Faster Sorting Methods

Chapter 9

THIRD EDITION
Data Structures
and Abstractions
with Java™
FRANK M. CARRANO

Contents

- Merge Sort
    - Merging Arrays
    - Recursive Merge Sort
    - The Efficiency of Merge Sort
    - Iterative Merge Sort
    - Merge Sort in the Java Class Library

Contents

Objectives

- Sort array into ascending order using
  - merge sort
  - quick sort
  - radix sort
- Assess efficiency of a sort and discuss relative efficiencies of various methods

Merge Sort

- Divide array into two halves
  - Sort the two halves
  - Merge them into one sorted array
- Uses strategy of "divide and conquer"
  - Divide problem up into two or more distinct, smaller tasks
- Good application for recursion

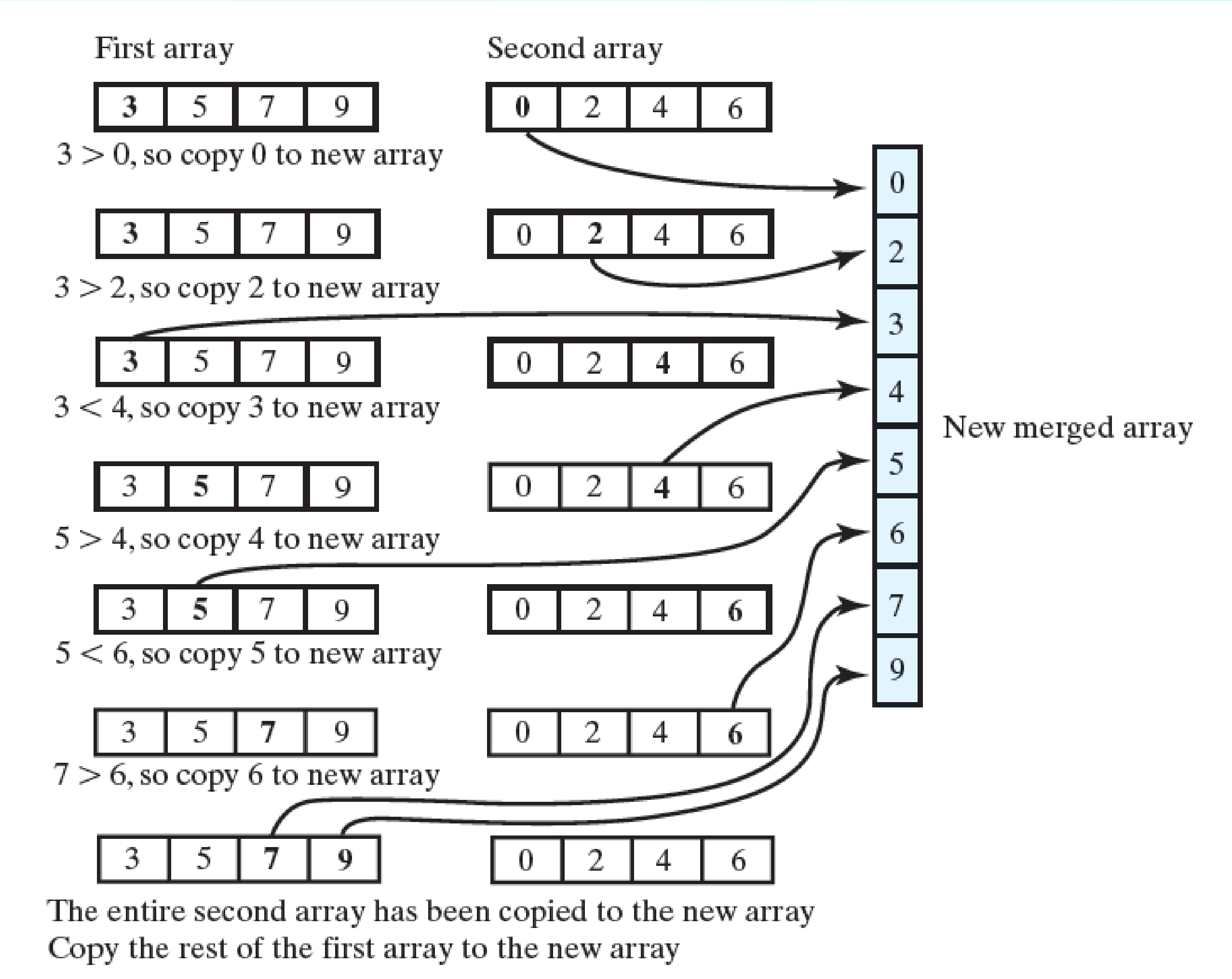
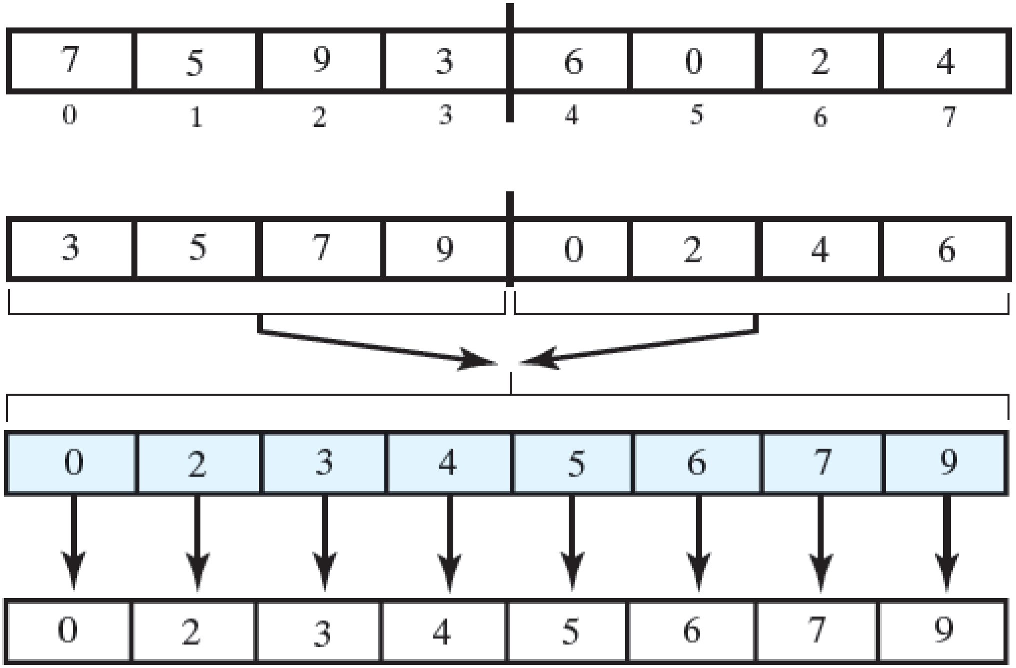


