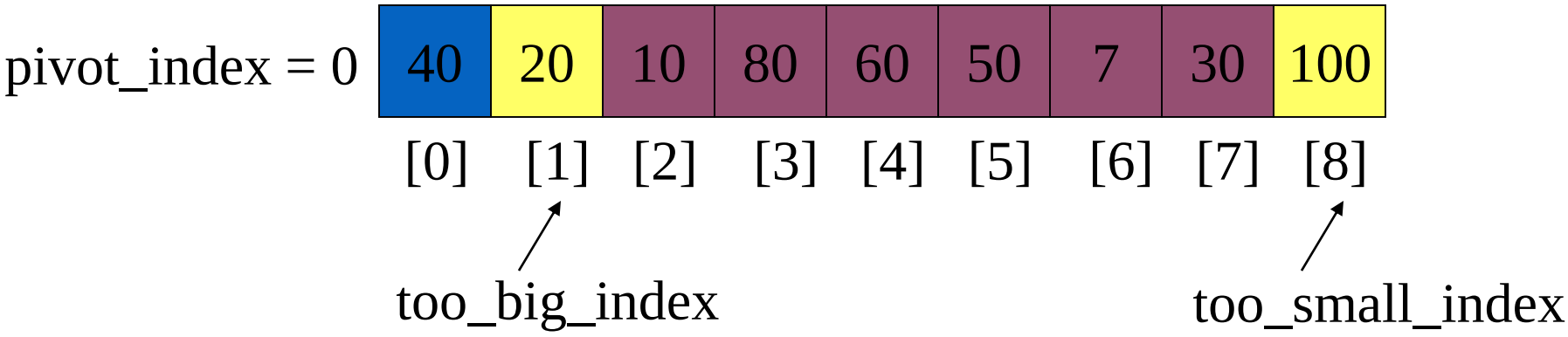
Partitioning Array

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

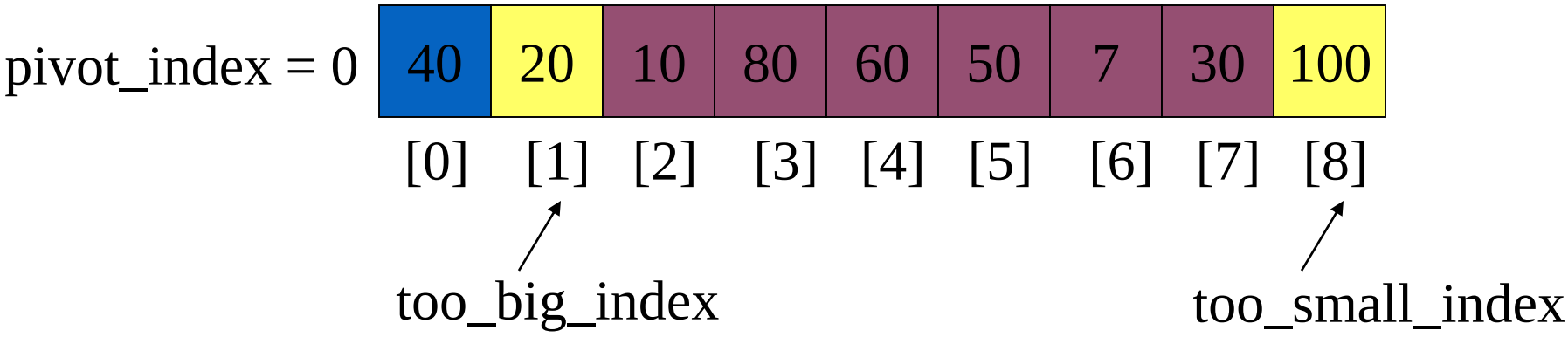
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

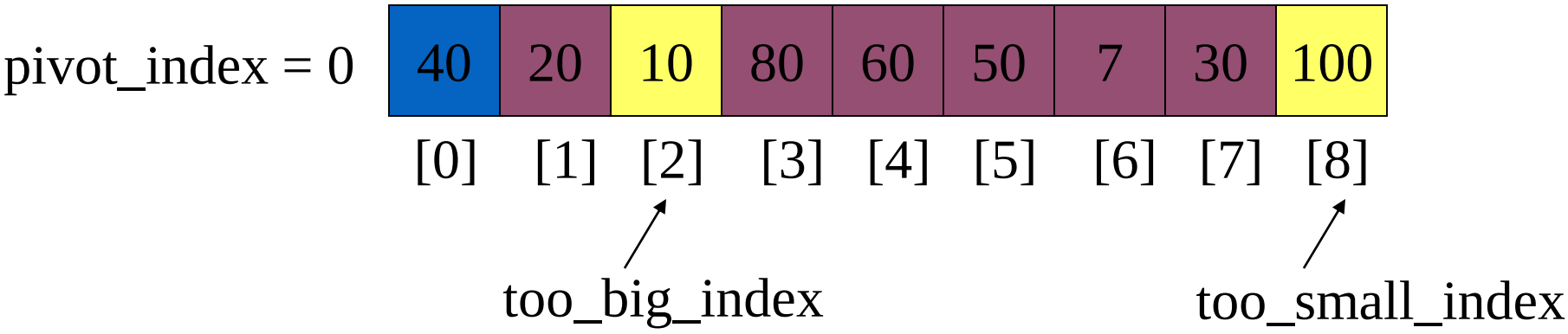
Partitioning loops through, swapping elements below/above pivot.


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

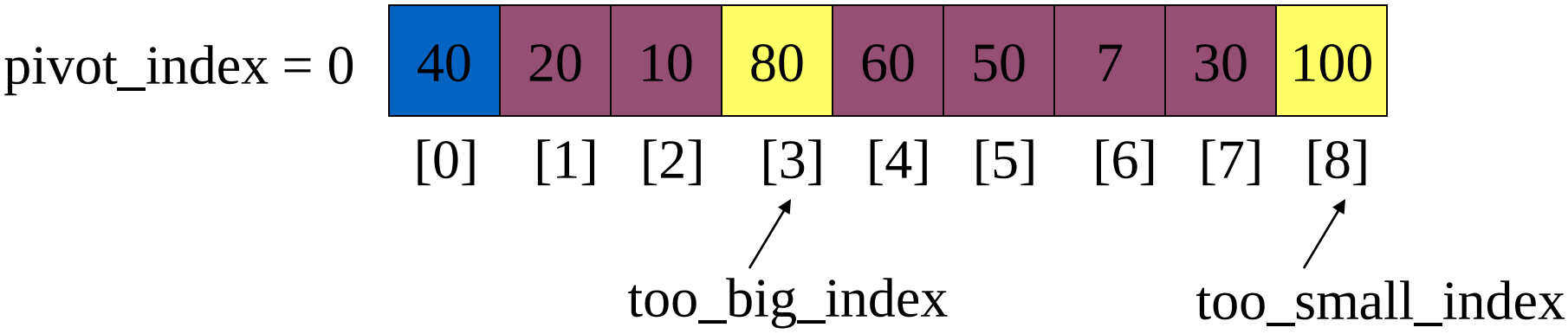
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



