Sorting Algorithms

Sorting

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

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

- We can use an ***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.

- We will analyze only internal sorting algorithms.

- A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons.

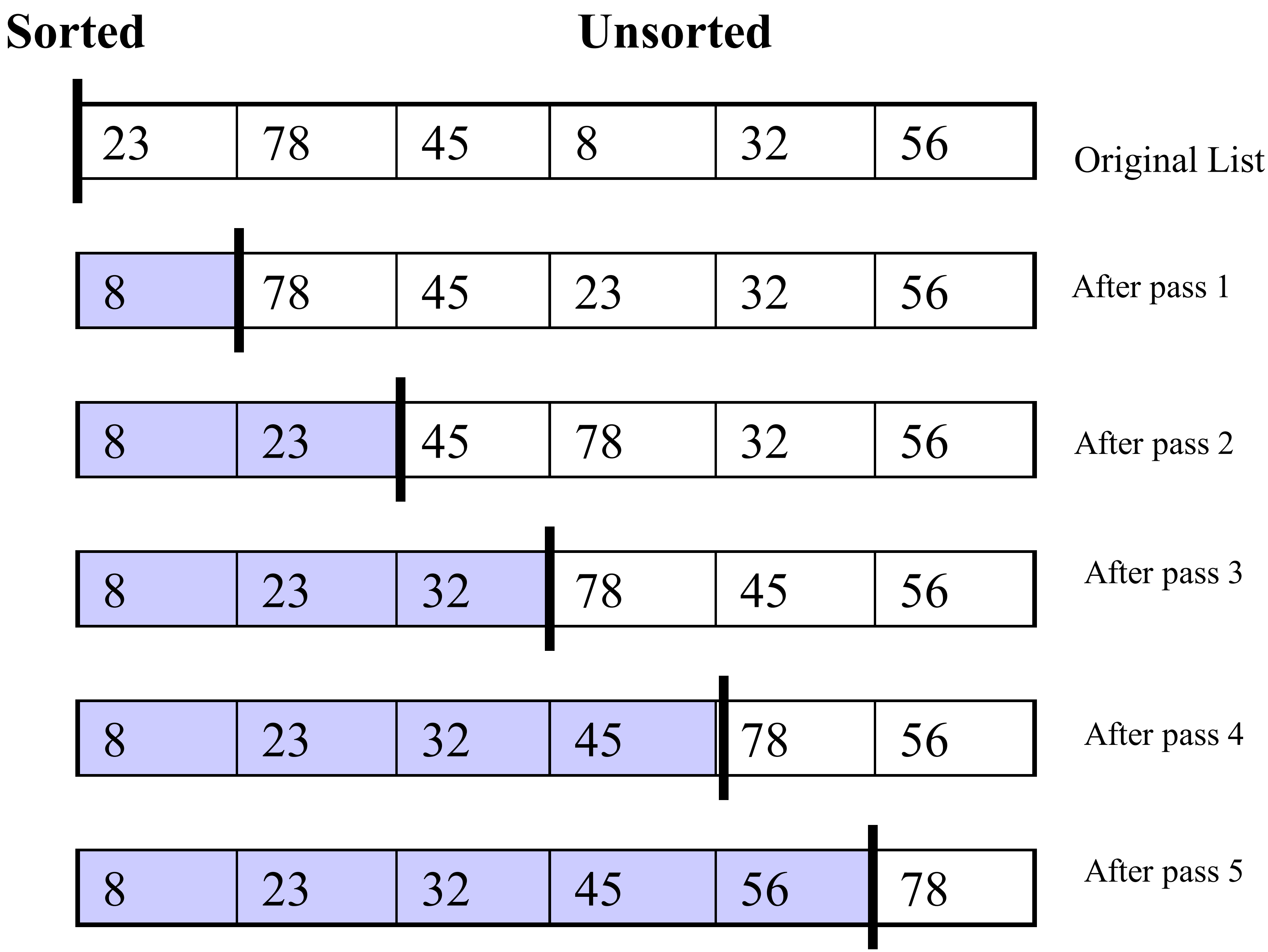
Sorting Algorithms

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

- The first three are the foundations for faster and more efficient algorithms.

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.)

```
void selectionSort( int 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;

    int tmp = a[i];
    a[i] = a[min];
    a[min] = tmp;
 }
}
```

Selection Sort -- Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

  → **So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.**

  - Ignoring other operations does not affect our final result.

- In selectionSort function, the outer for loop executes n-1 times.
- We invoke swap function once at each iteration.

  → Total Swaps: n-1

  → Total Moves: 3*(n-1) (Each swap has three moves)

Selection Sort – Analysis (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.

  ➔ # of key comparisons = 1+2+...+n-1 = n*(n-1)/2

  ➔ **So, Selection sort is O(n^2)**

- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **O(n^2)**

  – This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.

  – Since O(n^2) grows so rapidly, the selection sort algorithm is appropriate only for small n.

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.

- 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

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

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

        a[j] = tmp;
    }
}
```

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: 2*(n-1) ➔ O(n)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 2,3, …, n
  - The number of moves: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- Average-case: ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Insertion Sort is $O(n^2)$**

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

```
void bubleSort(int 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]{
            int temp=a[j];
            a[j]=a[j-1];
            a[j-1]=temp;
            sorted = false; // signal exchange
         }
    }
}
```

Bubble Sort – Analysis

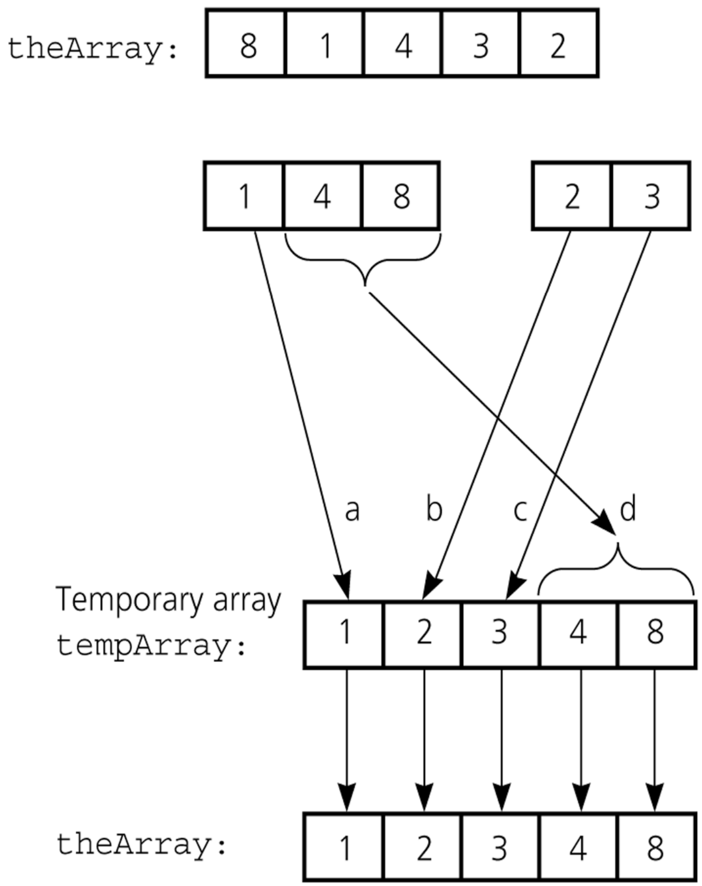
- ***Best-case:*** **➔ O(n)**
  - Array is already sorted in ascending order.
  - The number of moves: 0 ➔ O(1)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** **➔ $O(n^2)$**
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- Average-case: **➔ $O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is $O(n^2)$**

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



Divide the array in half

Sort the halves

Merge the halves:

a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

Merge

```
const int MAX_SIZE = maximum-number-of-items-in-array;
void merge(int theArray[], int first, int mid, int last) {
   int tempArray[MAX_SIZE];      // temporary array
   int first1 = first;     // beginning of first subarray
   int last1 = mid;        // end of first subarray
   int first2 = mid + 1;   // beginning of second subarray
   int last2 = last;       // end of second subarray
   int index = first1; // next available location in tempArray
   for ( ; (first1 <= last1) && (first2 <= last2); ++index) {
      if (theArray[first1] < theArray[first2]) {
         tempArray[index] = theArray[first1];
         ++first1;
      }
      else {
          tempArray[index] = theArray[first2];
          ++first2;
      } }
```

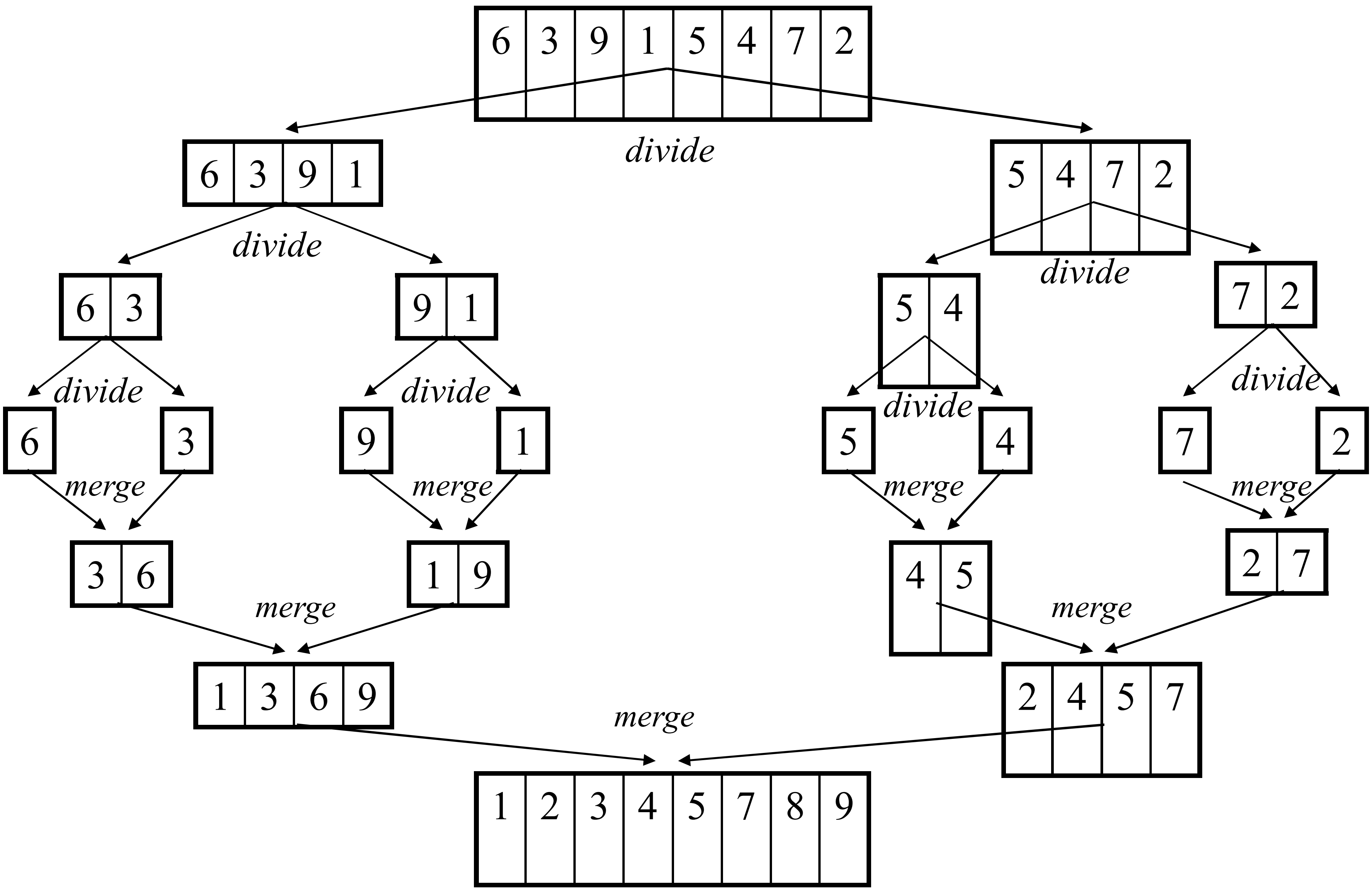
Merge (cont.)