Figure 9-1 Merging two sorted arrays into one sorted array



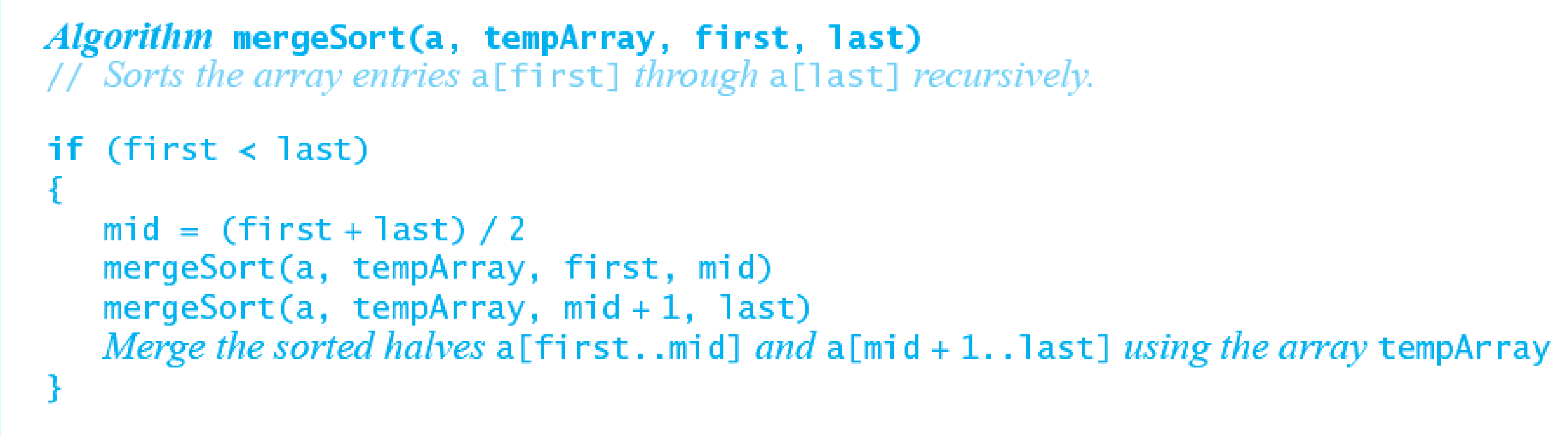
Divide the array into two halves

Sort the two halves

Merge the sorted halves into another array

Copy the merged array back into the original array

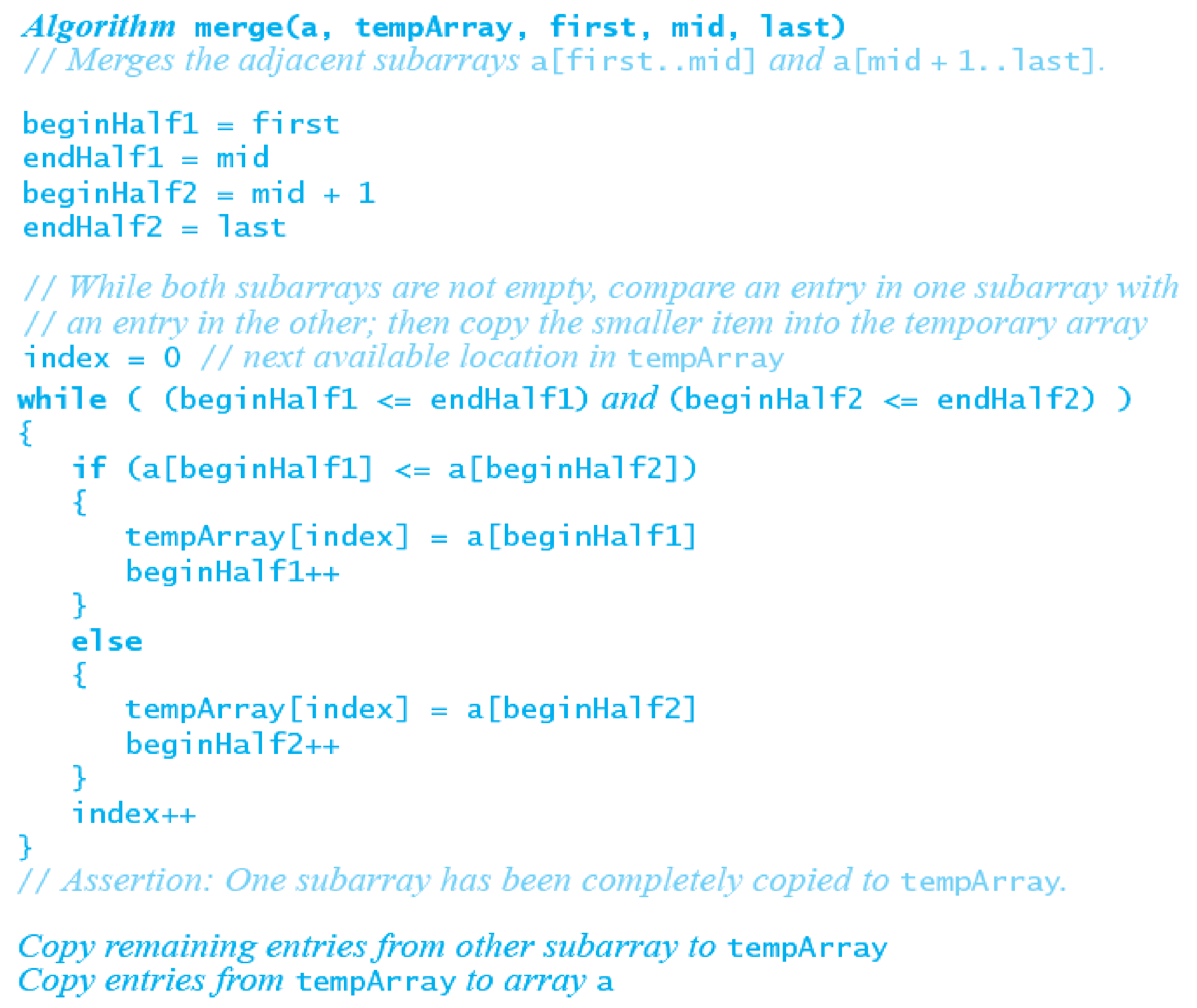
Figure 9-2 The major steps in a merge sort



```
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
   mid = (first + last) / 2
   mergeSort(a, tempArray, first, mid)
   mergeSort(a, tempArray, mid + 1, last)
   Merge the sorted halves a[first..mid] and a[mid + 1..last] using the array tempArray
}
```

Merge Sort Algorithm

Algorithm to Merge



```
Algorithm merge(a, tempArray, first, mid, last)
// Merges the adjacent subarrays a[first..mid] and a[mid + 1..last].

beginHalf1 = first
endHalf1 = mid
beginHalf2 = mid + 1
endHalf2 = last

// While both subarrays are not empty, compare an entry in one subarray with
// an entry in the other; then copy the smaller item into the temporary array
index = 0 // next available location in tempArray
while ( (beginHalf1 <= endHalf1) and (beginHalf2 <= endHalf2) )
{
   if (a[beginHalf1] <= a[beginHalf2])
   {
      tempArray[index] = a[beginHalf1]
      beginHalf1++
   }
   else
   {
      tempArray[index] = a[beginHalf2]
      beginHalf2++
   }
   index++
}
// Assertion: One subarray has been completely copied to tempArray.

Copy remaining entries from other subarray to tempArray
Copy entries from tempArray to array a
```