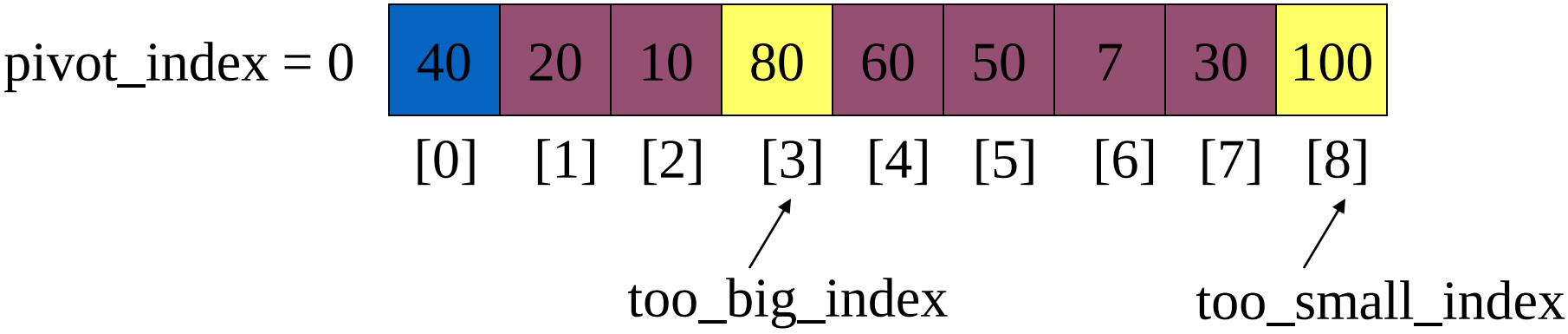
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



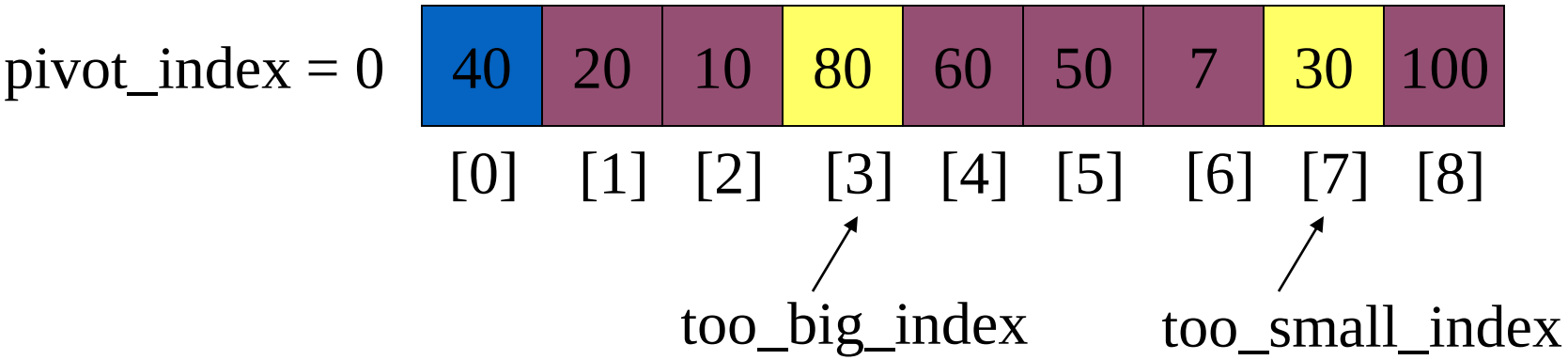
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



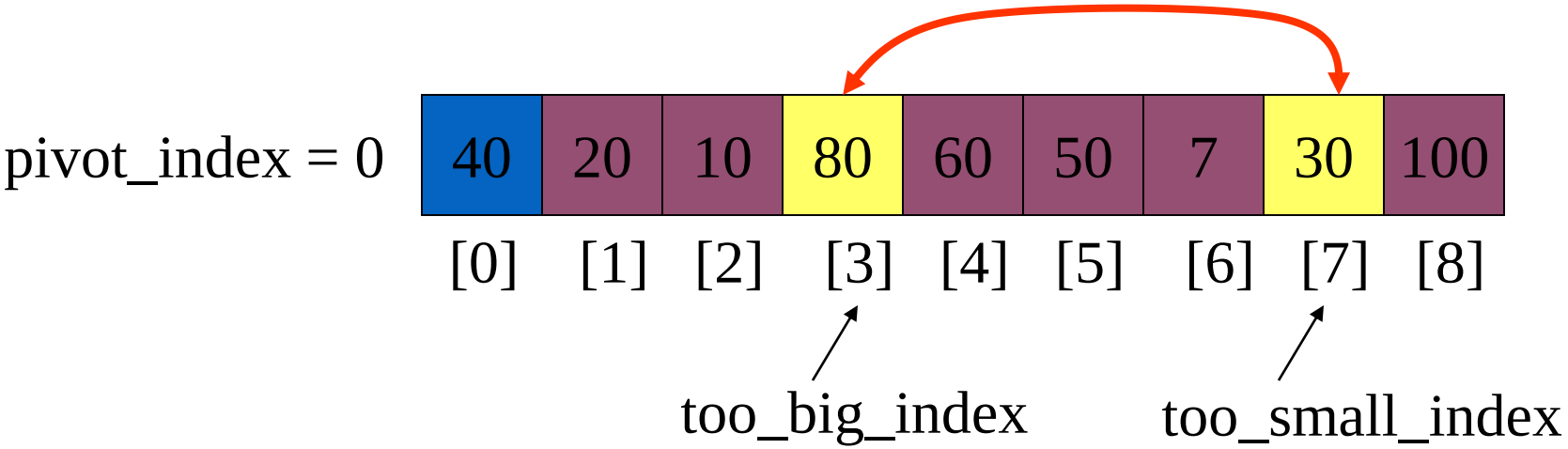
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



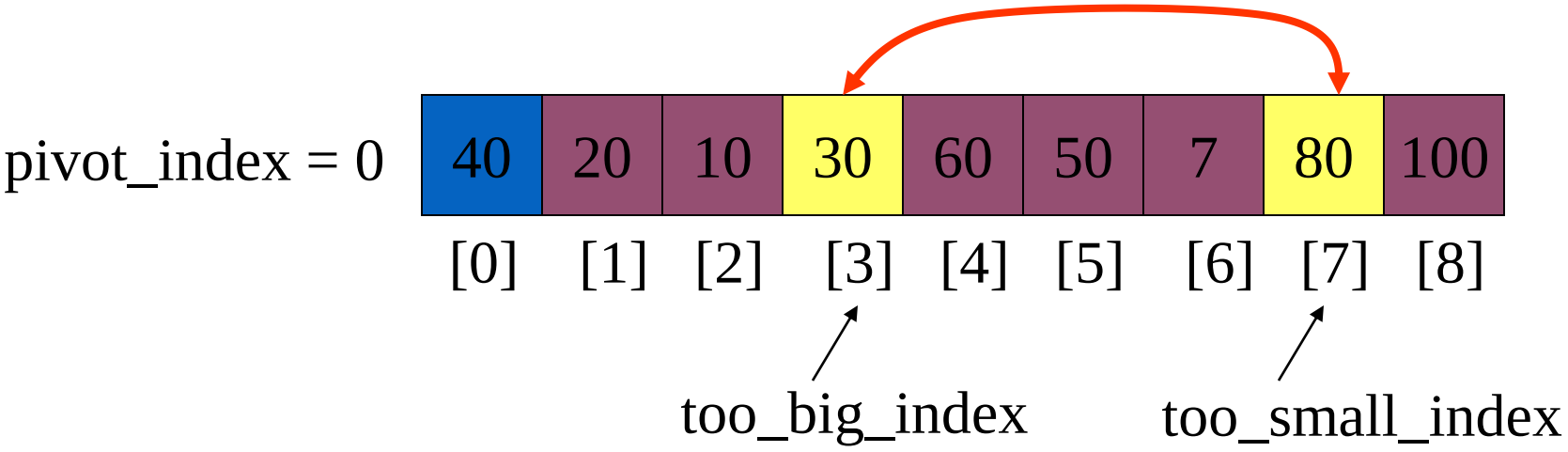
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