```
    // finish off the first subarray, if necessary
    for (; first1 <= last1; ++first1, ++index)
        tempArray[index] = theArray[first1];

    // finish off the second subarray, if necessary
    for (; first2 <= last2; ++first2, ++index)
        tempArray[index] = theArray[first2];

    // copy the result back into the original array
    for (index = first; index <= last; ++index)
        theArray[index] = tempArray[index];
}   // end merge
```

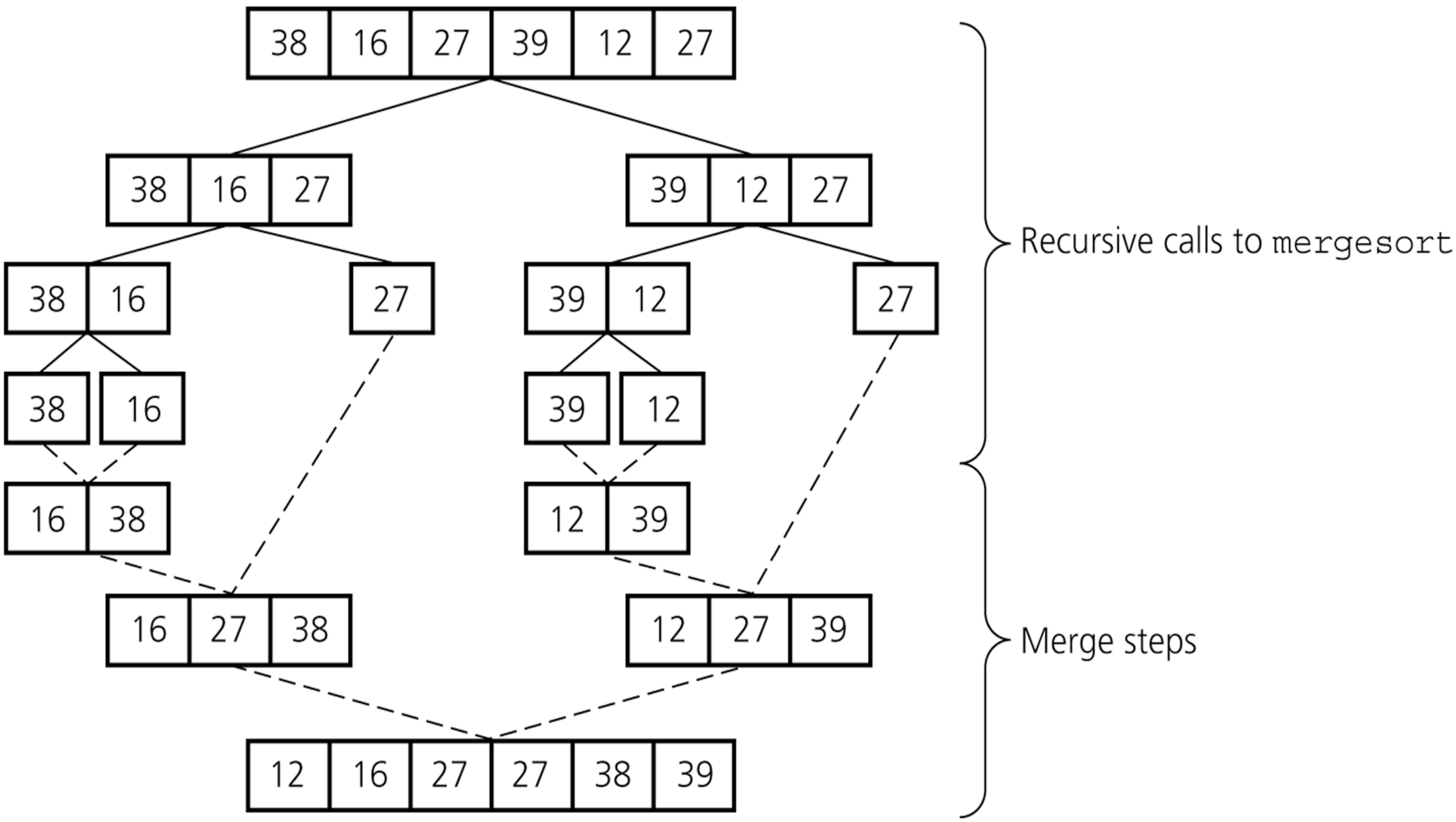
Mergesort

```
void mergesort(int theArray[], int first, int last) {
   if (first < last) {
      int mid = (first + last)/2;       // index of midpoint
      mergesort(theArray, first, mid);
      mergesort(theArray, mid+1, last);