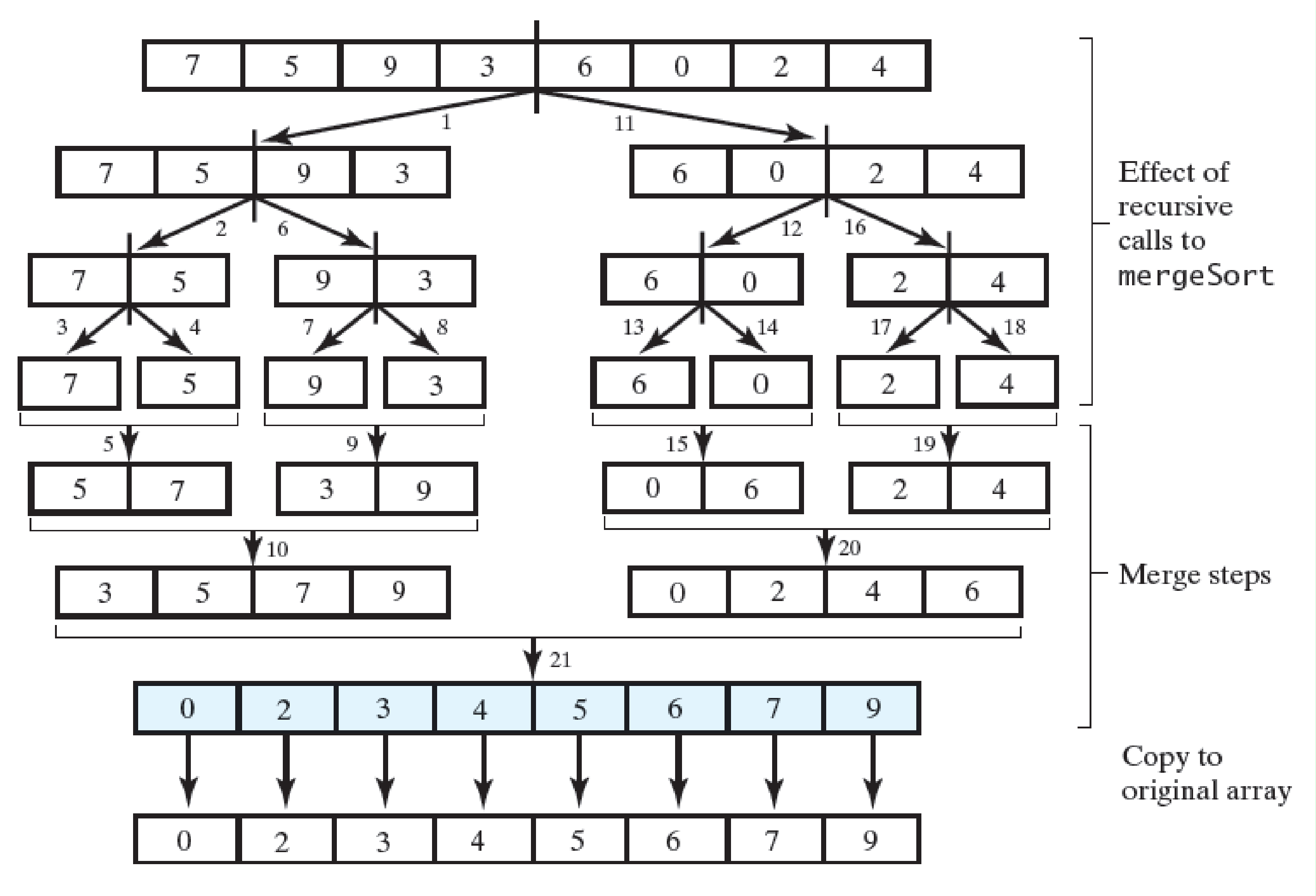
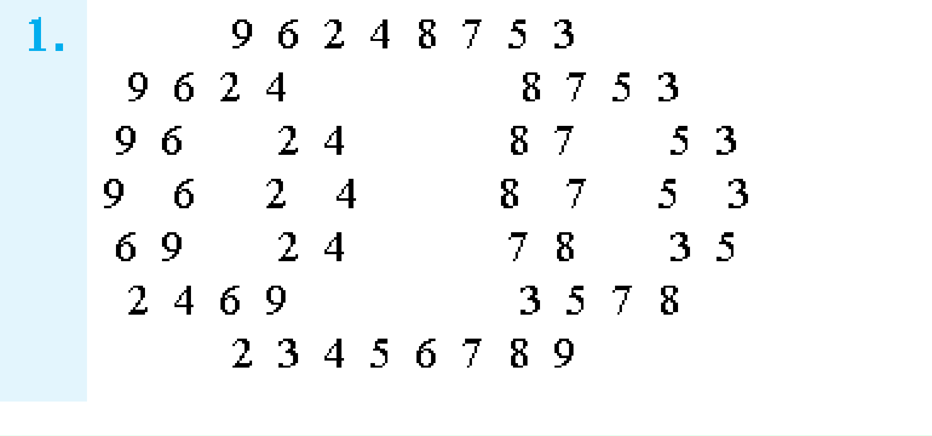


Figure 9-3 The effect of the recursive calls and the merges during a merge sort

Question 1 Trace the steps that a merge sort takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3.



