Northeastern University

Lecture 29: Faster Sorting Methods - 2

Prof. Chen-Hsiang (Jones) Yu, Ph.D.

College of Engineering

Materials are edited by Prof. Jones Yu from

Data Structures and Abstractions with Java, 5th edition. By Frank M. Carrano and Timothy M. Henry.
ISBN-13 978-0-13-483169-5

Quick Sort

Quick Sort

- Divides an array into two pieces
  - Pieces are not necessarily halves of the array
  - Chooses one entry in the array - called the pivot
- Partitions the array

Quick Sort

- When pivot is chosen, array rearranged such that:
  - Pivot is in position that it will occupy in final sorted array
  - Entries in positions before pivot are less than or equal to pivot
  - Entries in positions after pivot are greater than or equal to pivot

Quick Sort

Algorithm quicksort(a, first, last)

// Sorts the array entries a[first..last] recursively.

```
if (first < last)
{
    Choose a pivot
    Partition the array about the pivot
    pivotIndex = index of pivot
    quickSort(a, first, pivotIndex - 1)  // Sort Smaller
    quickSort(a, pivotIndex + 1, last)  //Sort Larger
}
```

Algorithm that describes our sorting strategy

Quick Sort

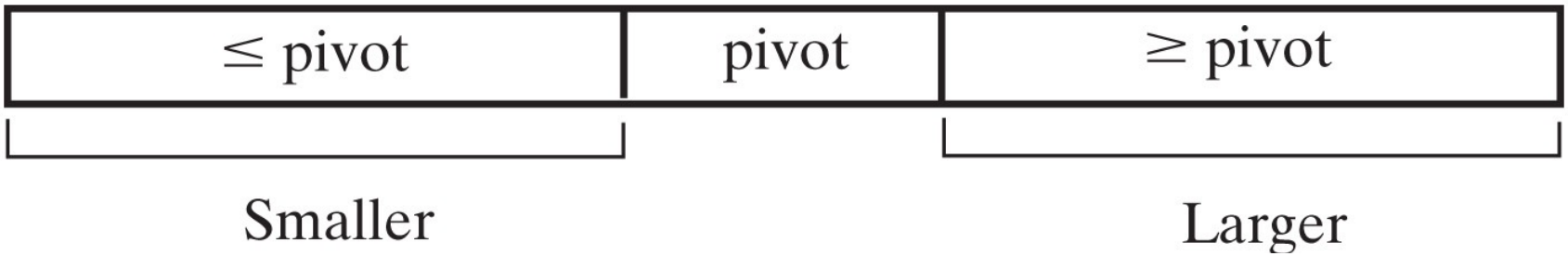


Figure 16-5: A partition of an array during a quick sort

- Quick sort: O(n log n) in average case, O(n^2) in worst case.
- Choice of pivots affects behavior