      // merge the two halves
      merge(theArray, first, mid, last);
   }
}  // end mergesort
```

Mergesort - Example

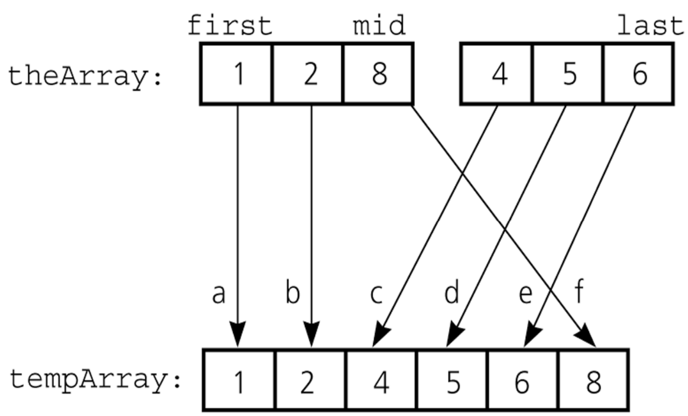


Mergesort – Example2



Mergesort – Analysis of Merge

A worst-case instance of the merge step in *mergesort*

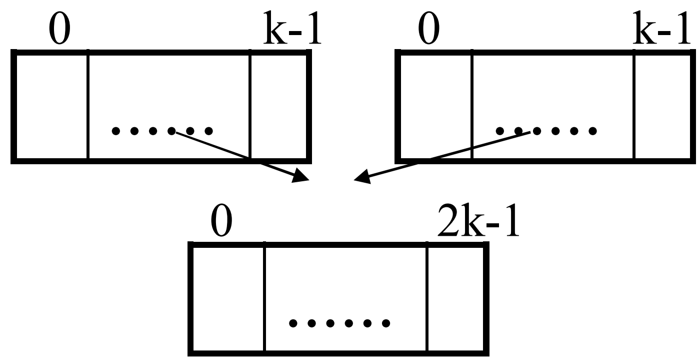


Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

Mergesort – Analysis of Merge (cont.)

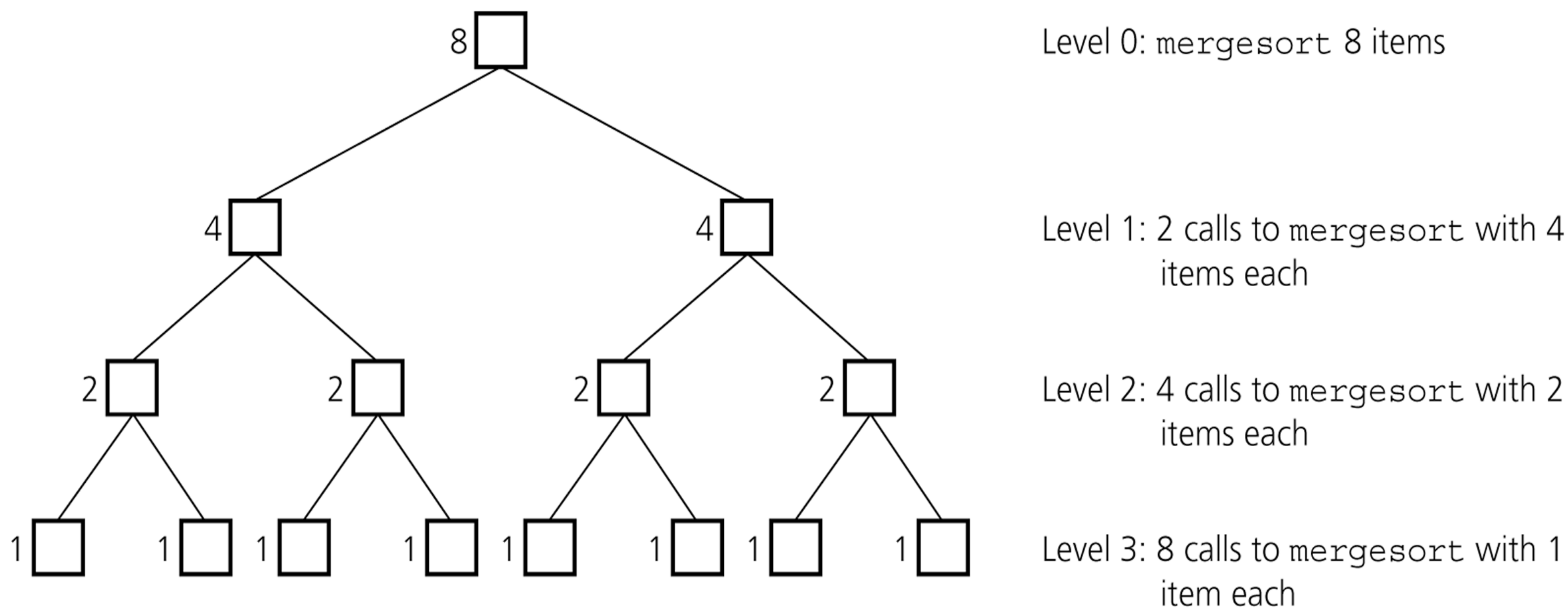
Merging two sorted arrays of size ***k***



- ***Best-case:***
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves: 2k
  - The number of key comparisons: k
- ***Worst-case:***
  - The number of moves: 2k
  - The number of key comparisons: 2k-1

Mergesort - Analysis

Levels of recursive calls to *mergesort*, given an array of eight items



Mergesort – Recurrence Relation

$T(n) = 2\ T(n/2) + cn$
$= 2\ [\mathbf{2\ T(n/4) + cn/2}] + cn$
$= 4\ T(n/4) + 2cn$
$= 4\ [\mathbf{2\ T(n/8) + cn/4}] + 2cn$
$= 8\ T(n/8) + 3cn$
…
…
$= 2^k\ T(n/2^k) + kcn$

We know that $T(1) = 1$

Putting $n/2^k = 1$, we get $n = 2^k$ *OR* $\log_2 n = k$

Hence,
$T(n)=nT(1)+cn\log_2 n = n + cn\log_2 n = O(\log n)$

Sorting Algorithms-2

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