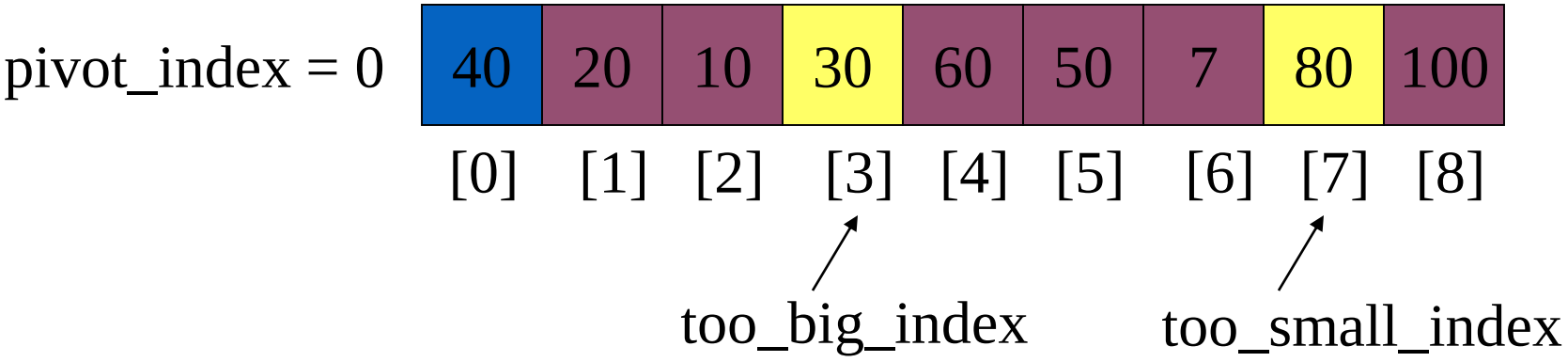
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



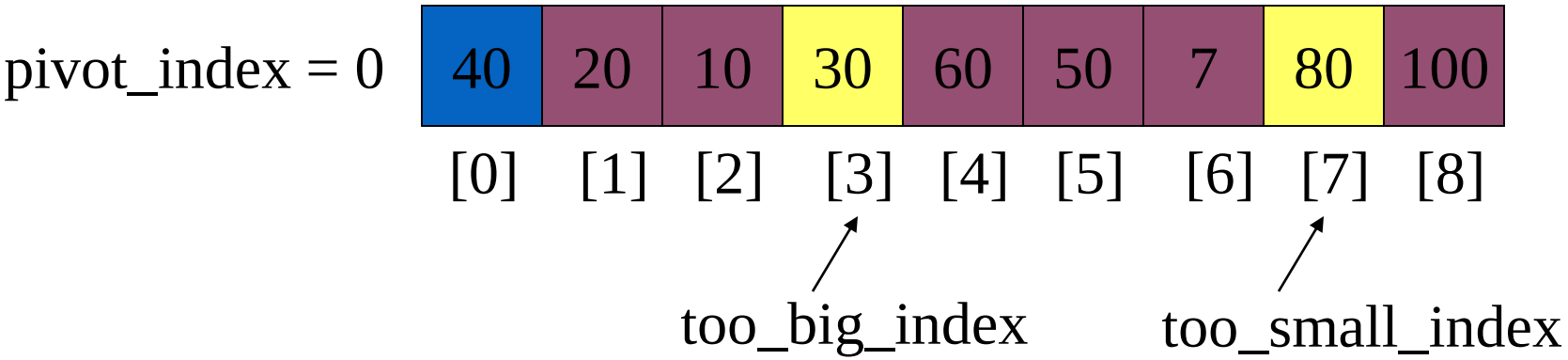
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



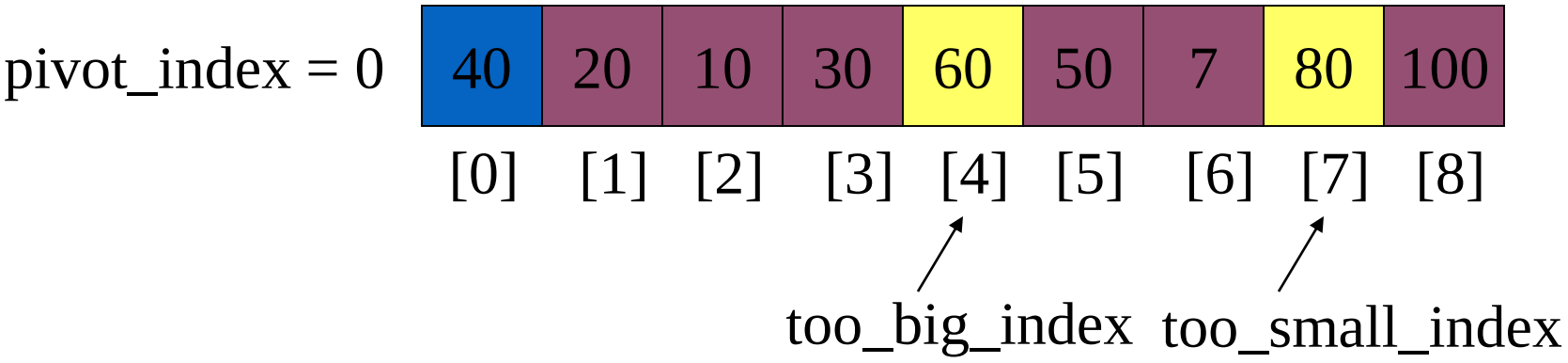
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



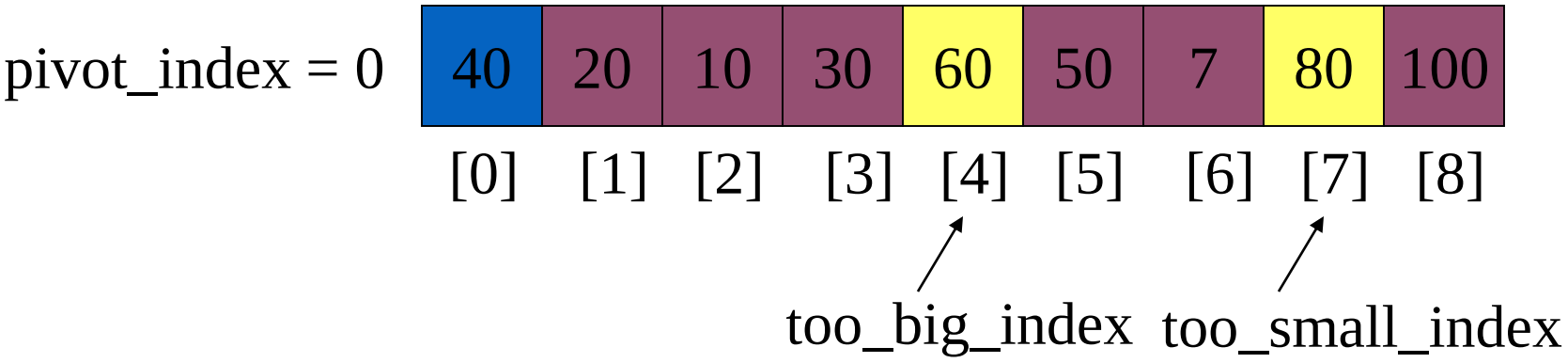
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