Creating the Partition

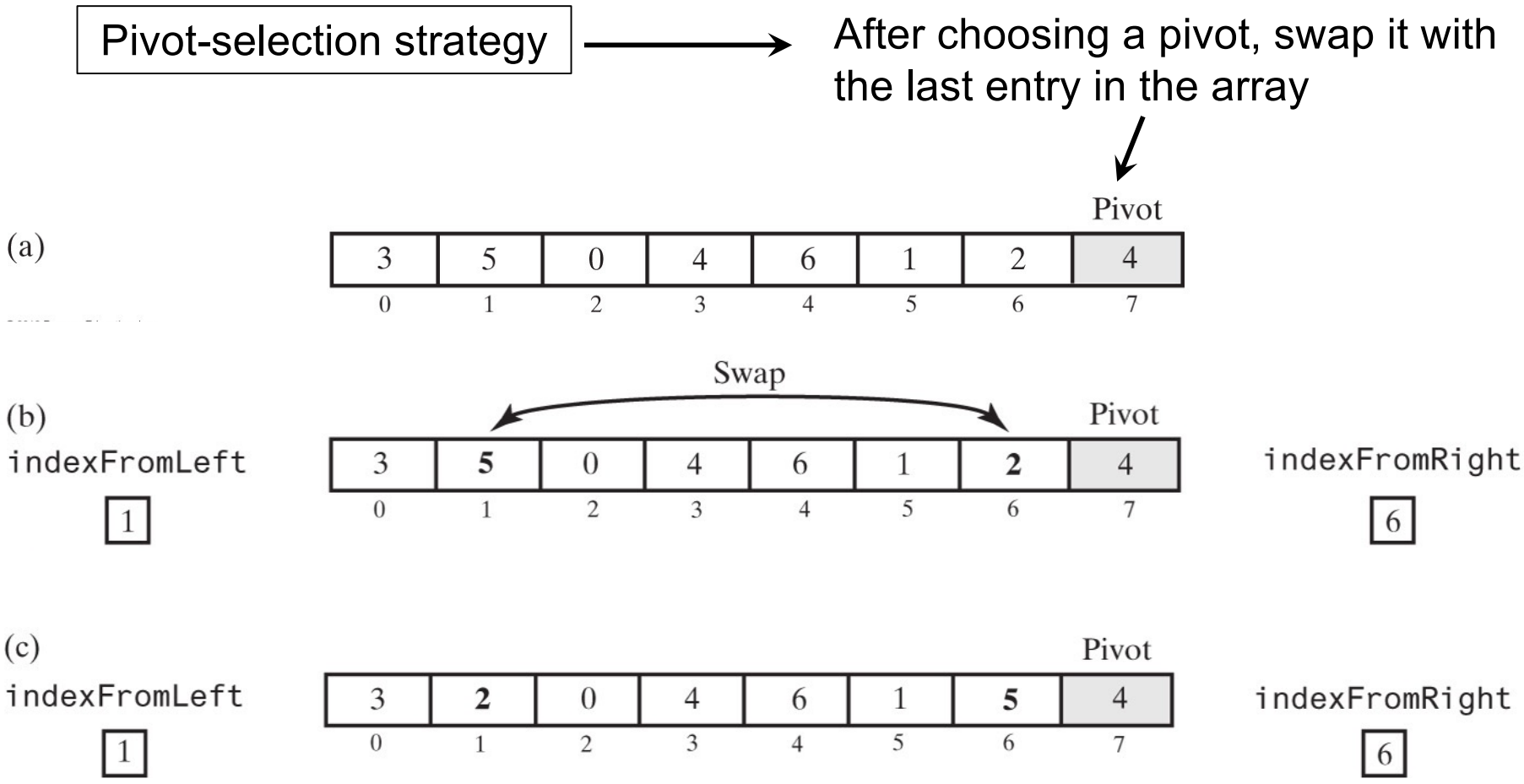


Figure 16-6: A partitioning strategy for quick sort

Creating the Partition