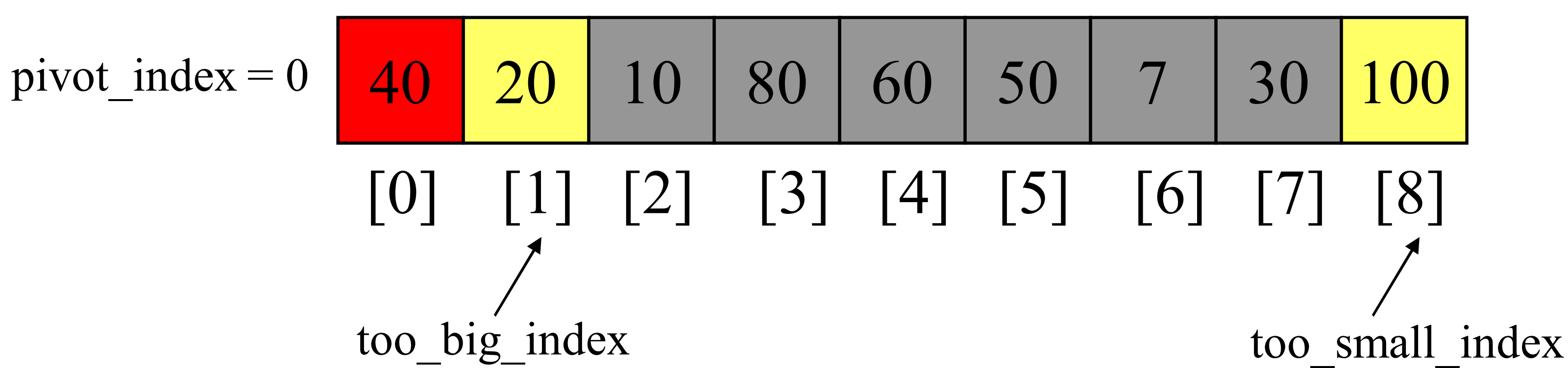
Partitioning Array

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

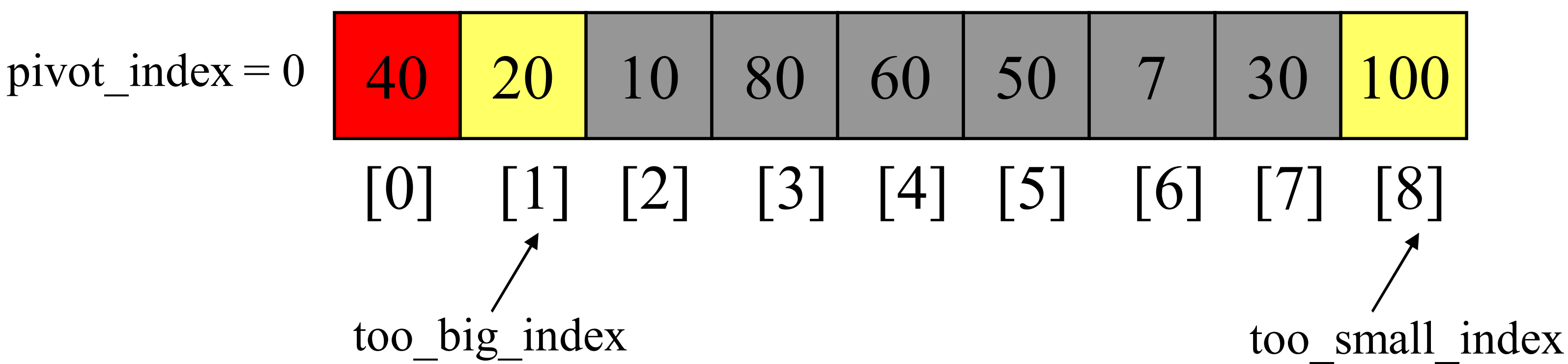
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

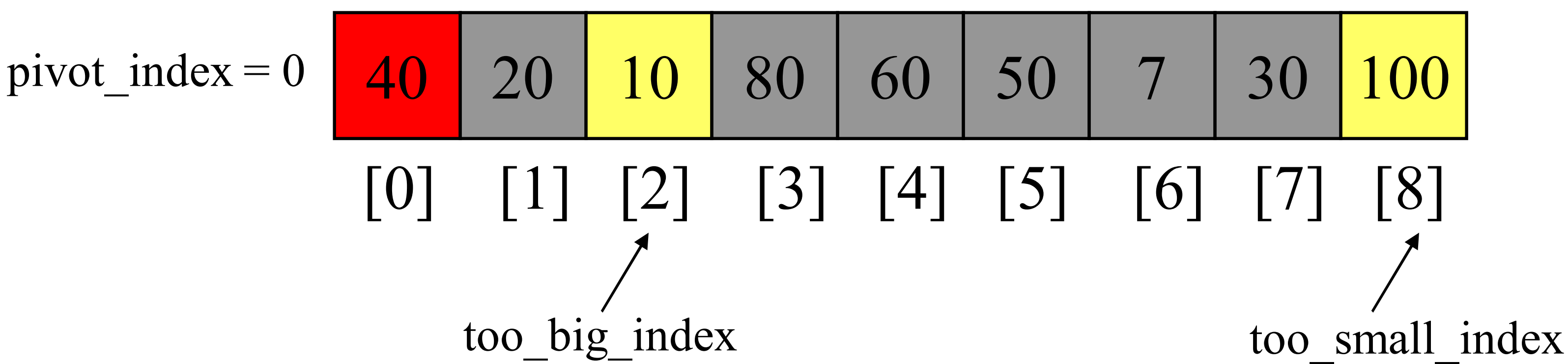
Partitioning loops through, swapping elements below/above pivot.


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

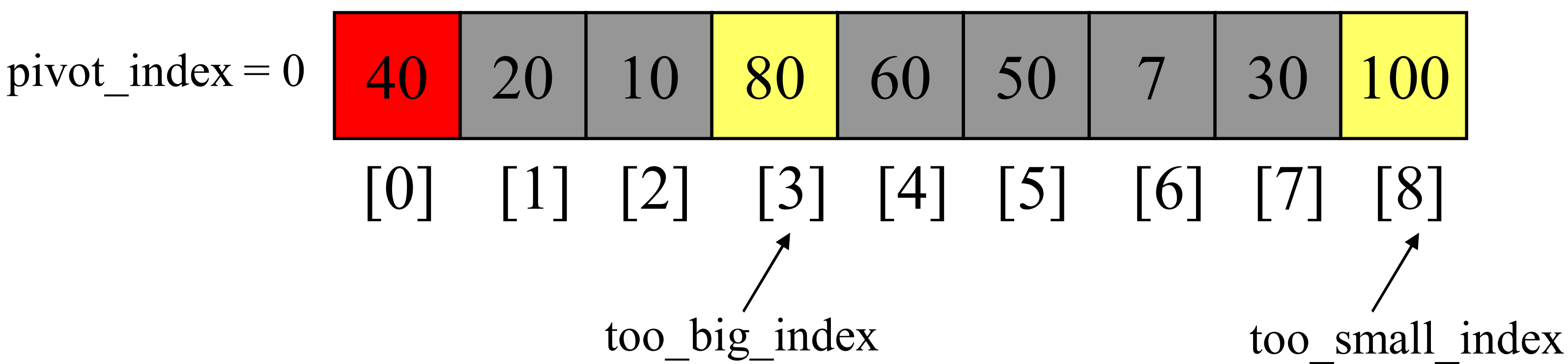
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



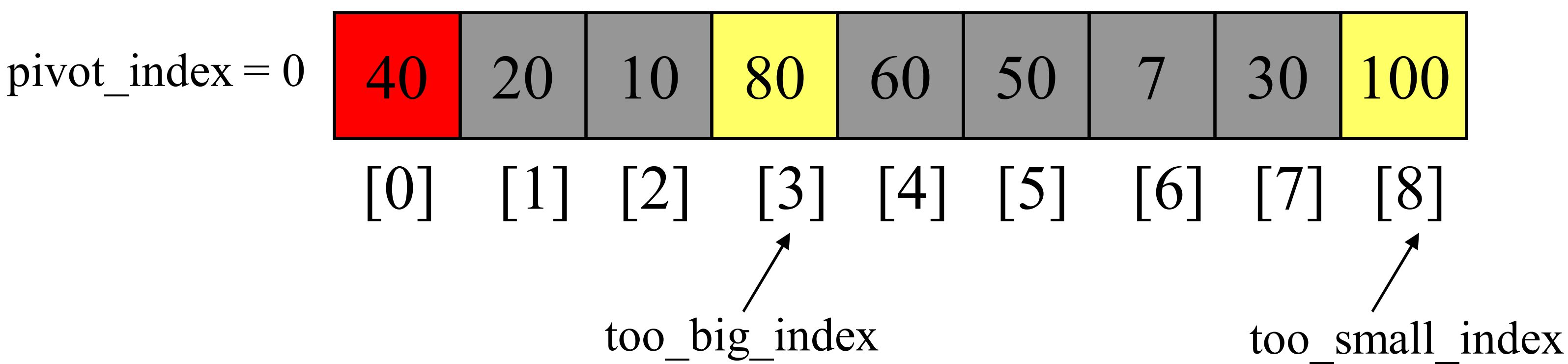
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



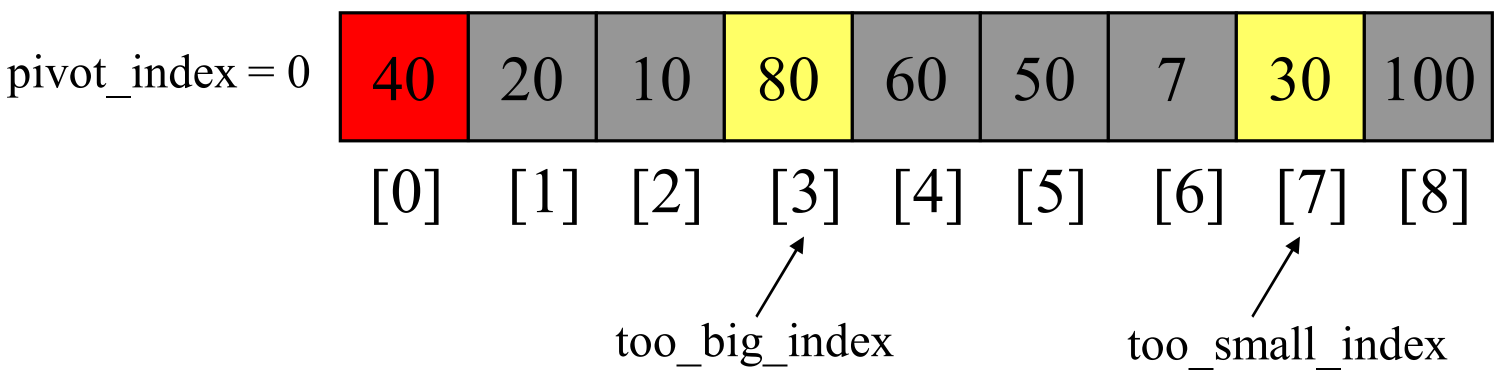
1. While data[too_big_index] <= data[pivot]
    ++too_big_index



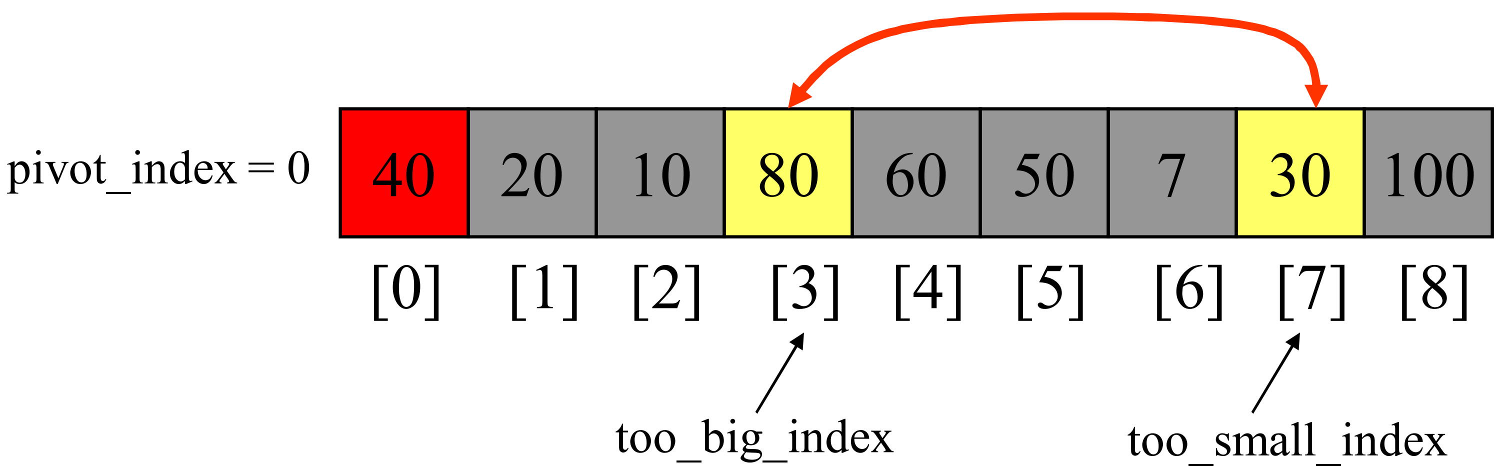
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



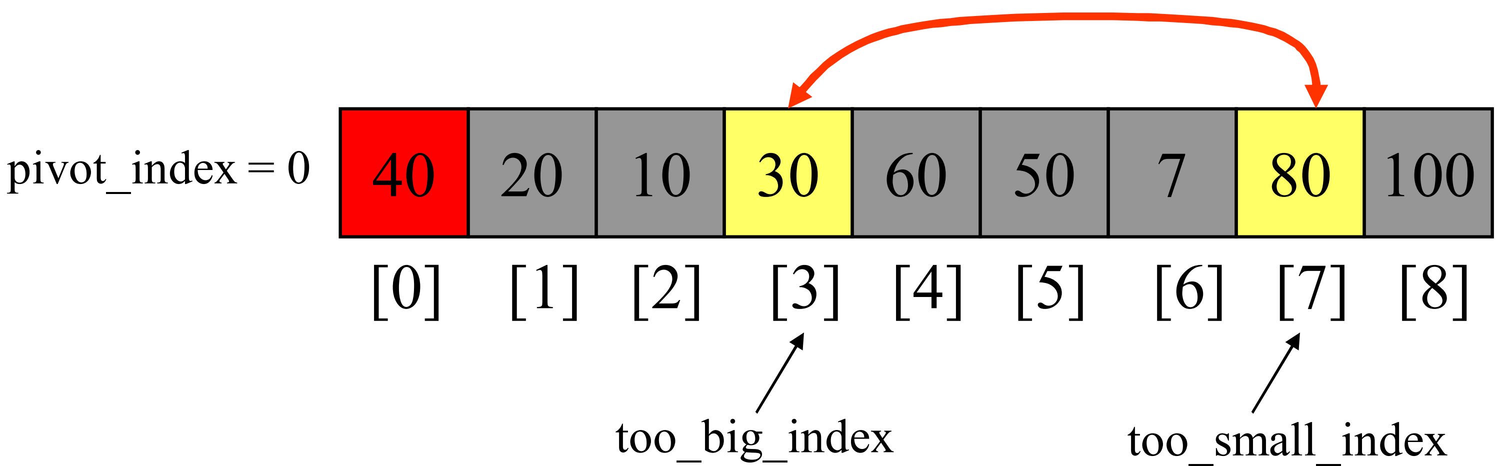
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



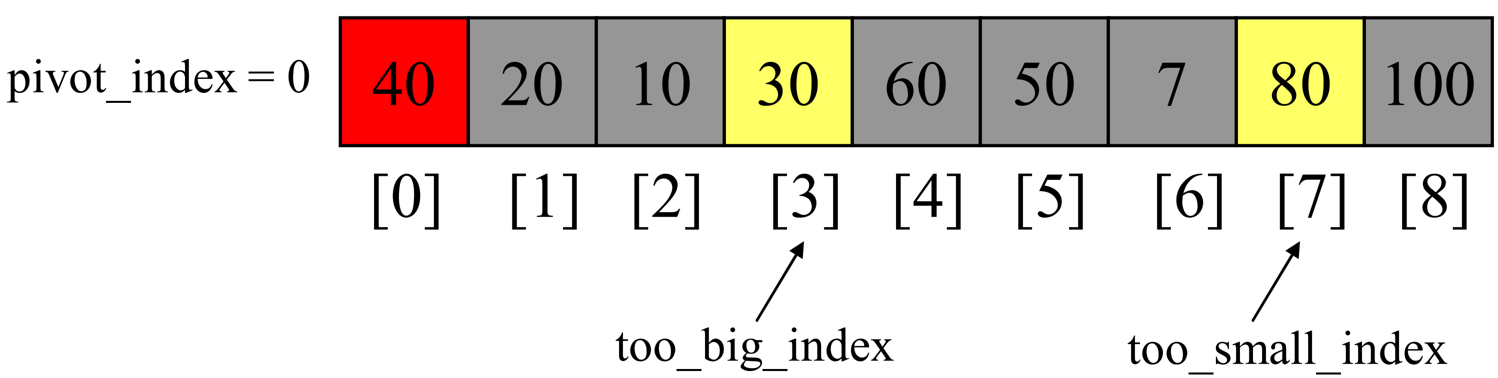
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



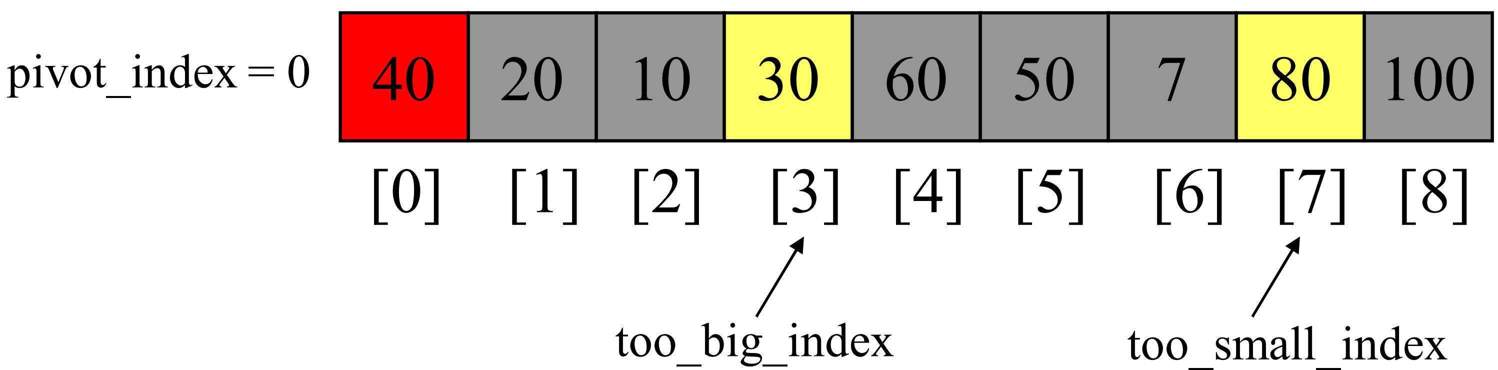
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



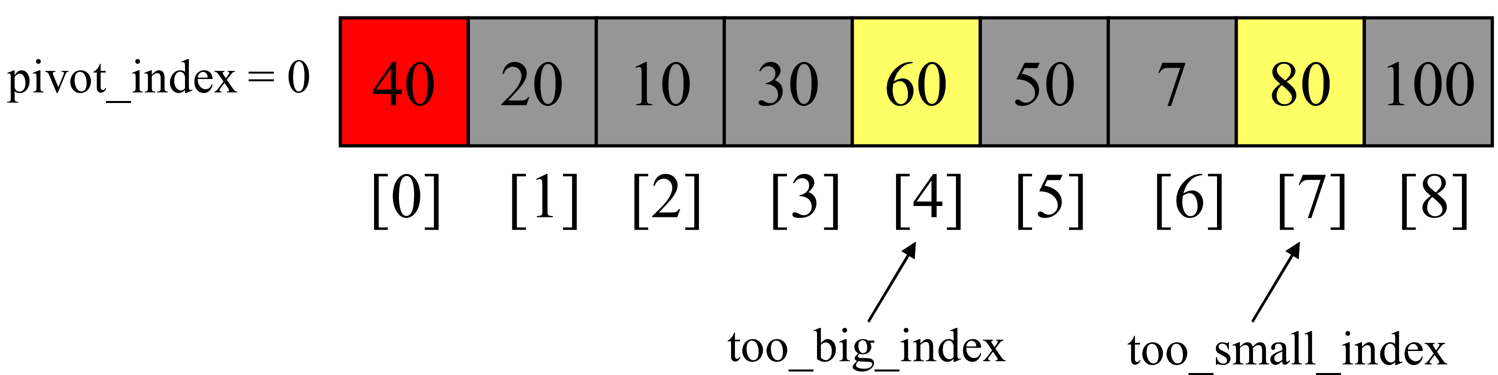
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



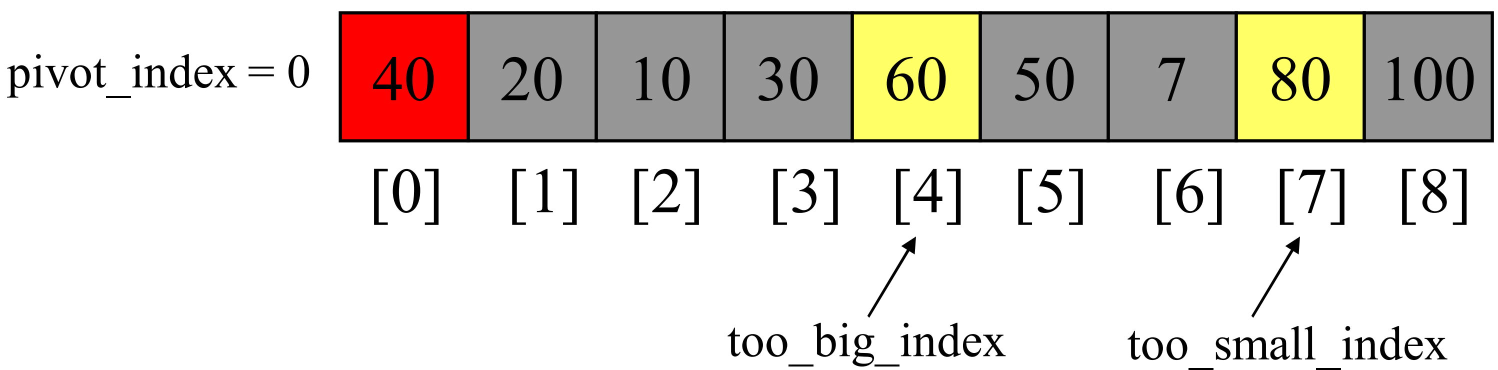
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