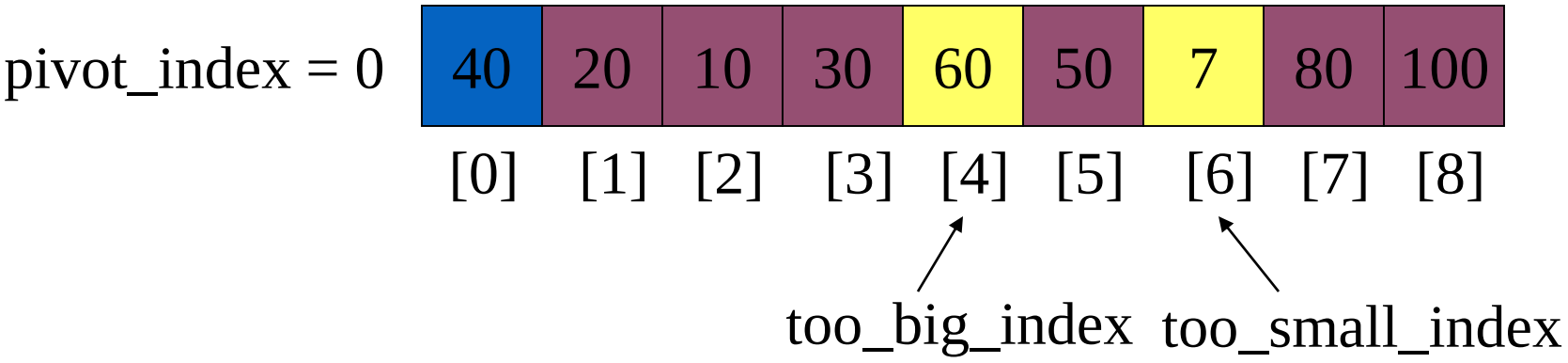
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



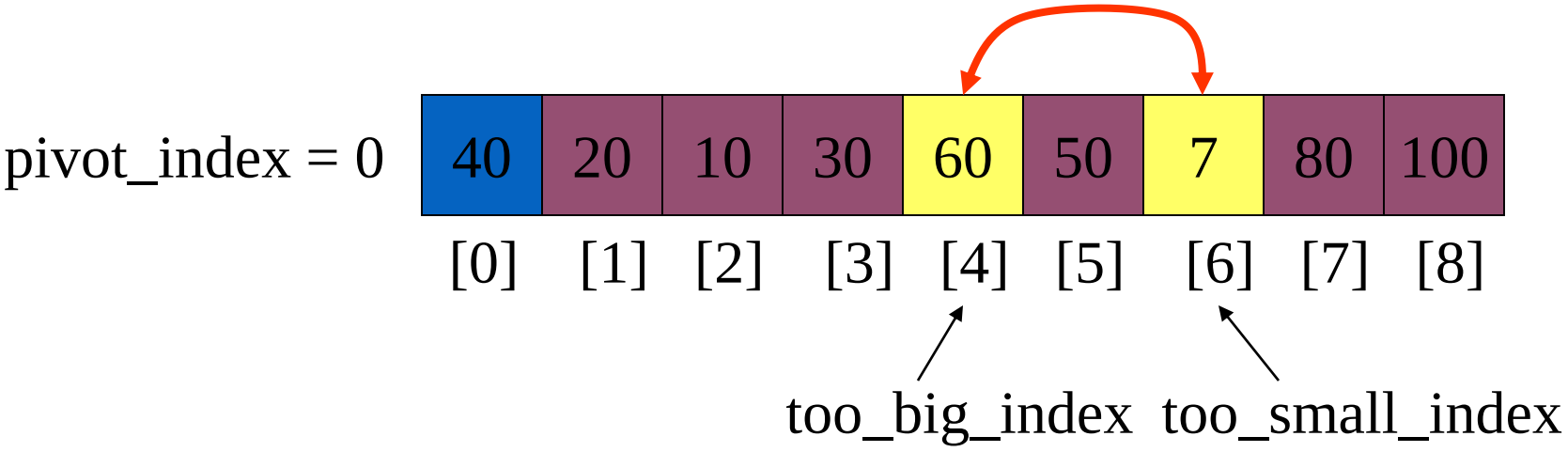
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



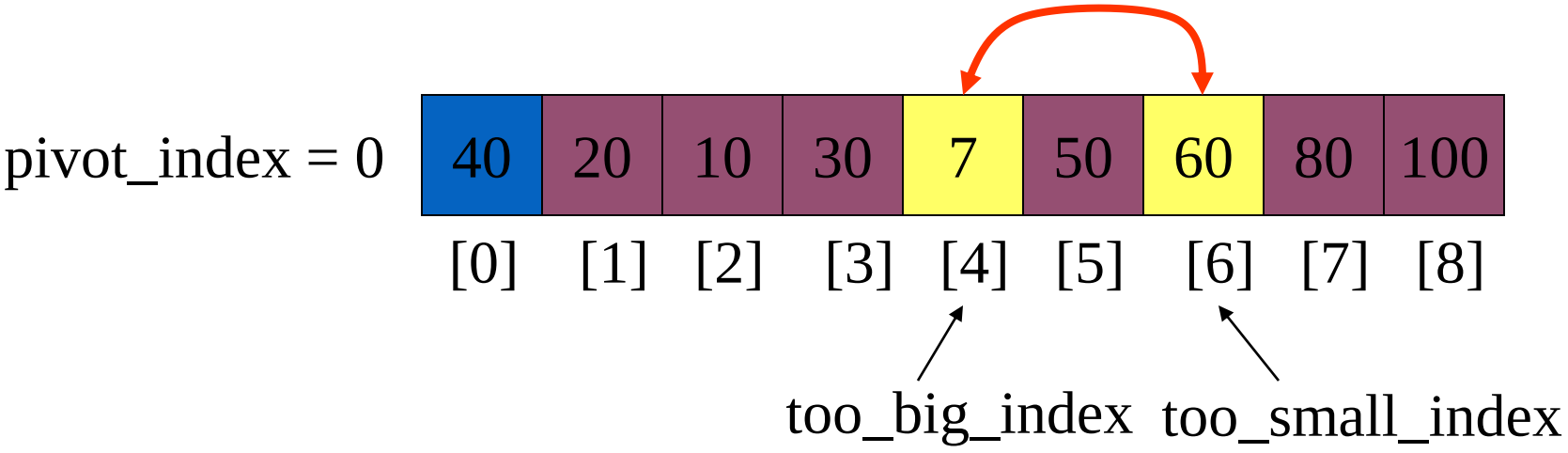
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