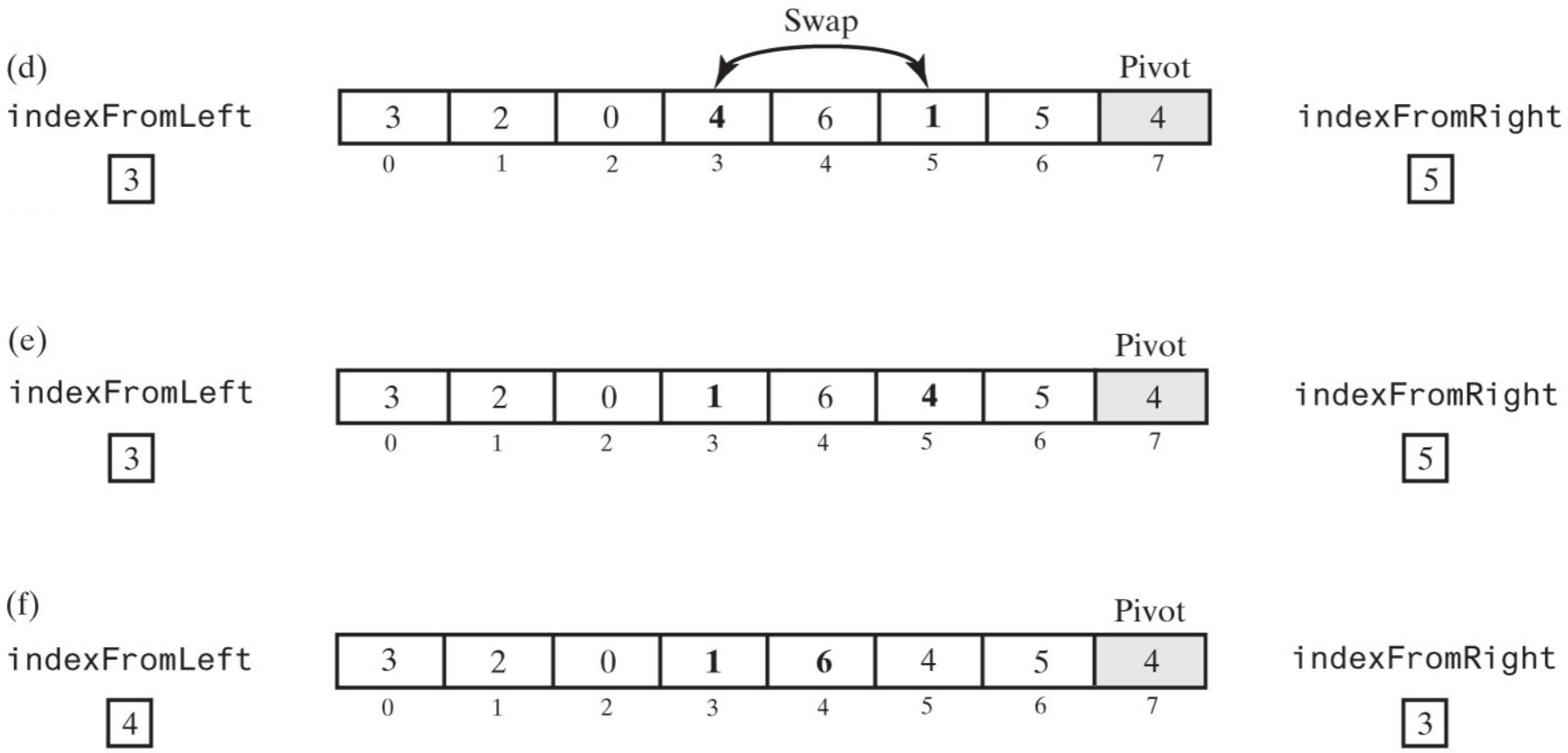


Figure 16-6: A partitioning strategy for quick sort

Creating the Partition

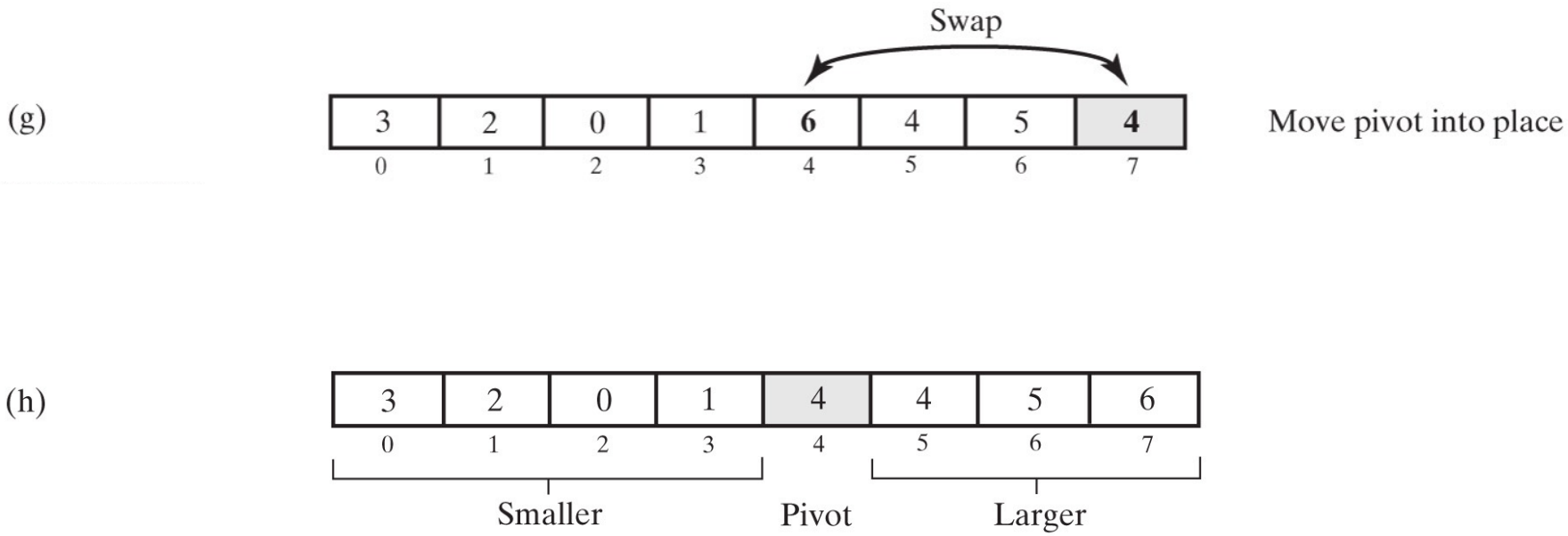