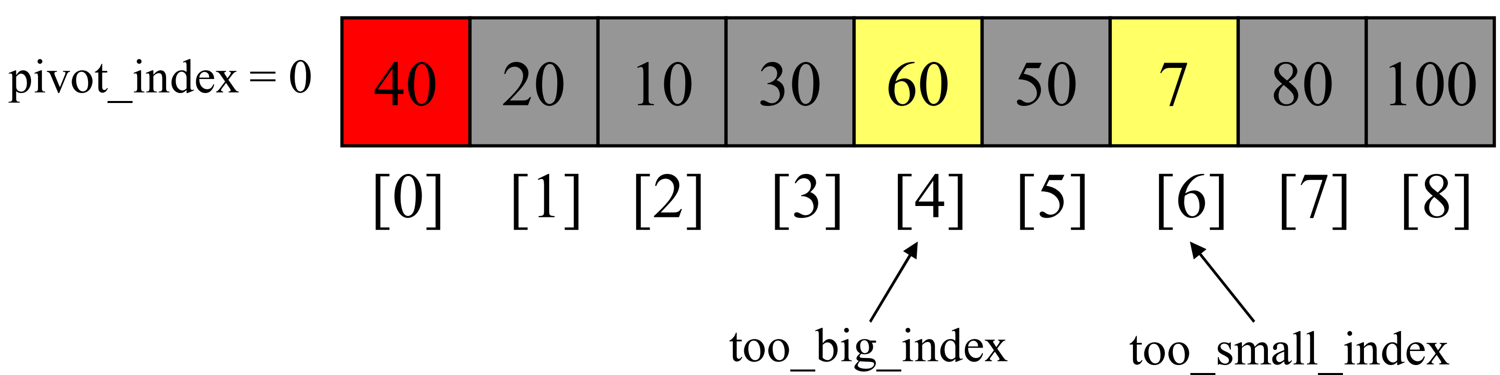
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



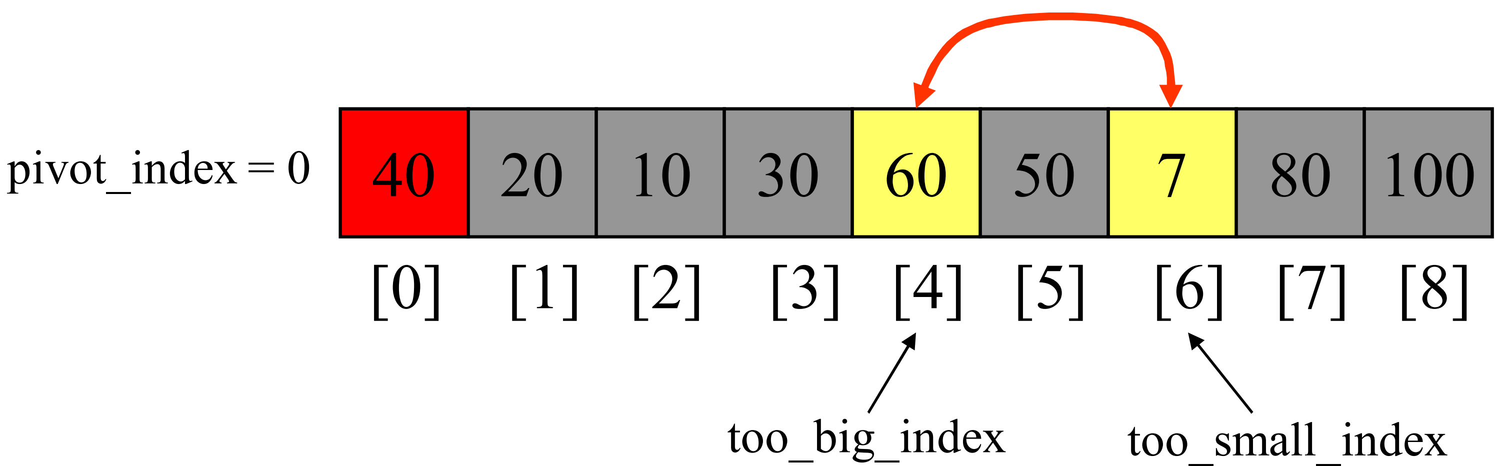
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



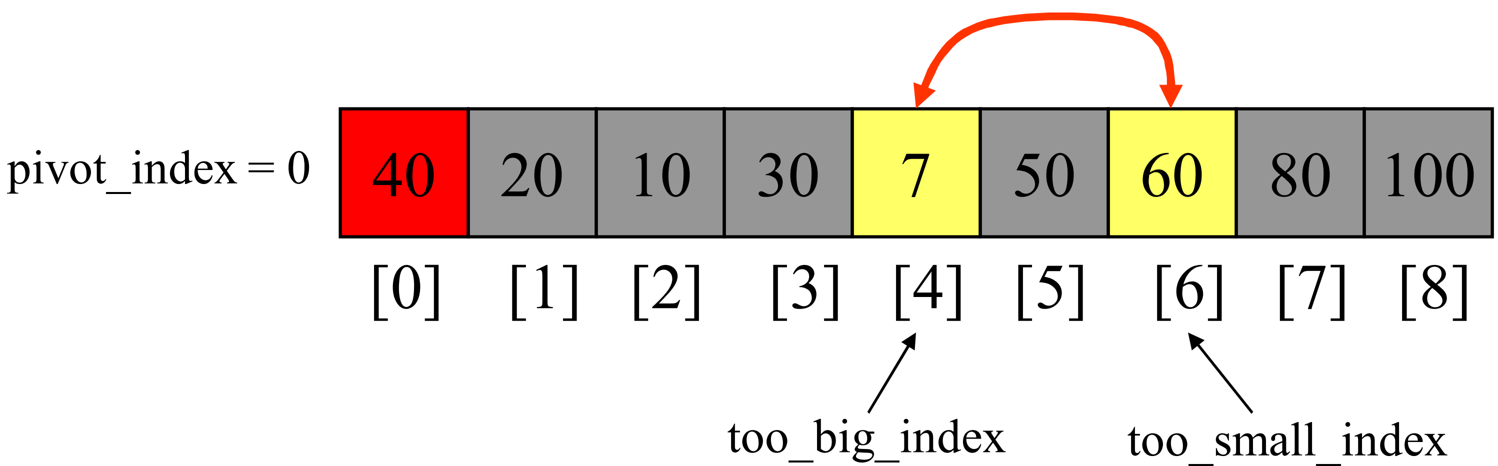
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



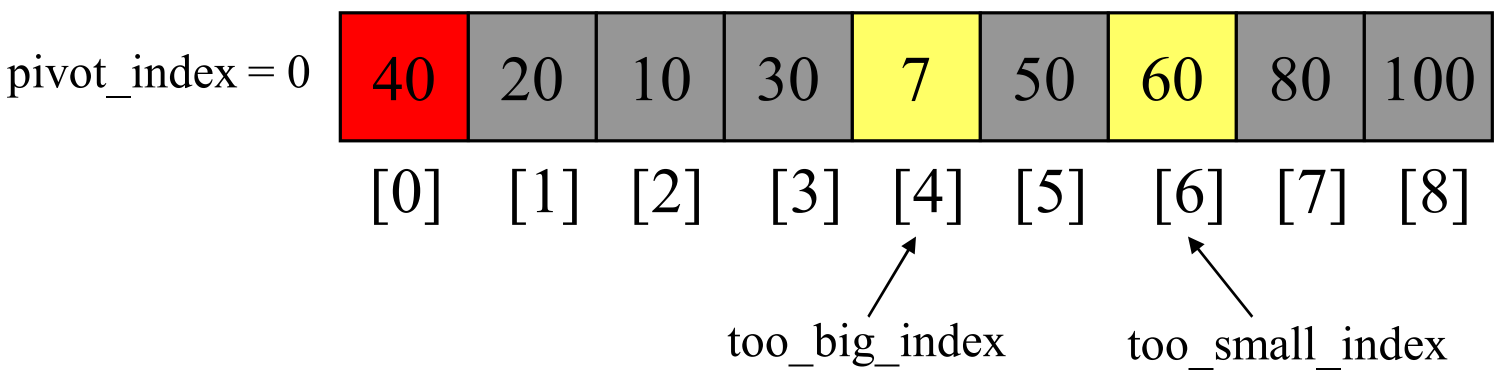
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



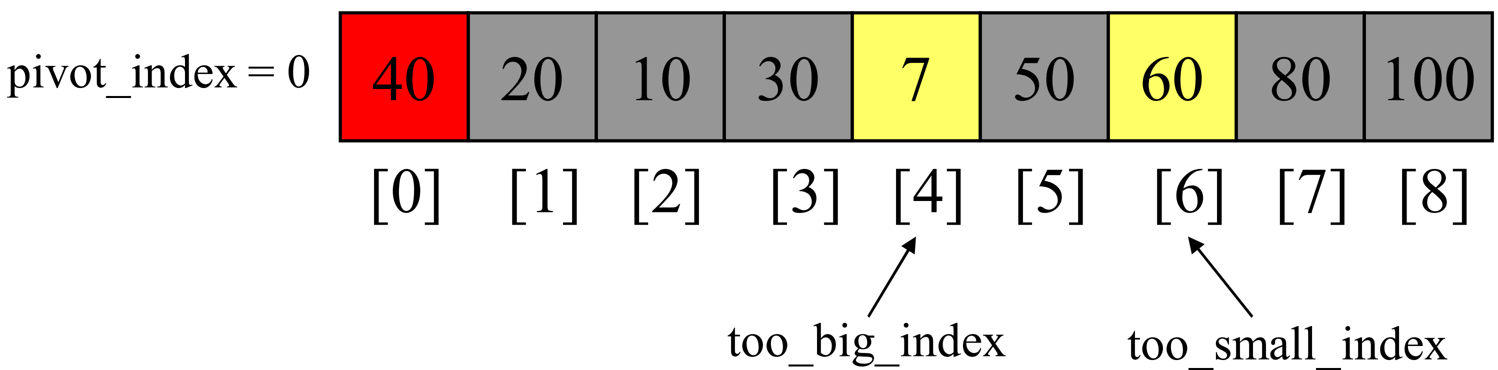
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



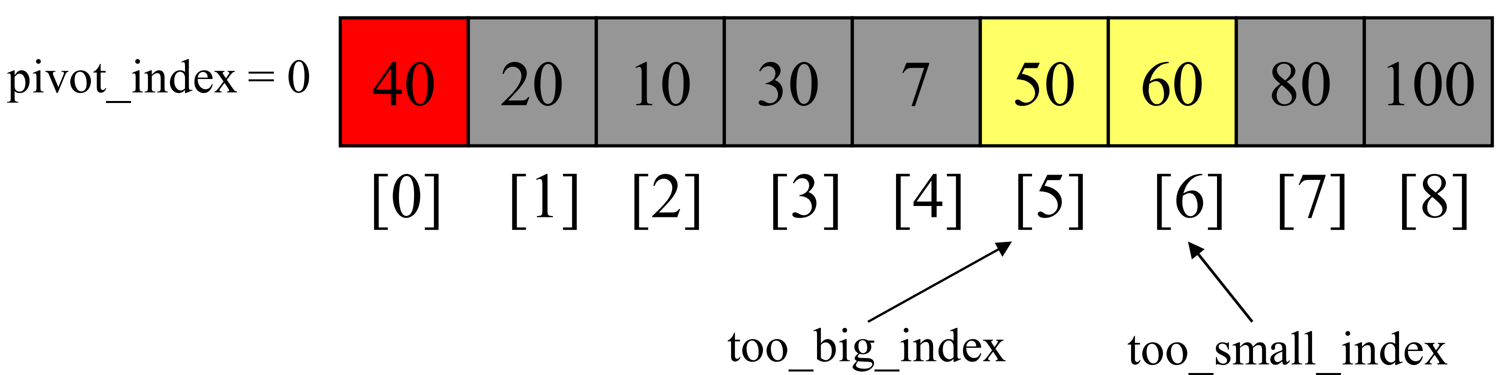
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



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.



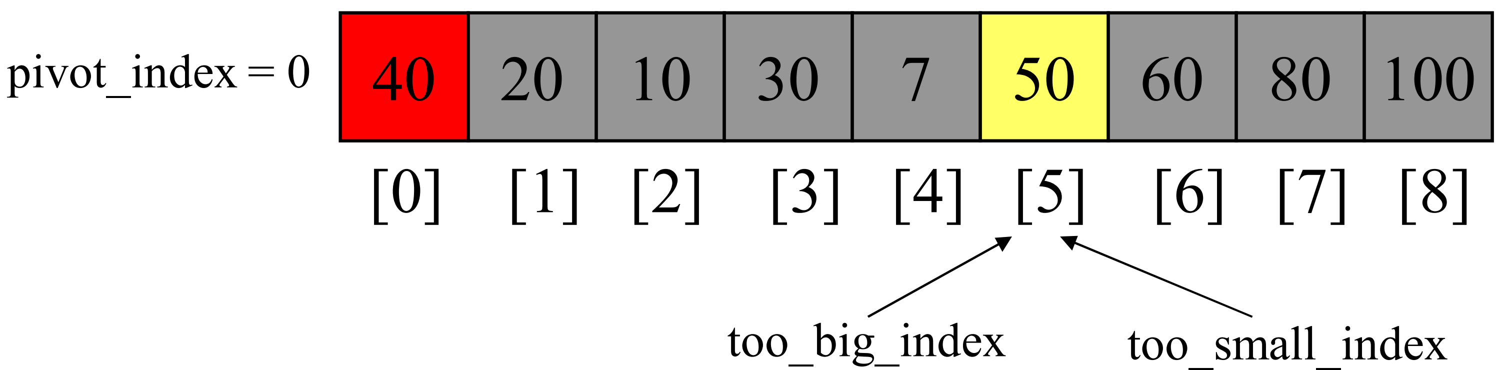
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

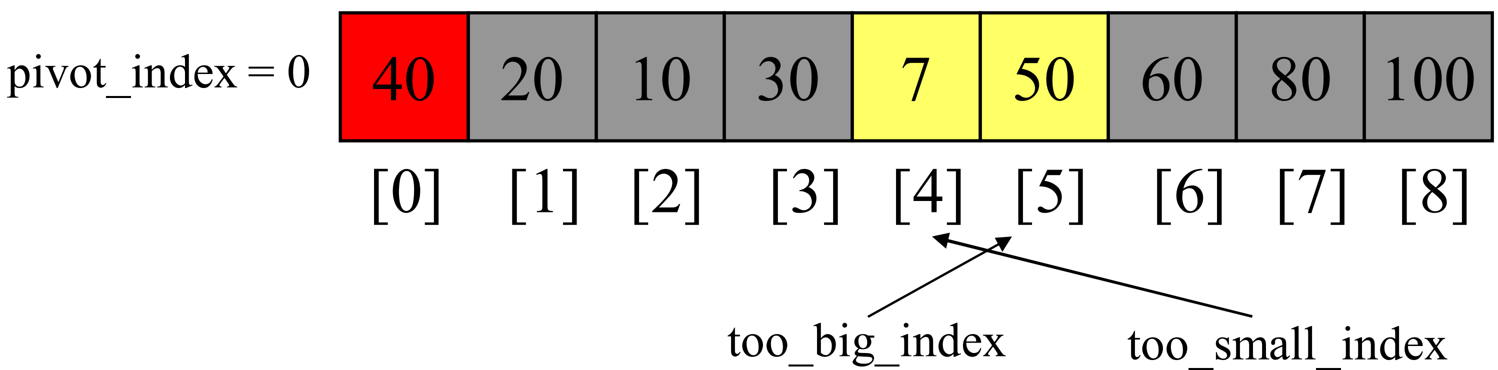
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

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

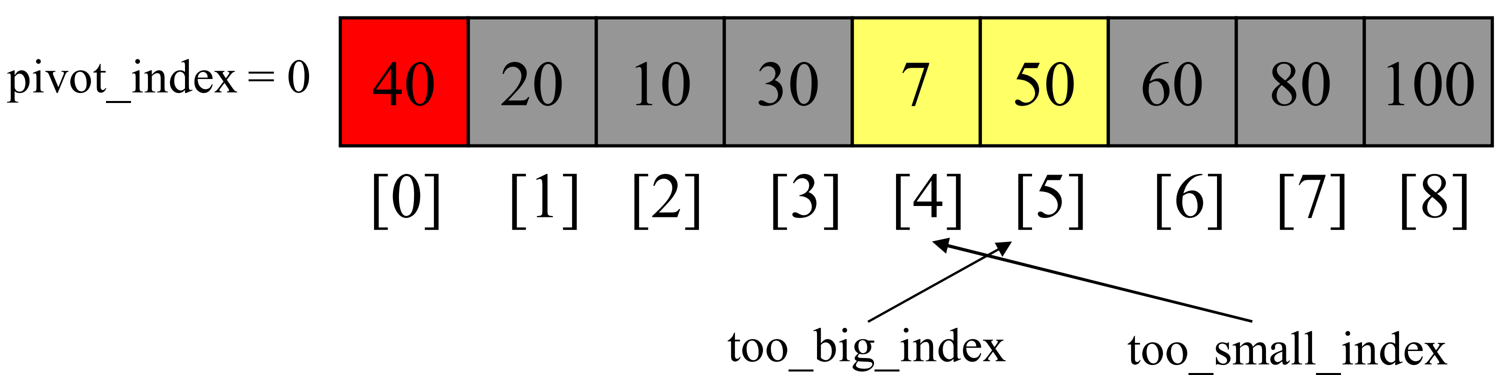
2. While data[too_small_index] > data[pivot]

    --too_small_index

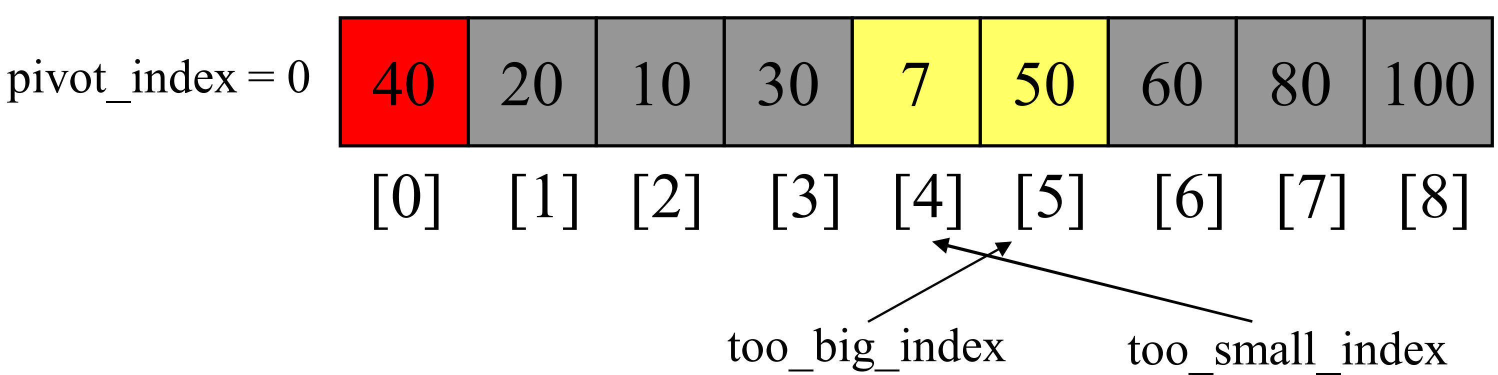
3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

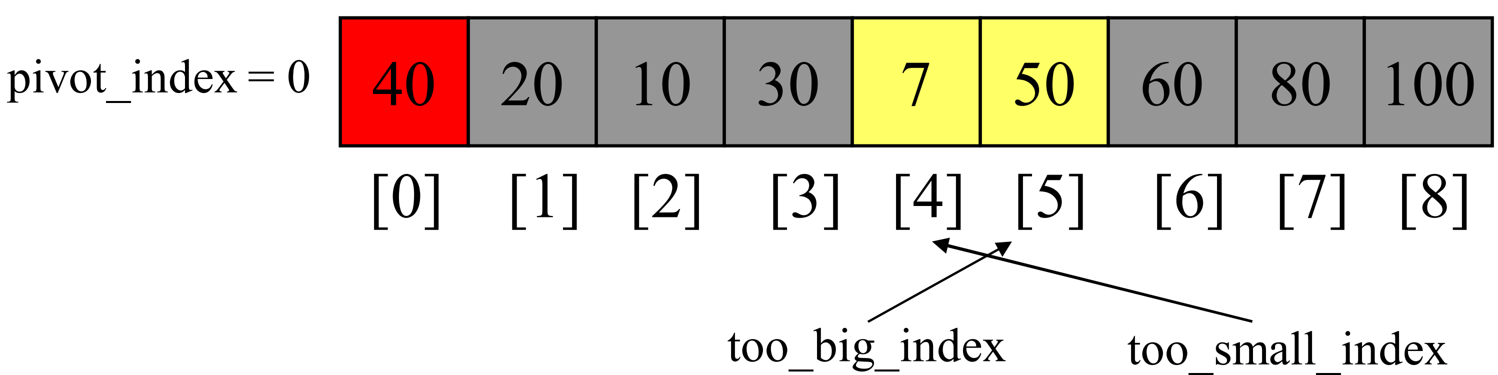
4. While too_small_index > too_big_index, go to 1.



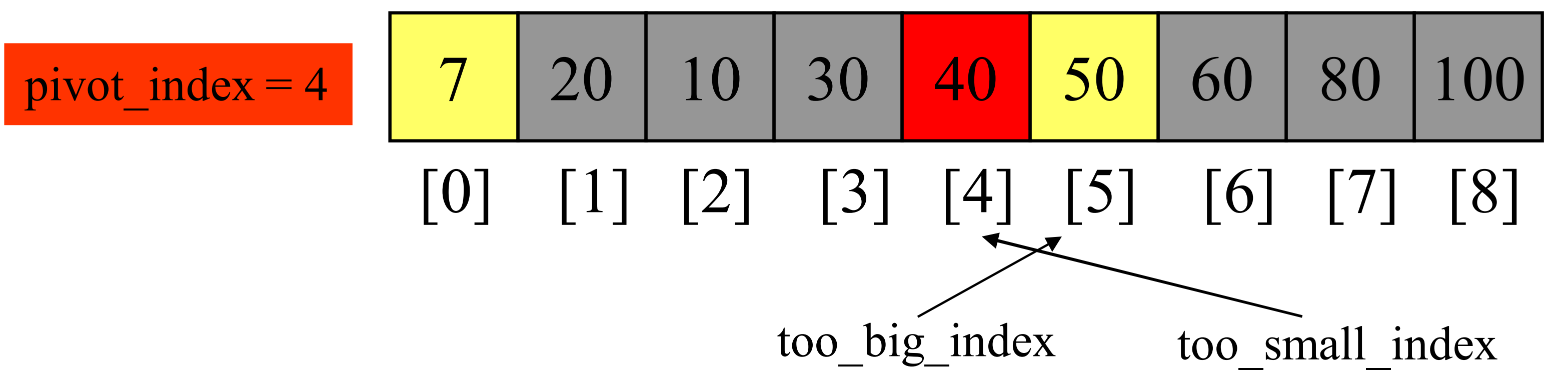