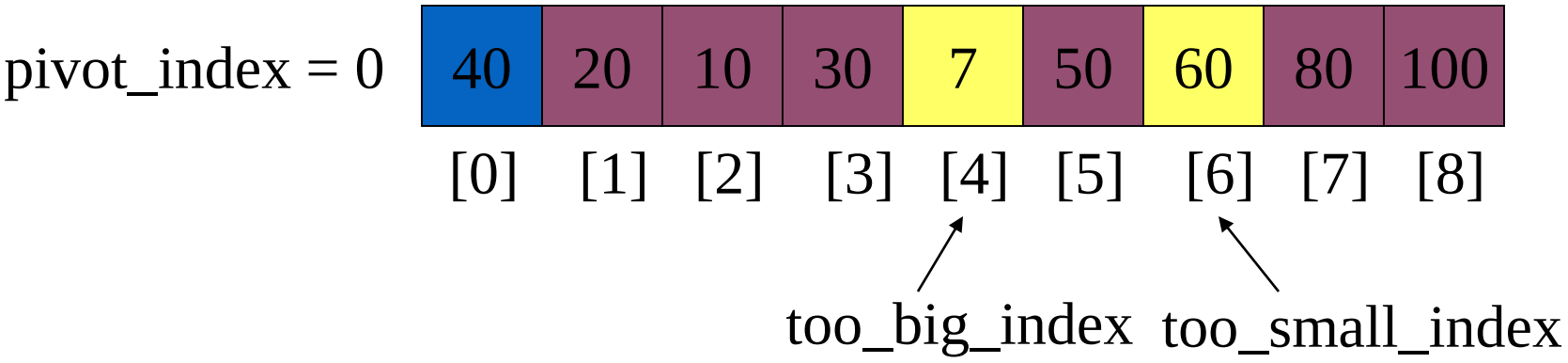
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



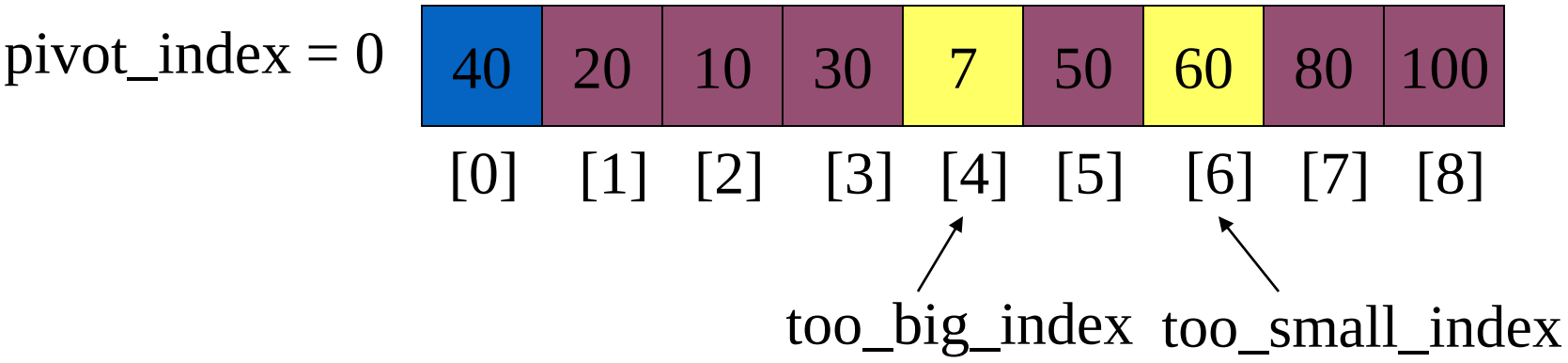
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



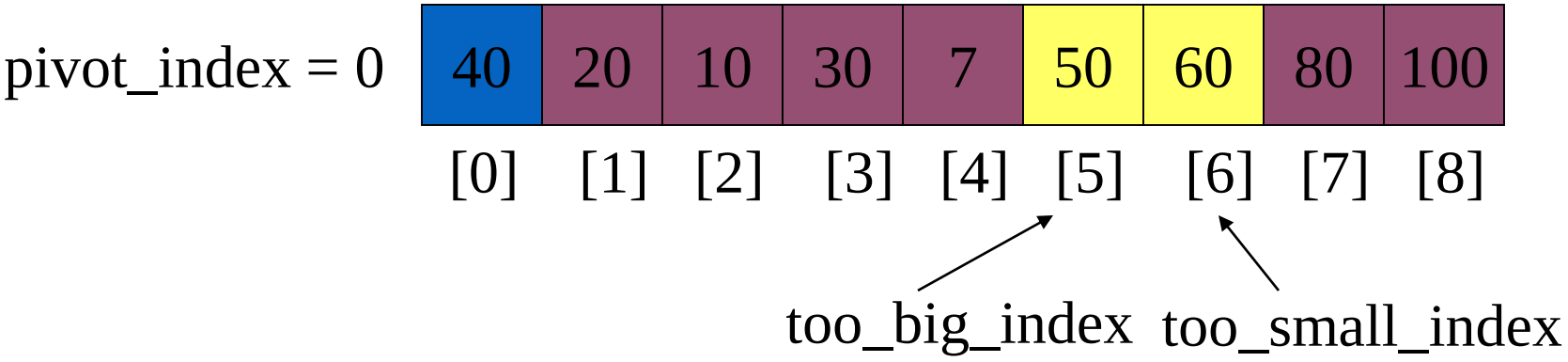
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



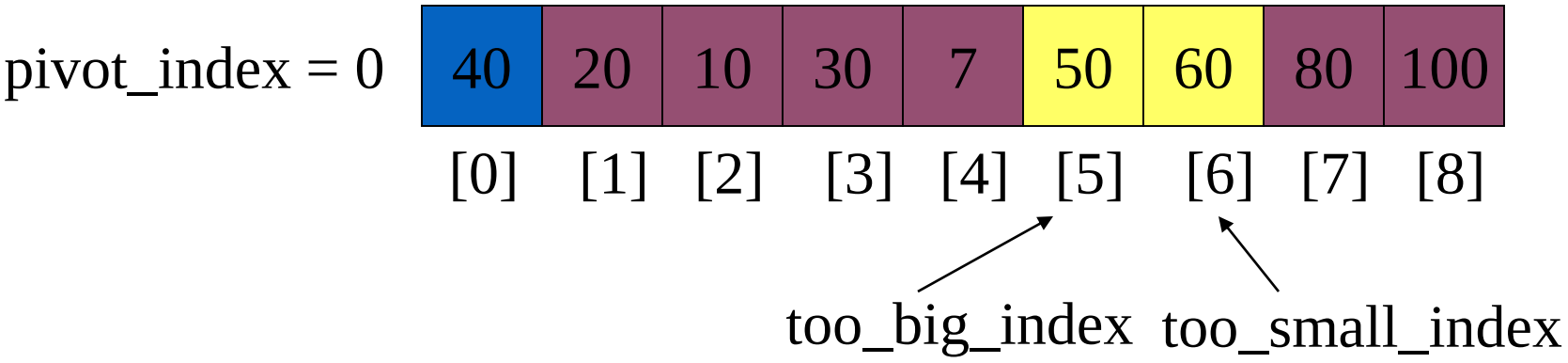
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



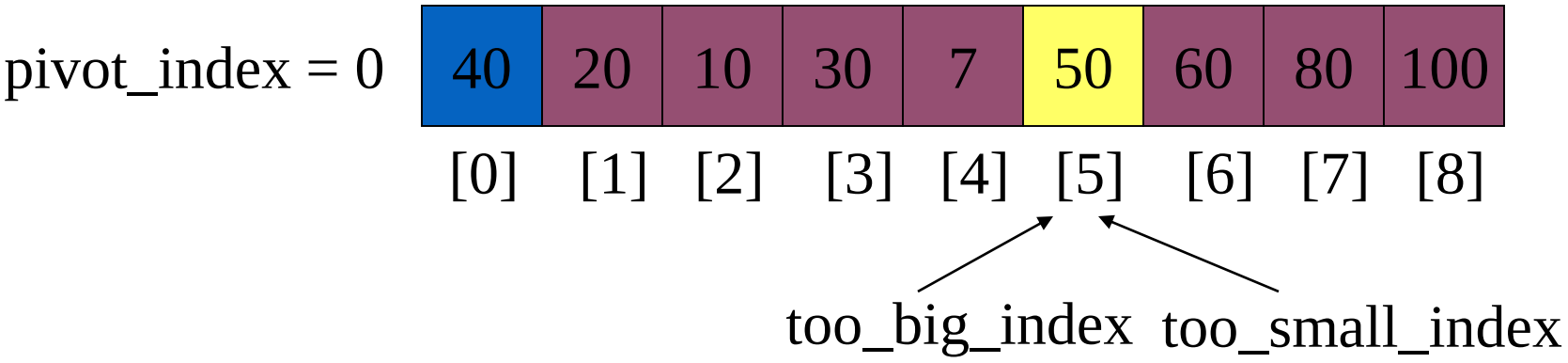
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