Figure 16-6: A partitioning strategy for quick sort

Pivot Selection

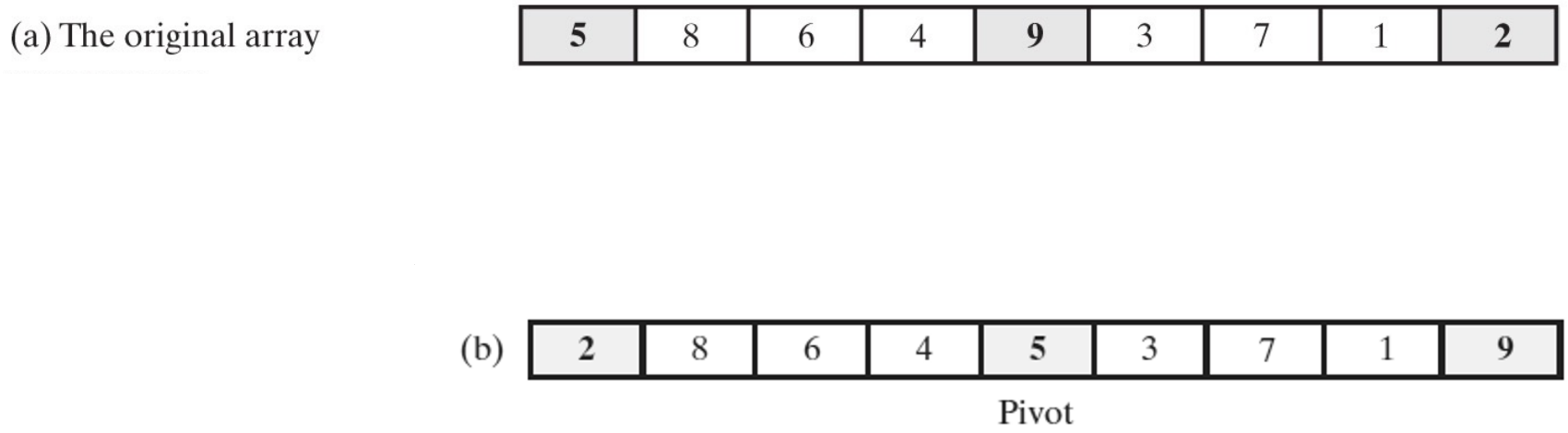
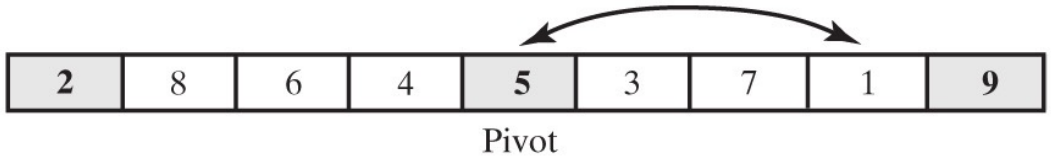