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



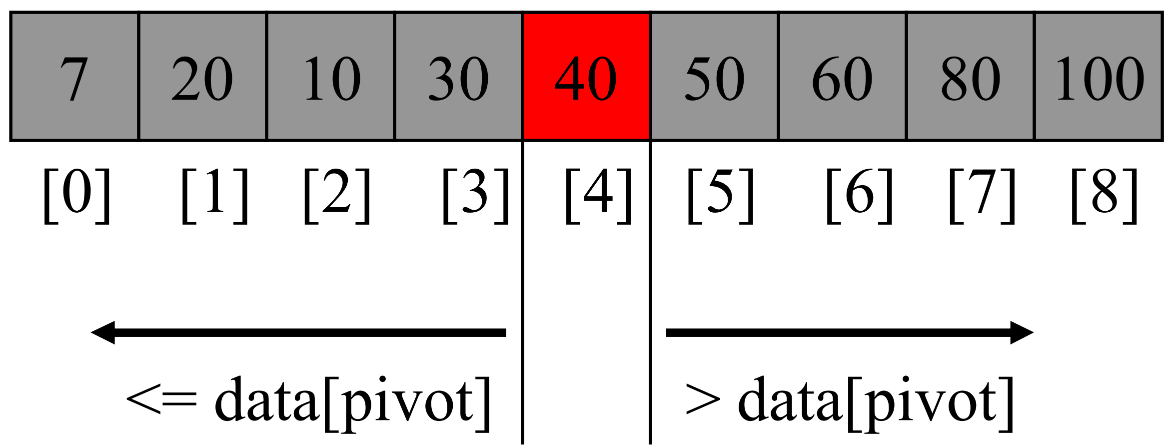
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



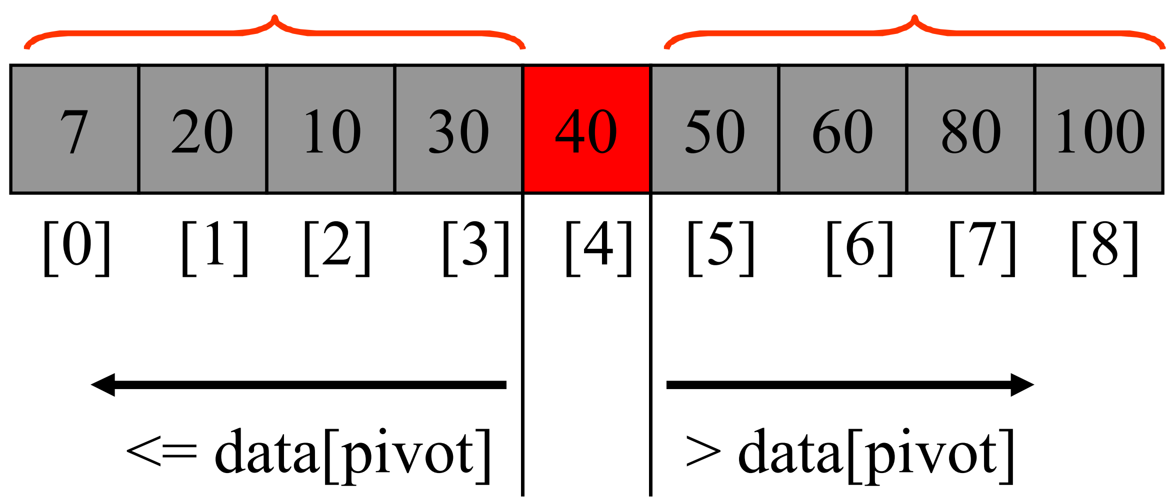
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



Partition Result



Recursion: Quicksort Sub-arrays



Complexity Analysis

$T(N) = T(i) + T(N - i - 1) + cN$

The time to sort the file is equal to

- the time to sort the left partition with **i** elements, plus
- the time to sort the right partition with **N-i-1** elements, plus
- the time to build the partitions.

Worst-Case Analysis

The pivot is the smallest (or the largest) element

$T(N) = T(N-1) + cN, N > 1$

Telescoping:

$$T(N-1) = T(N-2) + c(N-1)$$

$$T(N-2) = T(N-3) + c(N-2)$$

$$T(N-3) = T(N-4) + c(N-3)$$

....

$$T(2) = T(1) + c.2$$

Worst-Case Analysis

$T(N) + T(N-1) + T(N-2) + \ldots + T(2) =$

$= T(N-1) + T(N-2) + \ldots + T(2) + T(1) +$

$c(N) + c(N-1) + c(N-2) + \ldots + c.2$

$T(N) = T(1) +$

c times (the sum of 2 thru N)

$= T(1) + c\,(N\,(N+1)\,/\,2\,-1) =$ **$O(N^2)$**

Best-Case Analysis

The pivot is in the middle

$$T(N) = 2T(N/2) + cN$$

Like mergesort

$$T(N)=O(N \log N)$$

Quicksort: Worst Case

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