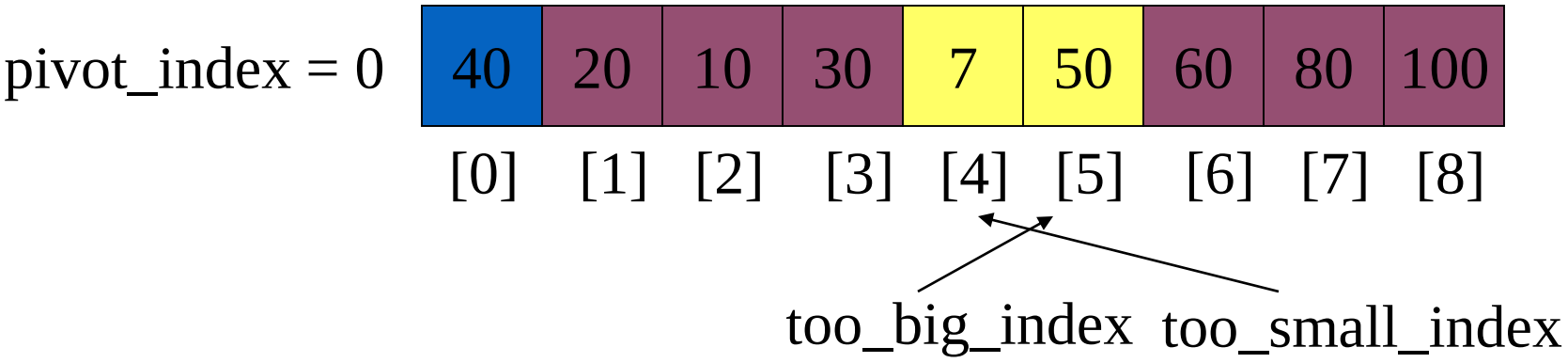
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



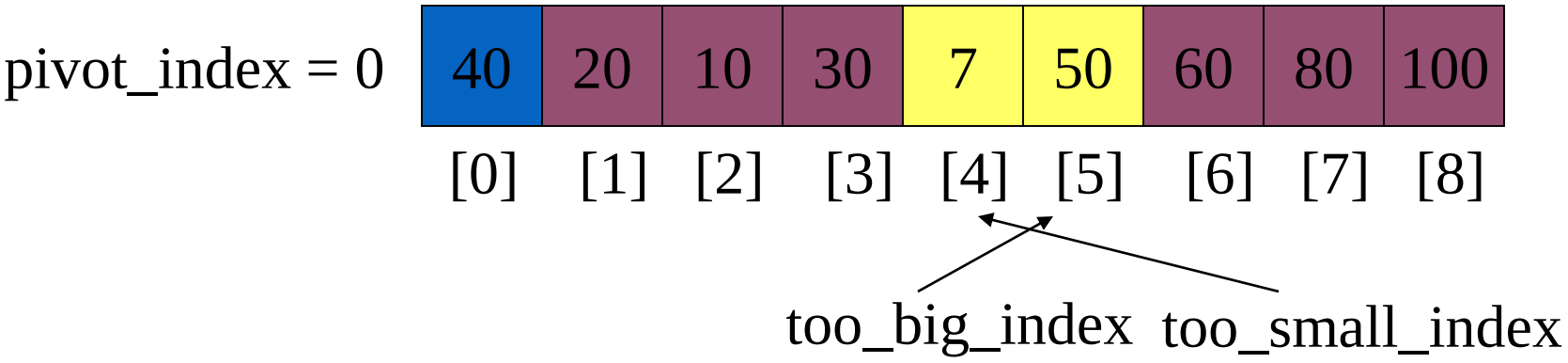
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



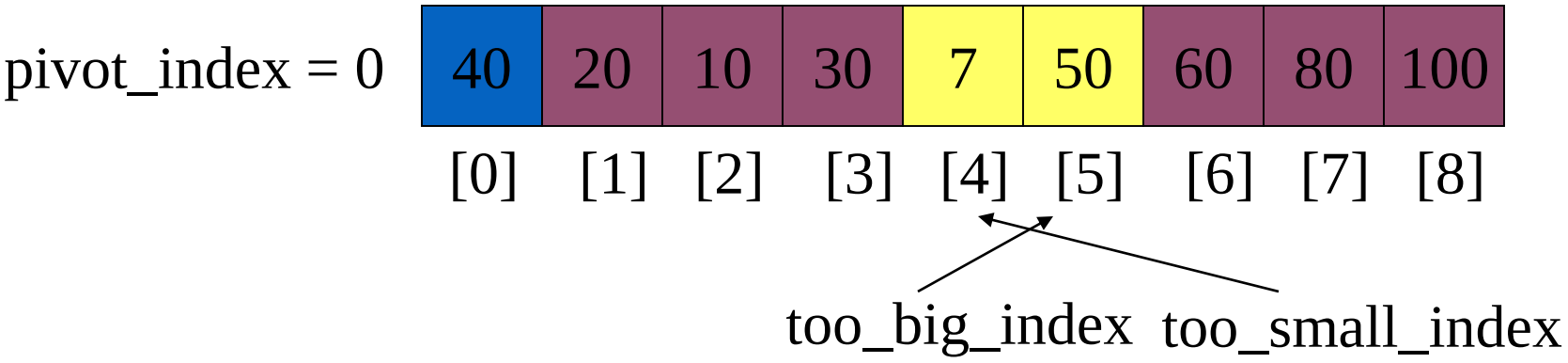
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



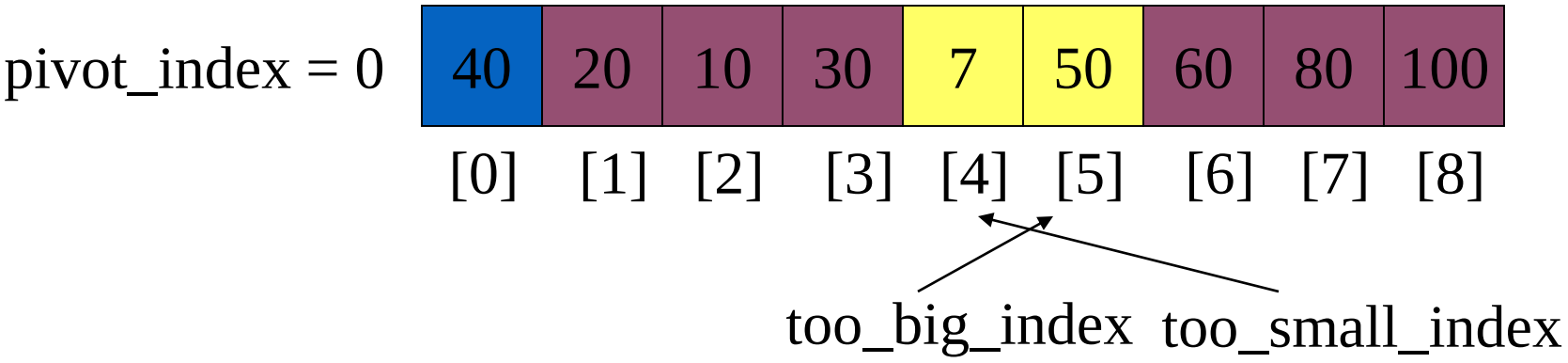
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