Figure 16-7: Median-of-three pivot selection: (a) The original array; (b) the array with its first, middle, and last entries sorted

Adjusting the Partition Algorithm

(a) The array after median-of-three pivot selection, as shown in Figure 16-7b



(b) The array before partitioning and just after positioning the pivot

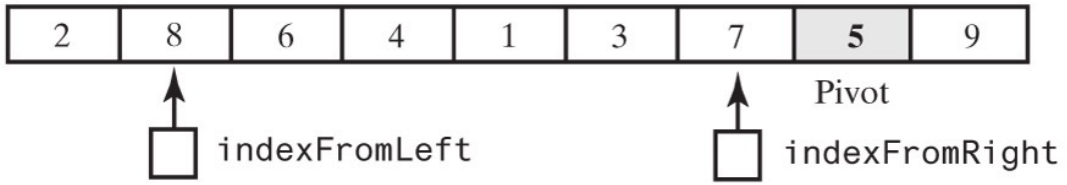


Figure 16-8: (a) The array with its first, middle, and last entries sorted; (b) the array after positioning the pivot and just before partitioning

Adjusting the Partition Algorithm

Algorithm partition(a, first, last)

// Partitions an array a[first..last] as part of quick sort into two subarrays named
// Smaller and Larger that are separated by a single entry -- the pivot -- named pivotValue.
// Entries in Smaller are <= pivotValue and appear before pivotValue in the array.
// Entries in Larger are >= pivotValue and appear after pivotValue in the array.
// Precondition: first >= 0; first < a.length; last – first >= 3; last < a.length.
// Returns the index of the pivot.

mid = index of the array's middle entry
sortFirstMiddleLast(a, first, mid, last)

// Assertion: a[mid] is the pivot, that is, pivotValue;
// a[first] <= pivotValue and a[last] >= pivotValue, so do not compare these two
// array entries with pivotValue.
// Move pivotValue to next-to-last position in array

Exchange a[mid] and a[last – 1]
pivotIndex = last – 1
pivotValue = a[pivotIndex]

```
// Determine two subarrays:
//   Smaller = a[first..endSmaller] and
//   Larger = a[endSmaller+1..last–1]
// such that entries in Smaller are <= pivotValue and entries in Larger are >= pivotValue.
// Initially, these subarrays are empty.

indexFromLeft = first + 1
indexFromRight = last − 2

done = false

while (!done)
{
      // Starting at the beginning of the array, leave entries that are < pivotValue and
      // locate the first entry that is >= pivotValue. You will find one, since the last
      // entry is >= pivotValue.
      while (a[indexFromLeft]  <  pivotValue)
            indexFromLeft++

      // Starting at the end of the array, leave entries that are > pivotValue and
      // locate the first entry that is <= pivotValue. You will find one, since the first
      // entry is <= pivotValue.
      while (a[indexFromRight]  >  pivotValue)
            indexFromRight−−
```

```
      // Assertion: a[indexFromLeft] >= pivotValue and a[indexFromRight] <= pivotValue

      if (indexFromLeft < indexFromRight)
      {
          Exchange a[indexFromLeft] and a[indexFromRight]
          indexFromLeft++
          indexFromRight−−
      } else {
          done = true
       }

}  // end of while

// Place pivotValue between the subarrays Smaller and Larger
Exchange a[pivotIndex] and a[indexFromLeft]
pivotIndex = indexFromLeft

// Assertion: Smaller = a[first..pivotIndex–1], pivotValue = a[pivotIndex], Larger = a[pivotIndex+1..last]

return pivotIndex
```