1.

9 6 2 4 8 7 5 3

9 6 2 4 8 7 5 3

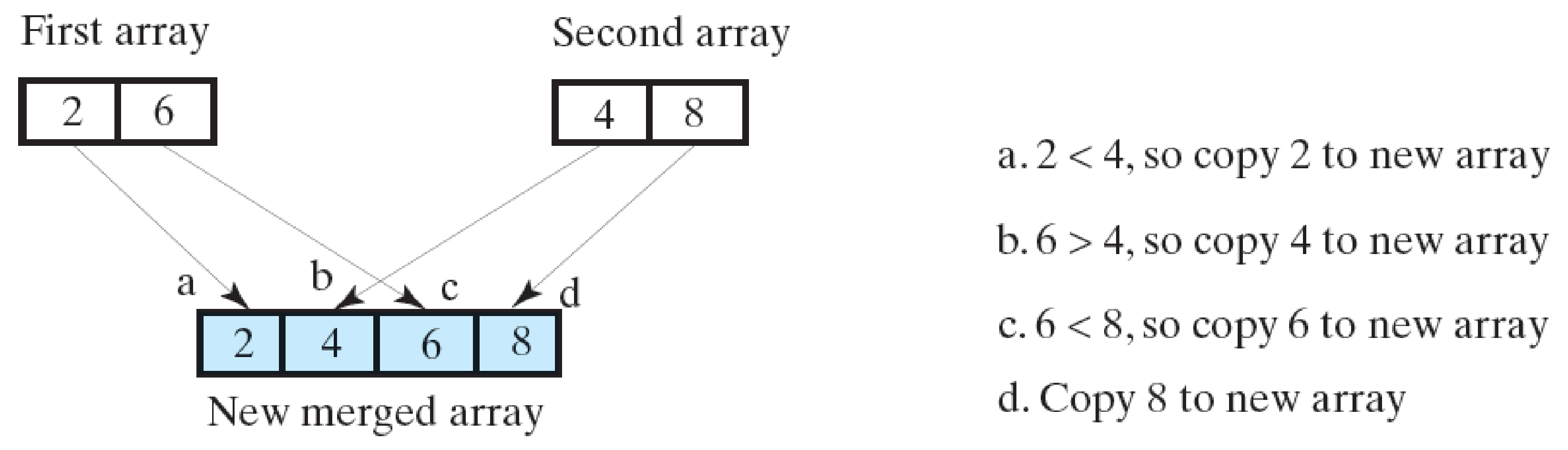
9 6 2 4 8 7 5 3

9 6 2 4 8 7 5 3

6 9 2 4 7 8 3 5

2 4 6 9 3 5 7 8

2 3 4 5 6 7 8 9



a. $2 < 4$, so copy 2 to new array

b. $6 > 4$, so copy 4 to new array

c. $6 < 8$, so copy 6 to new array

d. Copy 8 to new array

Figure 9-4 A worst-case merge of two sorted arrays

Efficiency of Merge Sort

- For $n = 2^k$ entries
    - In general k levels of recursive calls are made
- Each merge requires at most $3n - 1$ comparisons
- Calls to merge do at most $3n - 2^2$ operations
- Can be shown that efficiency is O(n log n)