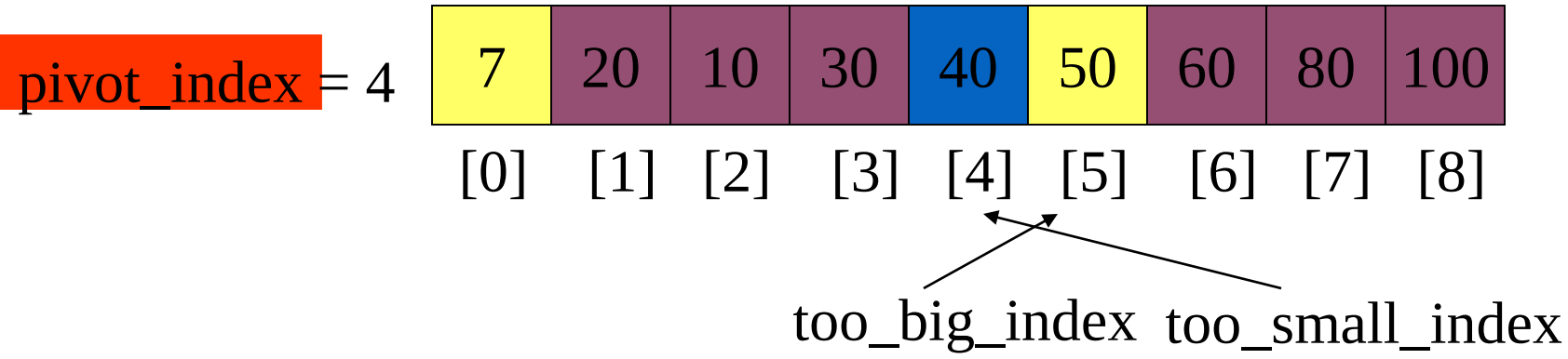
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



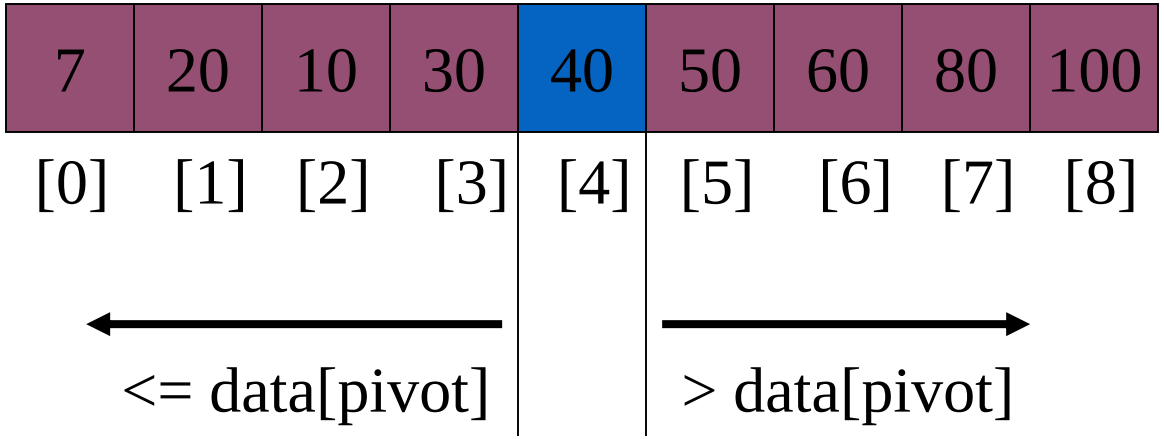
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



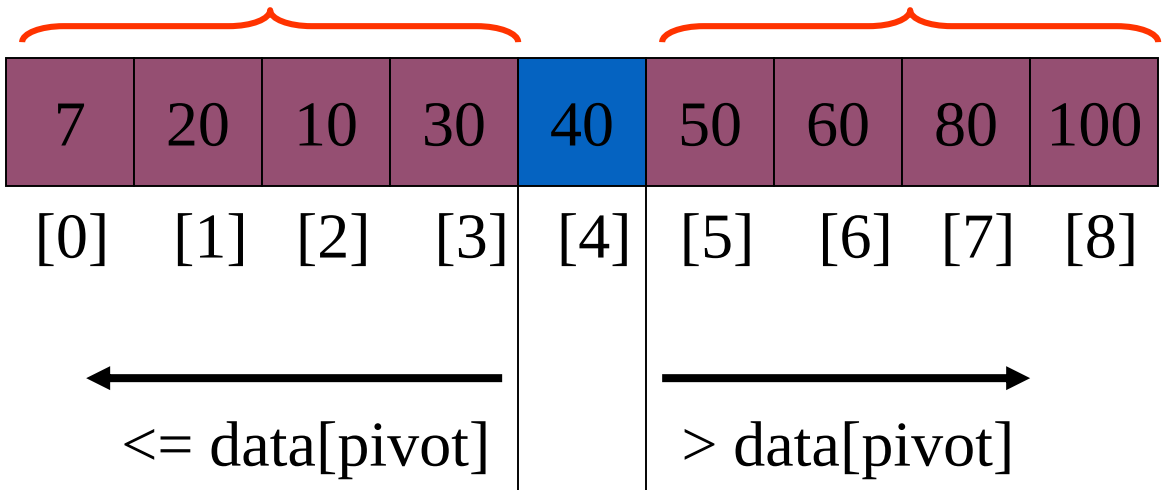
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



Partition Result



Recursion: Quicksort Sub-arrays



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

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

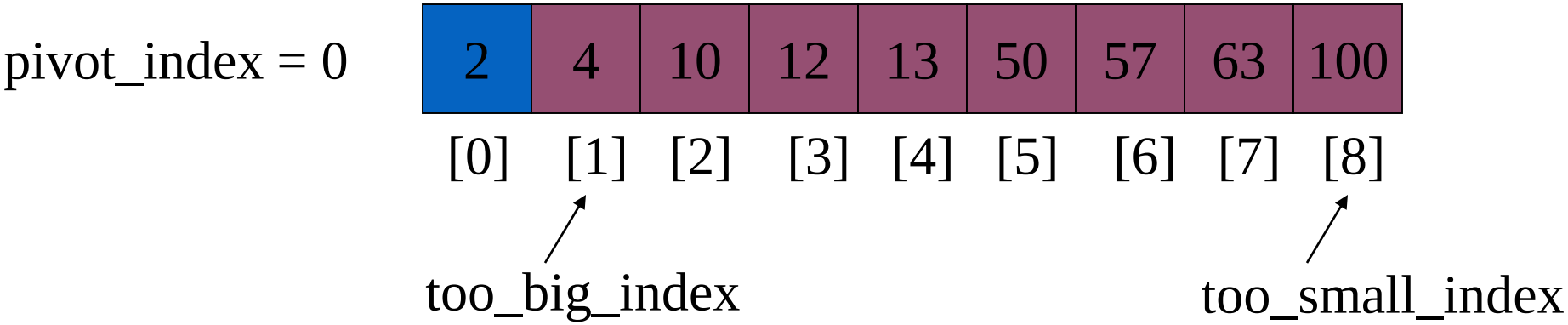
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