The Quick Sort Method

```
public static <T extends Comparable<? super T>>
       void quickSort(T[] a, int first, int last)
{
   if (last - first + 1 < MIN_SIZE)
   {
      insertionSort(a, first, last);
   }
   else
   {
      // Create the partition: Smaller | Pivot | Larger
      int pivotIndex = partition(a, first, last);

      // Sort subarrays Smaller and Larger
      quickSort(a, first, pivotIndex - 1);
      quickSort(a, pivotIndex + 1, last);
   } // end if
} // end quickSort
```

Above method implements quick sort

Quick Sort in the Java Class Library

```
public static void sort(type[] a)

public static void sort(type[] a, int first, int after)
```

Class **`Arrays`** in the package **`java.util`** uses a quick sort to sort arrays of primitive types into ascending order

Exercise

- Trace the steps that the method quickSort takes when sorting the following array into ascending order. Assume that MIN_SIZE = 4

9 6 2 4 8 7 5 3

Answer

```
quickSort(array, 0, 7)
partition(array, 0, 7)

 9 6 2 4 8 7 5 3
 3 6 2 4 8 7 5 9
 3 6 2 5 8 7 4 9
 3 2 6 5 8 7 4 9
 3 2 4 5 8 7 6 9

quickSort(array, 0, 1)
insertionSort(array, 0, 1)

 2 3 4 5 8 7 6 9

quickSort(array, 3, 7)
partition(array, 3, 7)

 2 3 4 5 8 7 6 9
 2 3 4 5 8 6 7 9
 2 3 4 5 6 8 7 9
 2 3 4 5 6 7 8 9

quickSort(array, 3, 4)
insertionSort(array, 3, 4)

 2 3 4 5 6 7 8 9

quickSort(array, 6, 7)
insertionSort(array, 6, 7)

 2 3 4 5 6 7 8 9
```

Radix Sort

Radix Sort

- Does not use comparison
- Treats array entries as if they were strings that have the same length.
  - Group integers according to their rightmost character (digit) into “buckets”
  - Repeat with next character (digit), etc.
- It is not suitable as a general-purpose sorting algorithm.