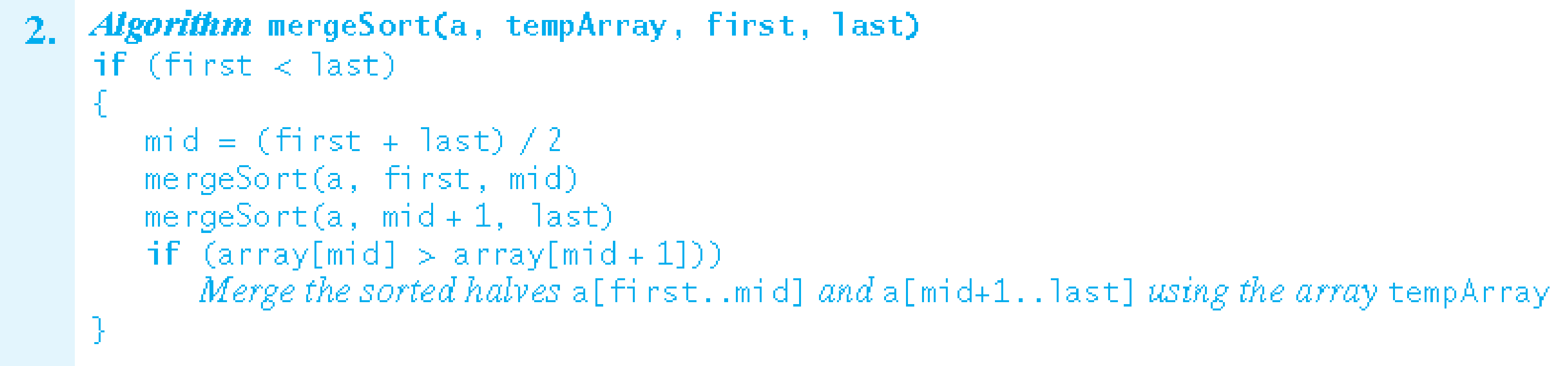
Iterative Merge Sort

- More difficult than recursive version
  - Recursion controls merging process
  - Iteration would require separate control
- Iterative more efficient in time, space required
  - More difficult to code correctly

Merge Sort in the Java Class Library

- Class **`Arrays`** in **`java.util`** has sort methods
  - **`public static void sort(Object[] a)`**
  - **`public static void sort (Object[] a, int first, int after)`**
- These methods use merge sort
  - Merge step skipped if none of entries in left half, greater than entries in right half

Question 2 Modify the merge sort algorithm given in Segment 9.3 so that it skips any unnecessary merges, as just described.

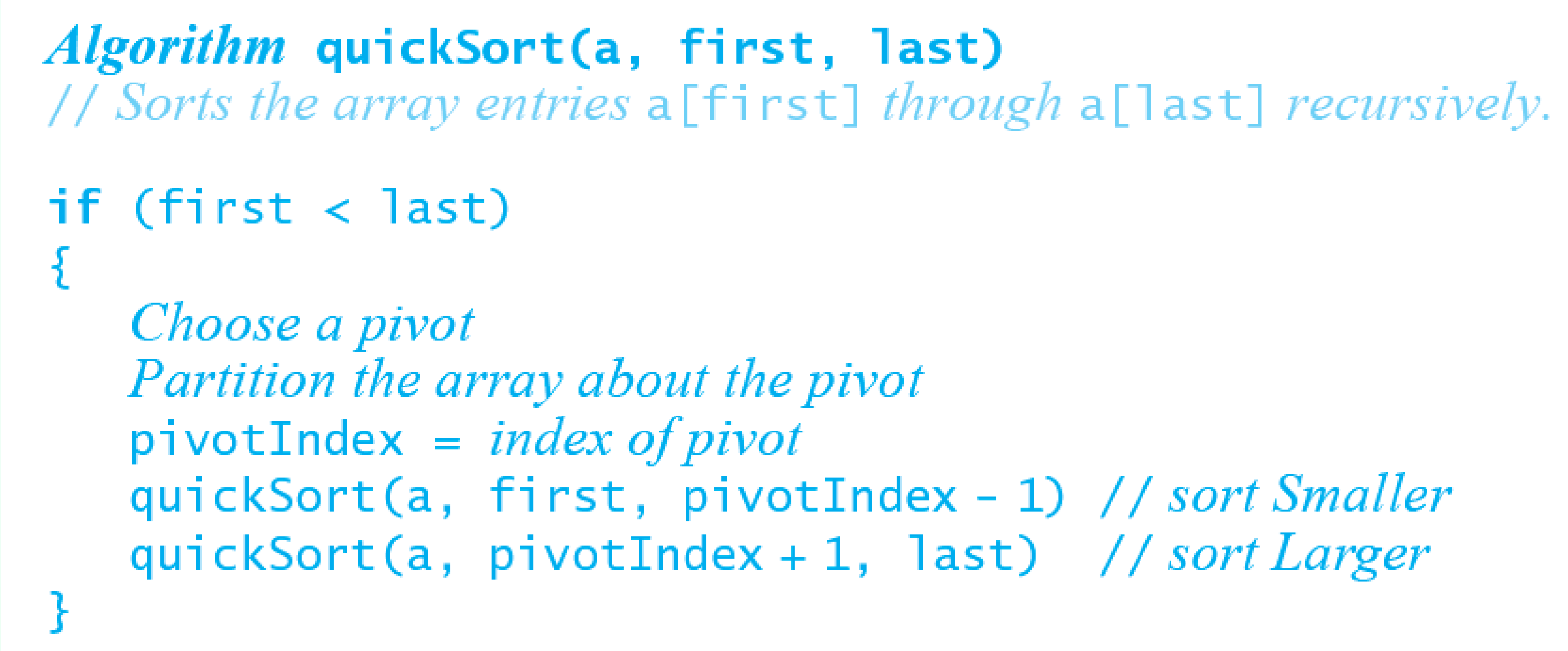


```
2. Algorithm mergeSort(a, tempArray, first, last)
   if (first < last)
   {
      mid = (first + last) / 2
      mergeSort(a, first, mid)
      mergeSort(a, mid + 1, last)
      if (array[mid] > array[mid + 1]))
         Merge the sorted halves a[first..mid] and a[mid+1..last] using the array tempArray
   }
```