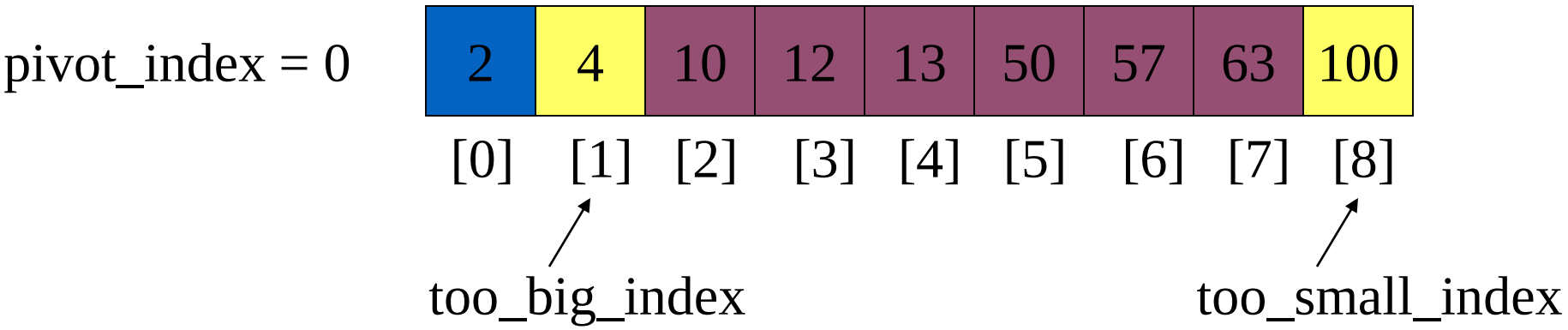
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Quicksort: Worst Case

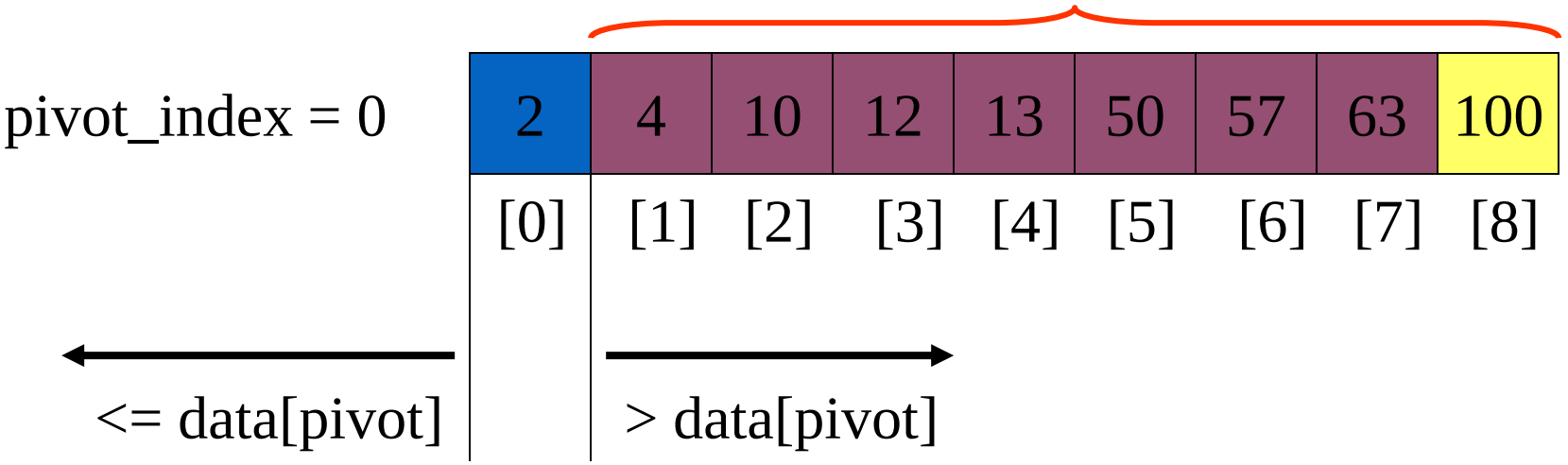
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



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

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- 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].

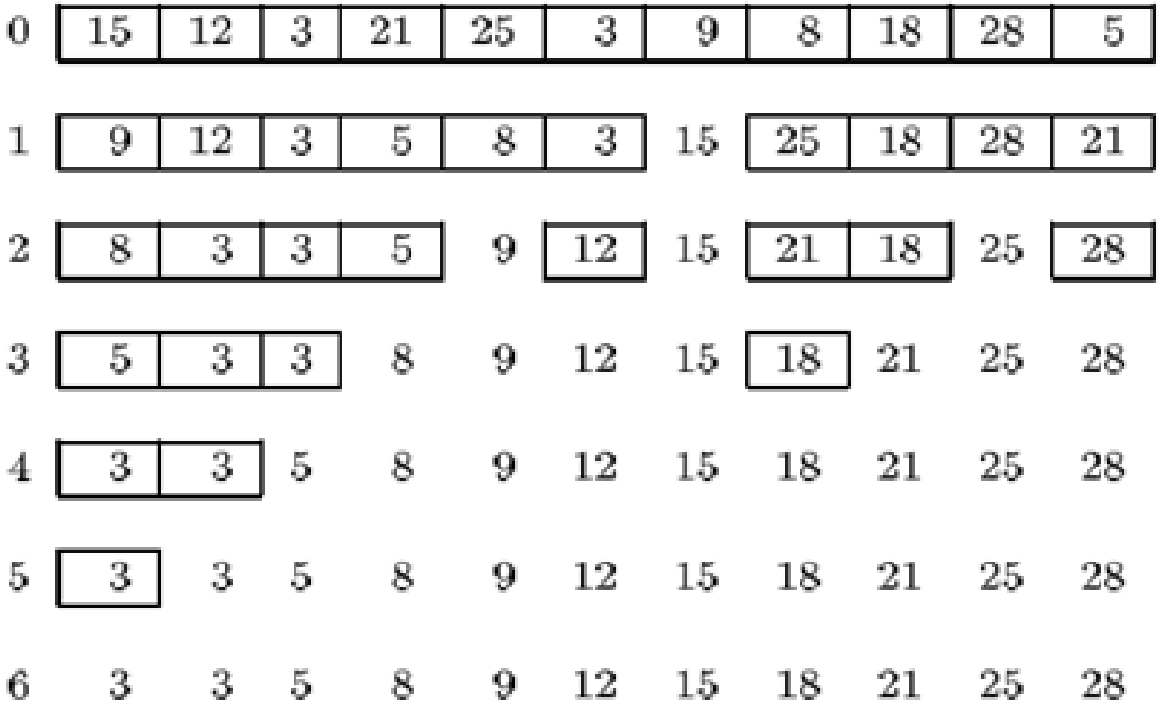
Use this median value as pivot.

Quicksort Analysis

- Average case is rather complex, but is where the algorithm earns its name. The bottom line is: $O(n \lg n)$

Quicksort Example

- Recursive implementation with the left most array entry selected as the pivot element.



Question

- Determine the circular shift in a sorted list

- Find the middle value of the array a.
- If a[0] < a[mid], then all values in the first half of the array are sorted.
- If a[mid] < a[last], then all values in the second half of the array are sorted.