Radix Sort

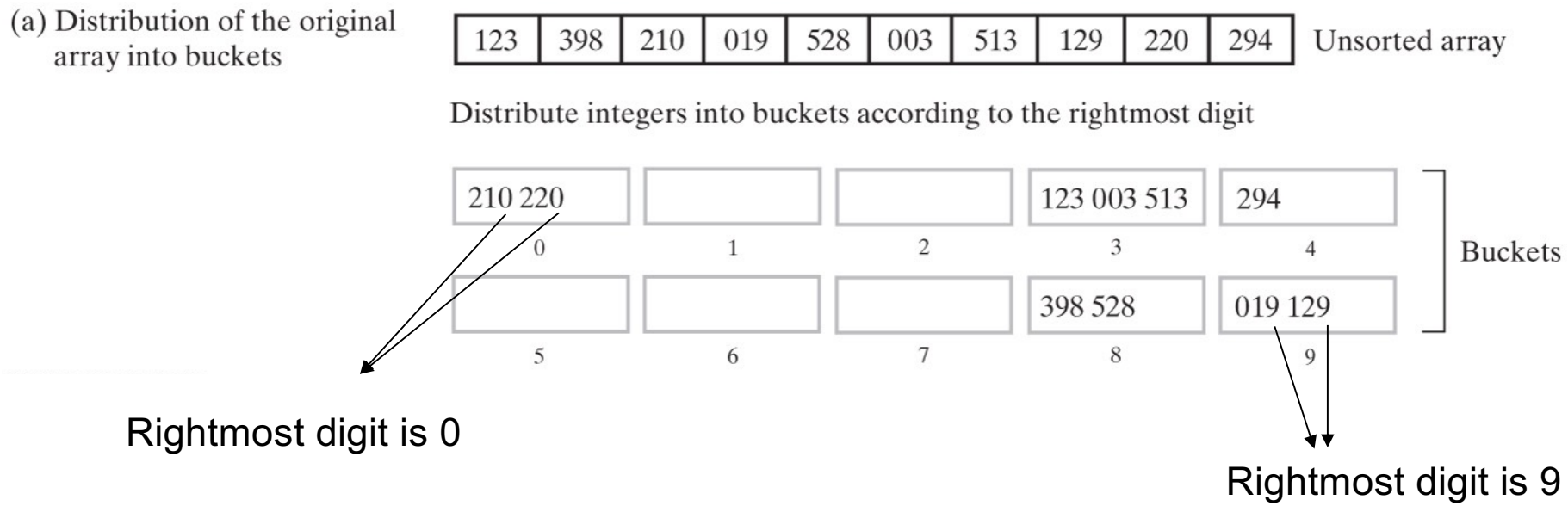


Figure 16-9: Radix sort: (a) Original array and buckets after first distribution;

Radix Sort

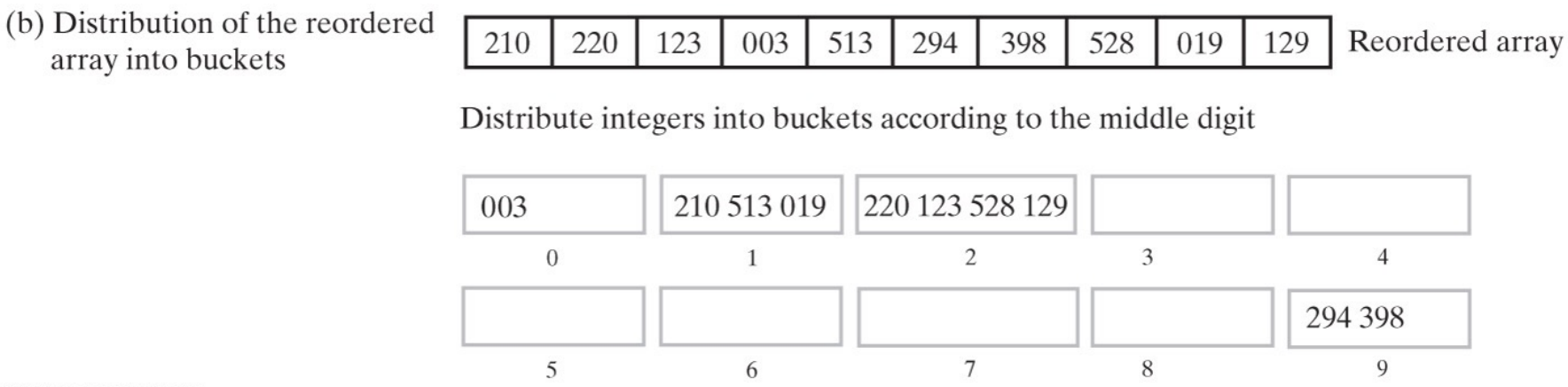


Figure 16-9: Radix sort: (b) reordered array and buckets after second distribution