Quick Sort

- Like merge sort, divides arrays into two portions
  - Unlike merge sort, portions not necessarily halves of the array
- One entry called the “pivot”
  - Pivot in position that it will occupy in final sorted array
  - Entries in positions before pivot less than or equal to the pivot
  - Entries in positions after pivot are greater than or equal to the pivot

Algorithm



```
Algorithm quickSort(a, first, last)
// Sorts the array entries a[first] through a[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
}
```

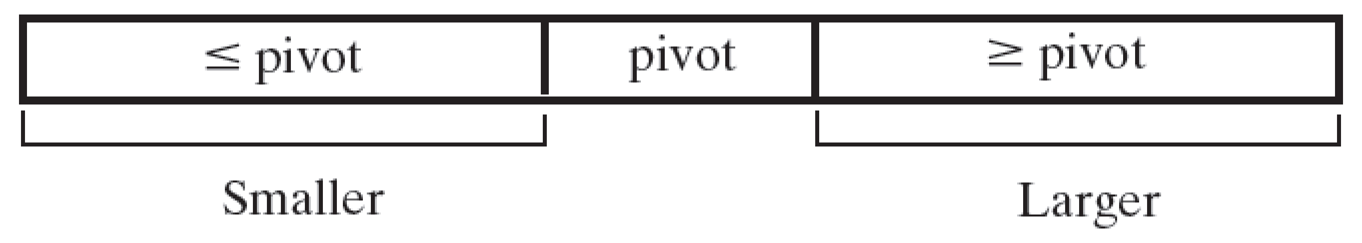


Figure 9-5 A partition of an array during a quick sort

Efficiency of Quick Sort

- For *n* items
    - *n* comparisons to find pivot
- If every choice of pivot cause equal sized arrays, recursive calls halve the array
- Results in O(n log n)