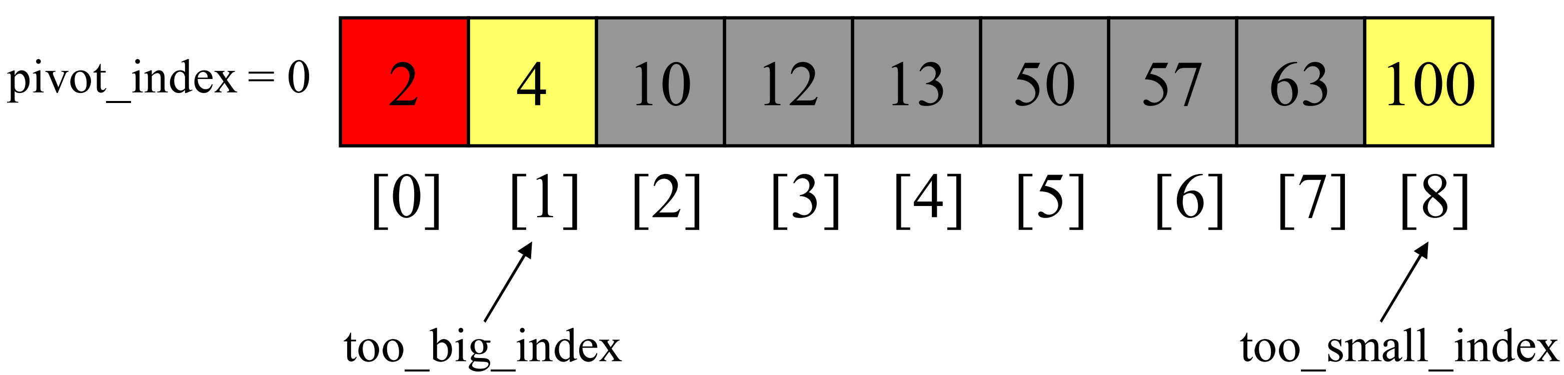
pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

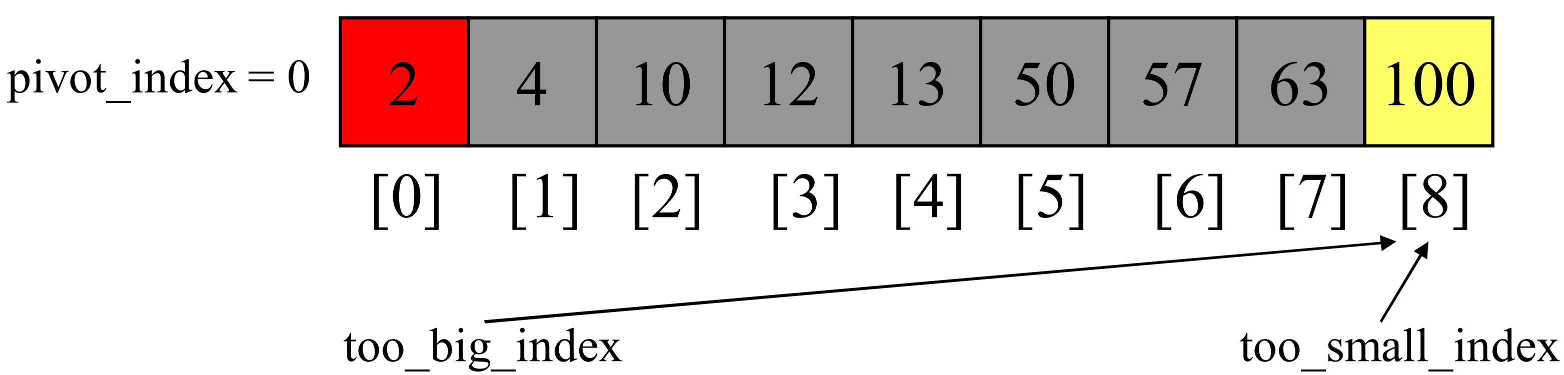
too_big_index

too_small_index

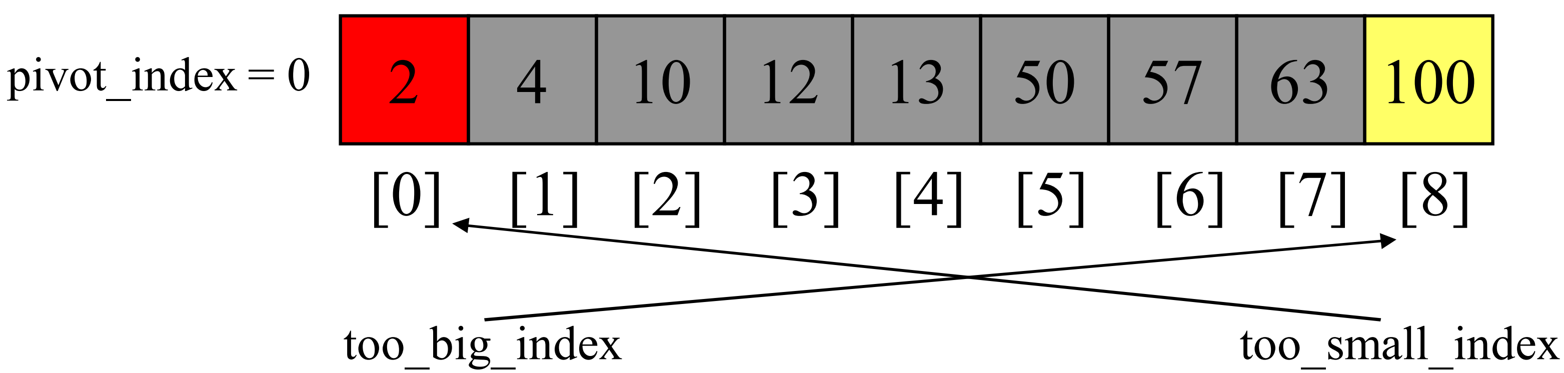
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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

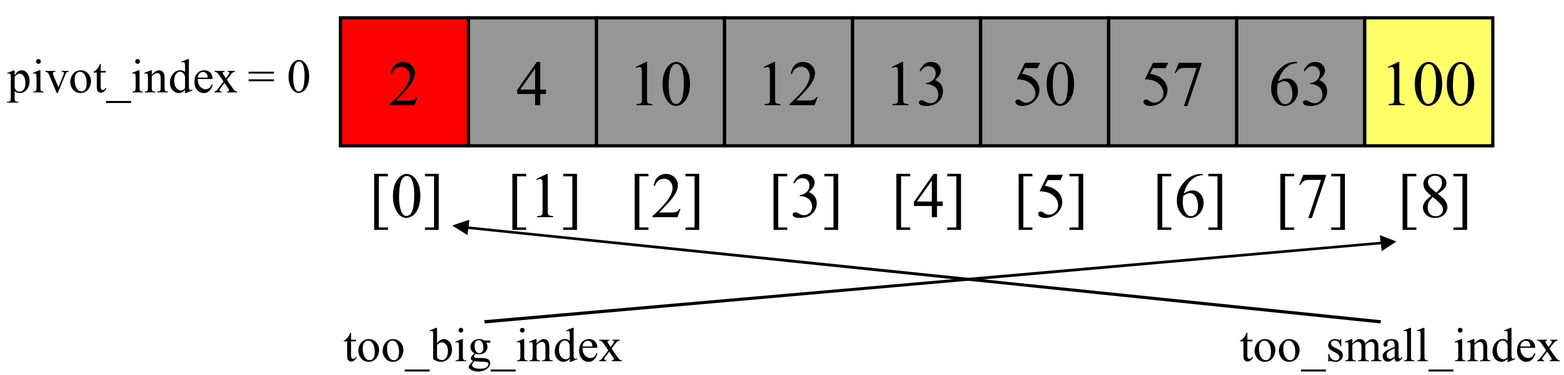
2. While data[too_small_index] > data[pivot]

   --too_small_index

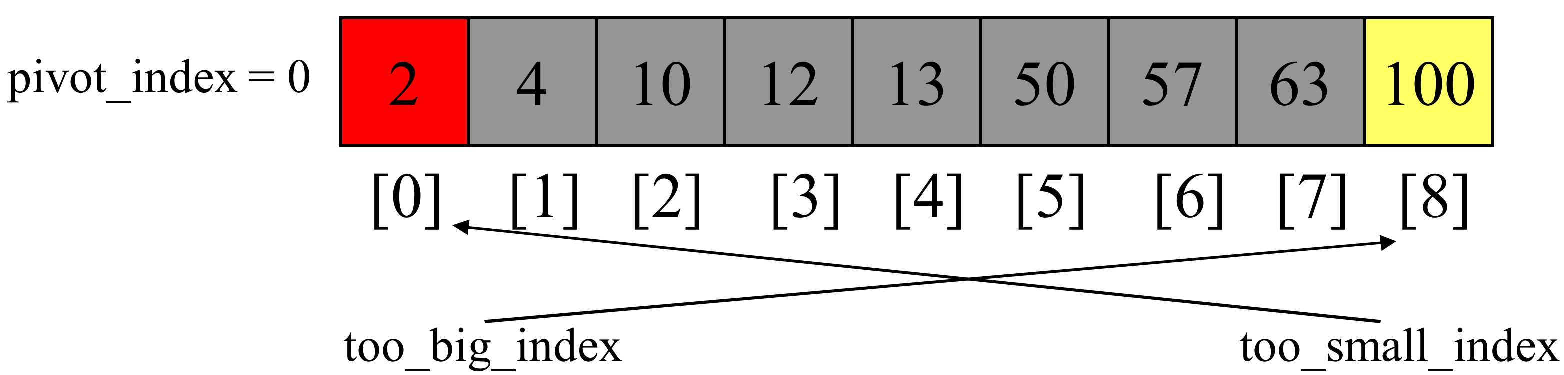
3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

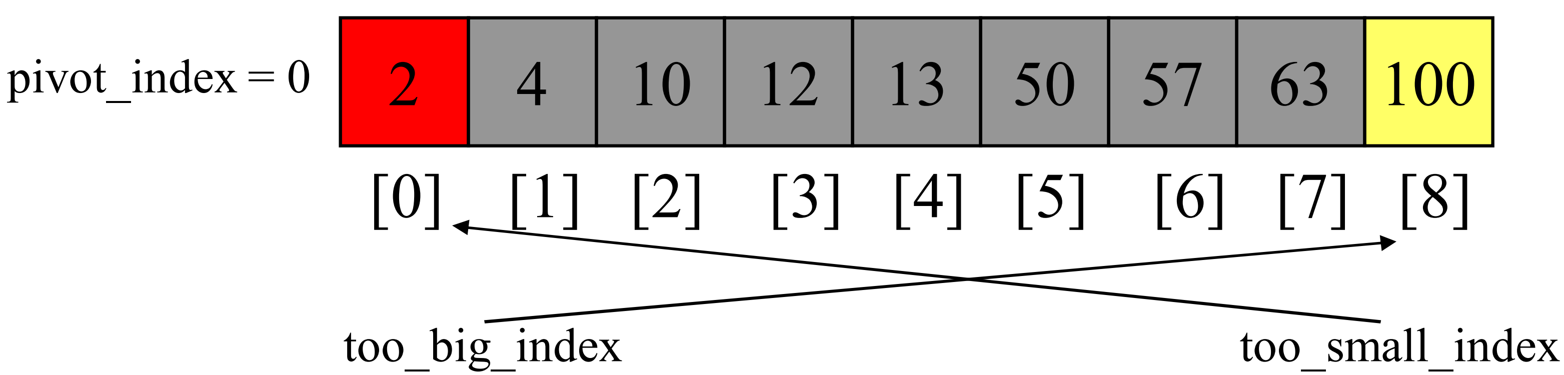
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



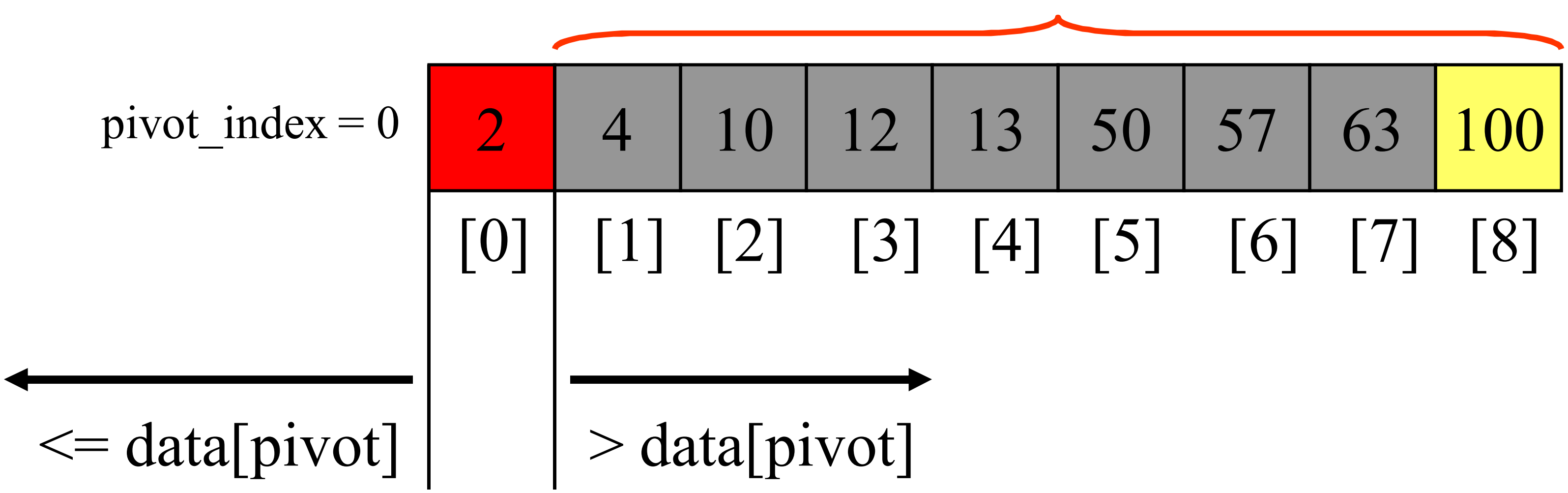
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



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.

However selection of median value takes O(n) time.

Radix Sort

Sort by keys

K^0, K^1, …, K^{r-1}

Most significant key

Least significant key

$R_0, R_1, \ldots, R_{n-1}$ are said to be sorted w.r.t. $K_0, K_1, \ldots, K_{r-1}$ iff

$$(k_i^0, k_i^1, \ldots, k_i^{r-1}) <= (k_{i+1}^0, k_{i+1}^1, \ldots, k_{i+1}^{r-1}) \quad 0 \le i < n-1$$

Most significant digit first: sort on K^0, then K^1, ...

Least significant digit first: sort on K^{r-1}, then K^{r-2}, ...

Radix Sort

$0 \leq K \leq 999$