Radix Sort

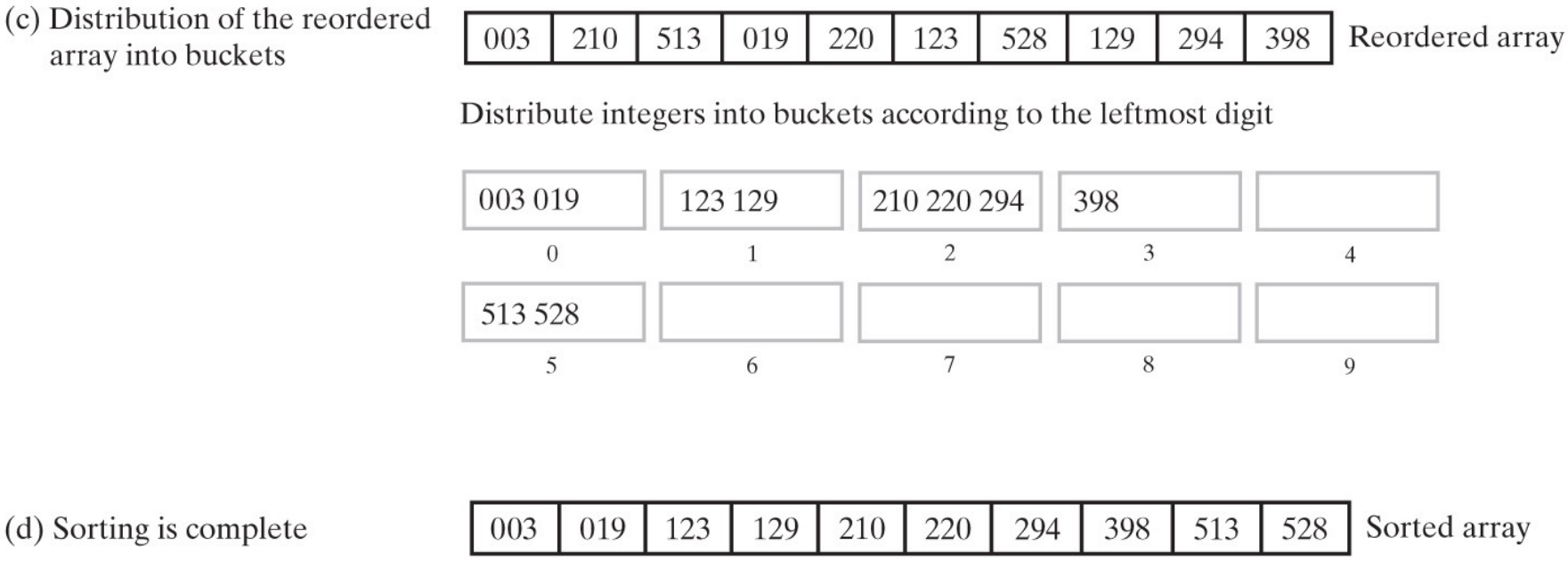


Figure 16-9: Radix sort: (c) reordered array and buckets after third distribution; (d) sorted array

Pseudocode for Radix Sort

```
Algorithm radixSort(a, first, last, maxDigits)
// Sorts the array of positive decimal integers a[first..last] into ascending order;
// maxDigits is the number of digits in the longest integer.

for (i = 0 to maxDigits - 1)
{
    Clear bucket[0], bucket[1], . . . , bucket[9]
    for (index = first to last)
    {
        digit = digit i of a[index]
        Place a[index] at end of bucket[digit]
    }
    Place contents of bucket[0], bucket[1], . . . , bucket[9] into the array a
}
```

Radix sort is an O(n) algorithm for certain data, it is not appropriate for all data

Comparing the Algorithms

	Best Case	Average Case	Worst Case
Radix Sort	$O(n)$	$O(n)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Shell Sort	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Figure 16-10: The time efficiency of various sorting algorithms, expressed in Big Oh notation

Exercise

- You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B.
- Write a method to merge B into A in sorted order.

Answer

```
public static void mergeArrays(int[] a, int[] b, int numA, int numB) {

      int newLastIndex = numA + numB - 1;
      int lastIndexA = numA - 1;
      int lastIndexB = numB - 1;

      int loop = numA >= numB? numB : numA;

      for(int i = loop; i >= 0; i--) {
            if(a[lastIndexA] > b[lastIndexB]) {
                  a[newLastIndex] = a[lastIndexA];
                  lastIndexA--;
            } else {
                  a[newLastIndex] = b[lastIndexB];
                  lastIndexB--;
            }
            newLastIndex--;
      }

      //Copy remaining elements of a or b to array a
      if(loop == numA) {
            for(int j = lastIndexB; j >= 0; j--) {
                  a[newLastIndex] = b[j];
                  newLastIndex--;
            }
      }else {
            for(int j = lastIndexA; j >= 0; j--) {
                  a[newLastIndex] = a[j];
                  newLastIndex--;
            }
      }
}
```