$(K^0,\quad K^1,\quad K^2)$

MSD LSD

0-9 0-9 0-9

radix 10 sort

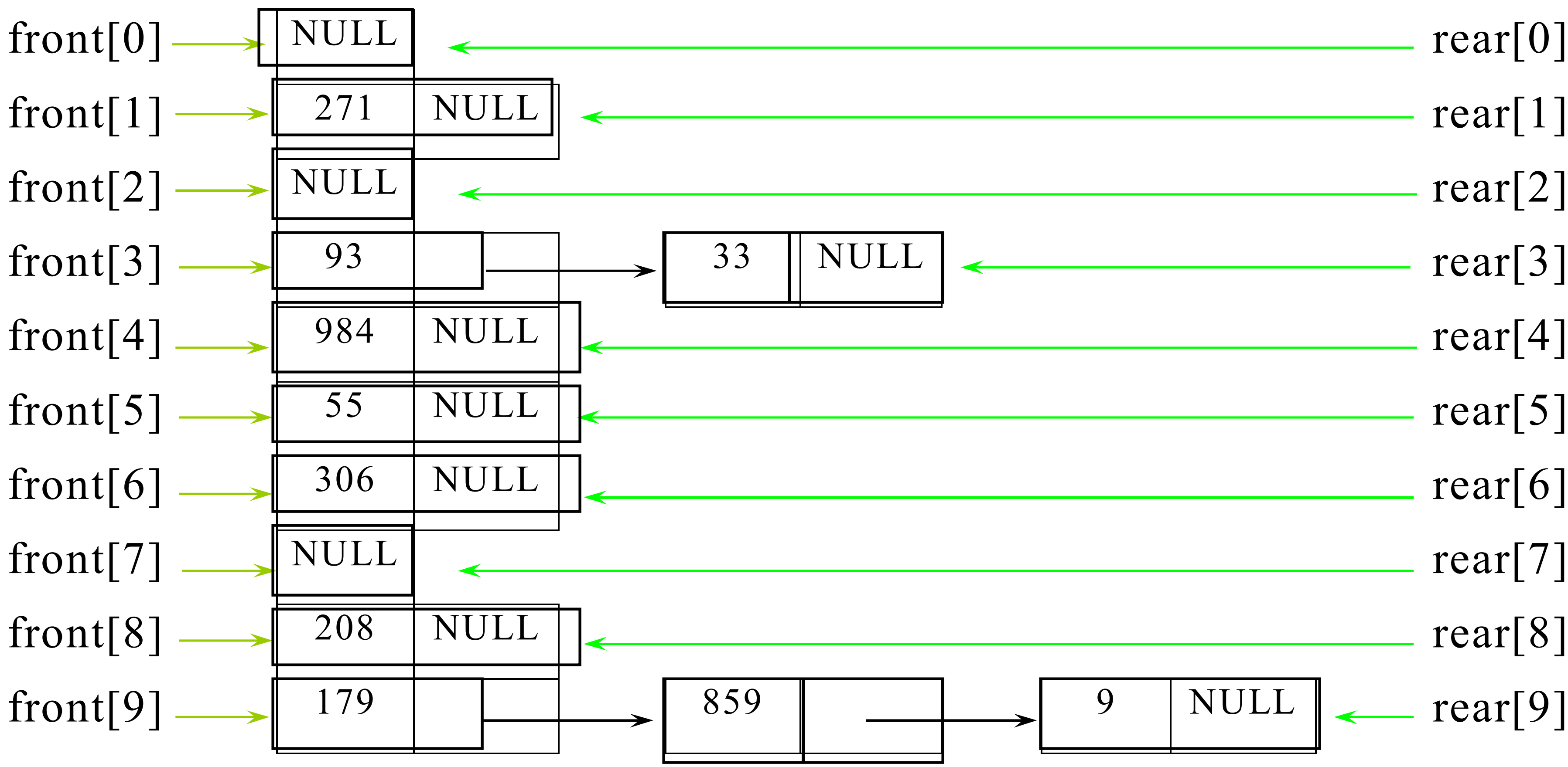
radix 2 sort

Example for LSD Radix Sort

d (digit) = 3, r (radix) = 10 ascending order

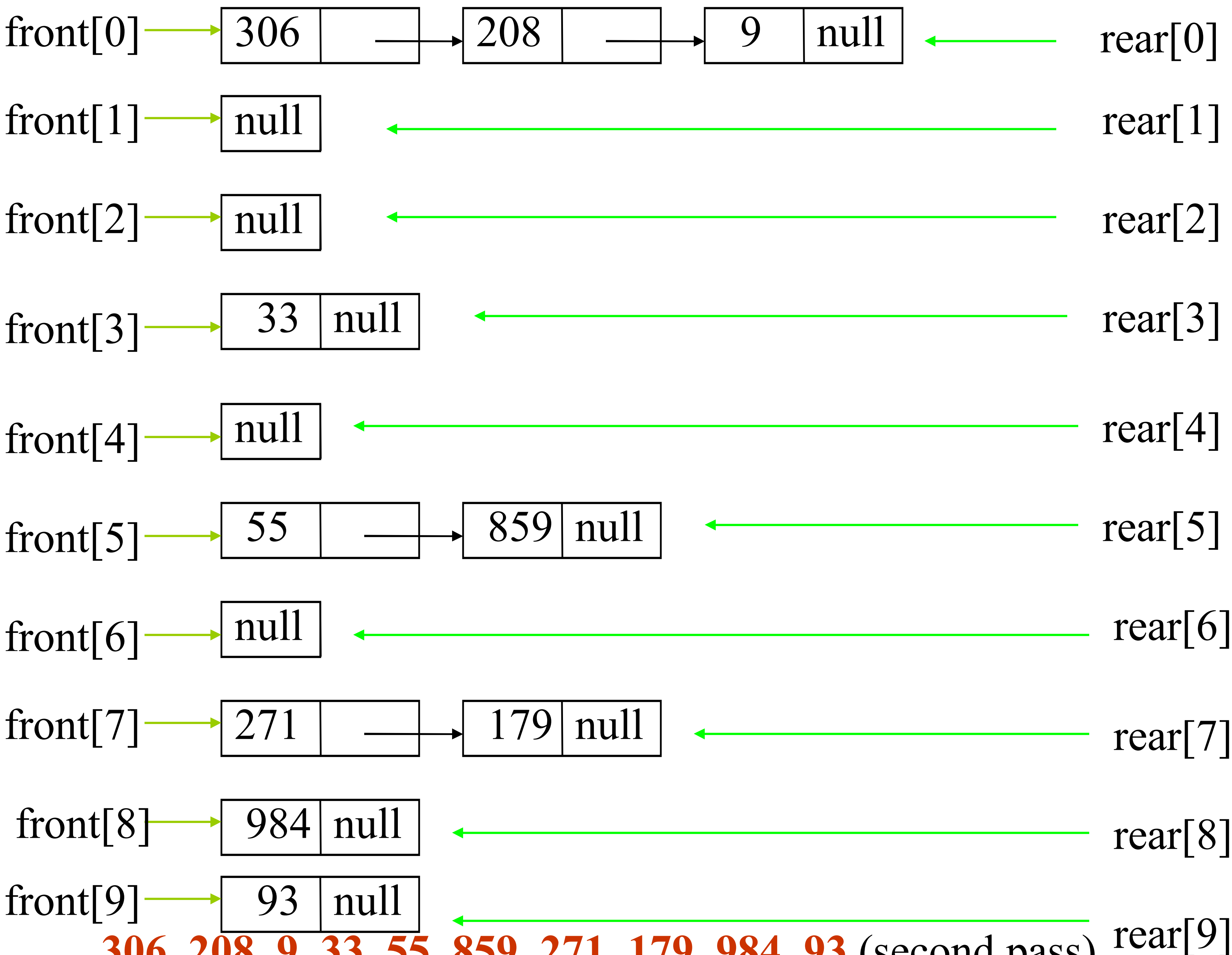
179, 208, 306, 93, 859, 984, 55, 9, 271, 33

Sort by digit

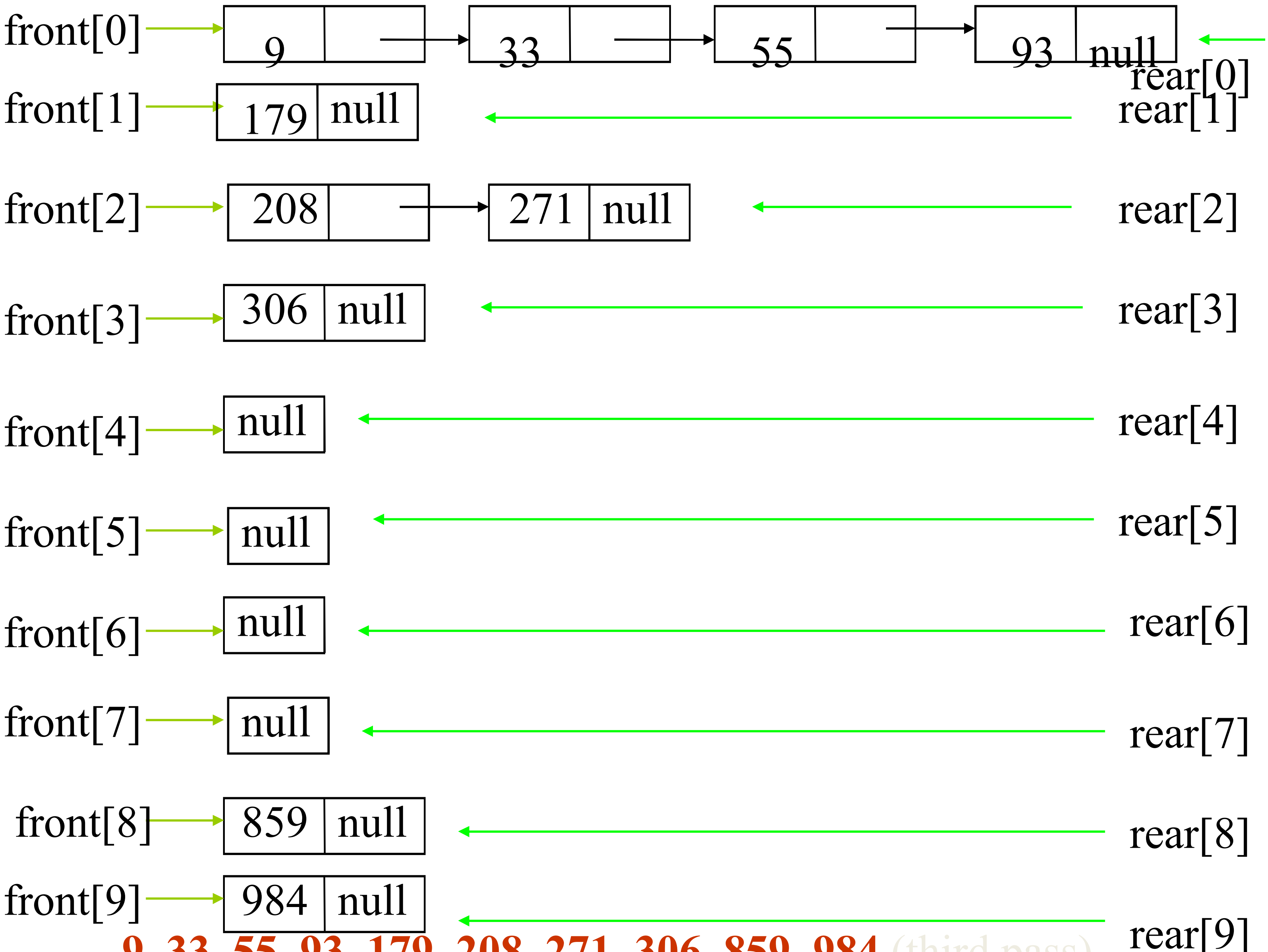


concatenate

271, 93, 33, 984, 55, 306, 208, 179, 859, 9 After the first pass



306, 208, 9, 33, 55, 859, 271, 179, 984, 93 (second pass)



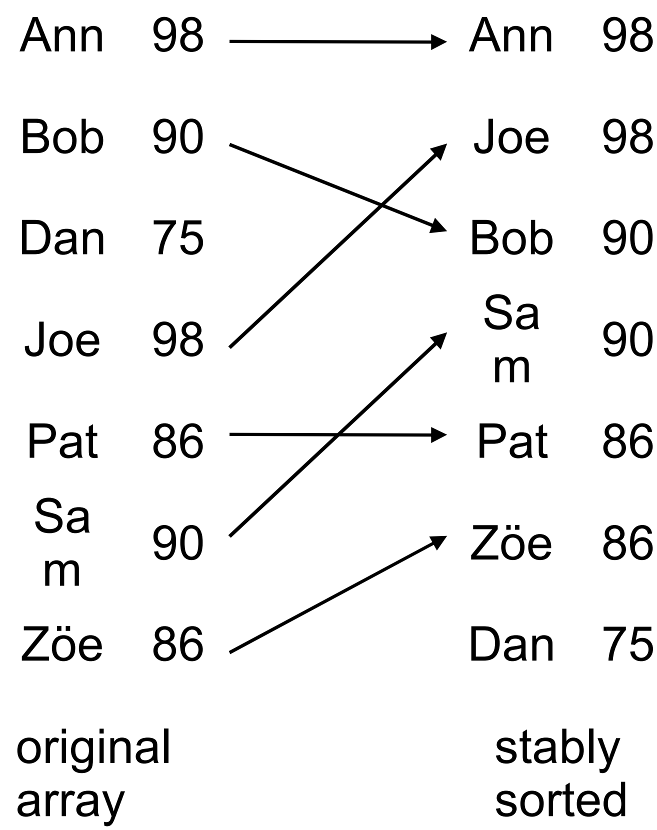
9, 33, 55, 93, 179, 208, 271, 306, 859, 984 (third pass)

Time Complexity of Radix Sort

If d is the maximum number of digits in any key and there are n keys then the worst case time complexity of Radix sort is O(dn).

Stable sort algorithms

- A stable sort keeps equal elements in the same order
- This may matter when you are sorting data according to some characteristic
- Example: sorting students by test scores



Unstable sort algorithms

- An unstable sort may or may not keep equal elements in the same order
- Stability is usually not important, but sometimes it is important

Ann	98	Joe	98
Bob	90	Ann	98
Dan	75	Bob	90
Joe	98	Sam	90
Pat	86	Zöe	86
Sam	90	Pat	86
Zöe	86	Dan	75
original array		unstably sorted	