- This we conclude *before* we develop strategy

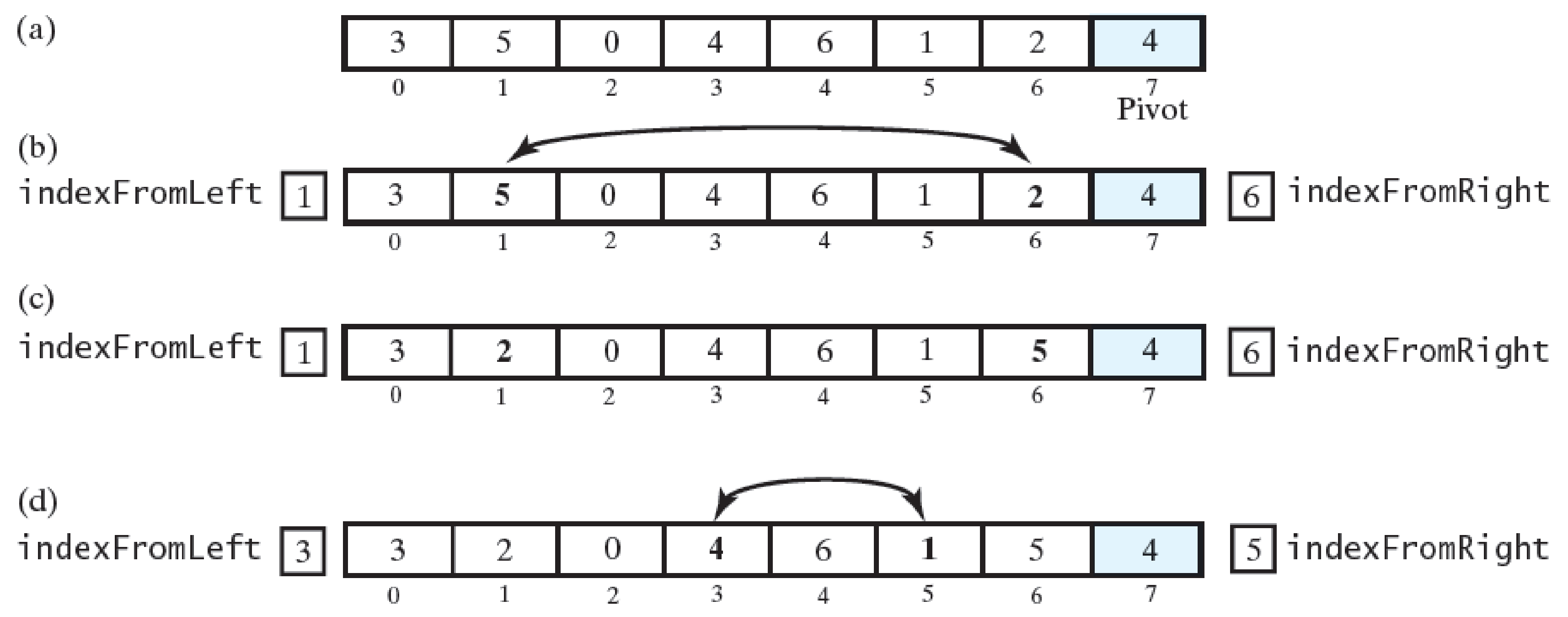


Figure 9-6 A partitioning strategy for quick sort

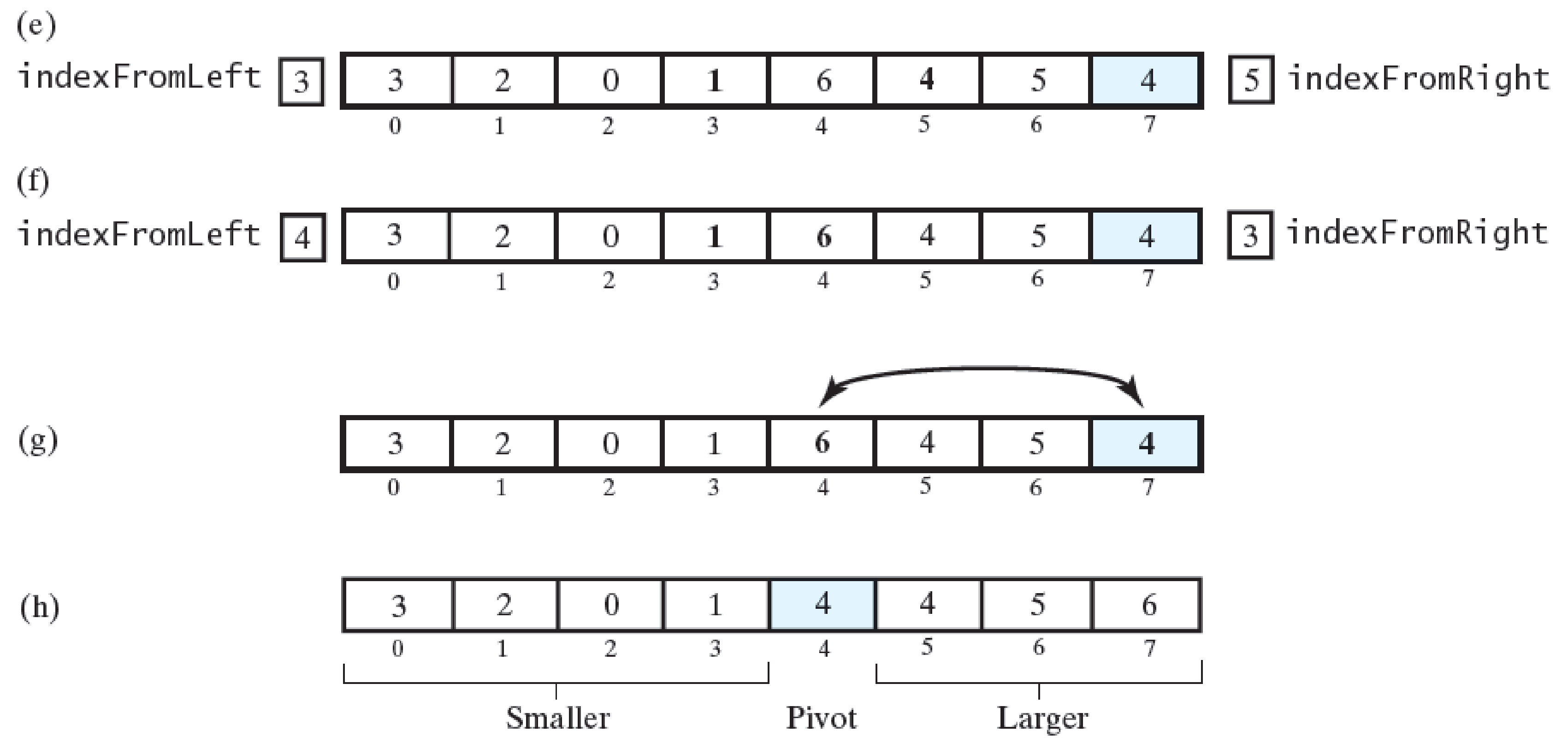
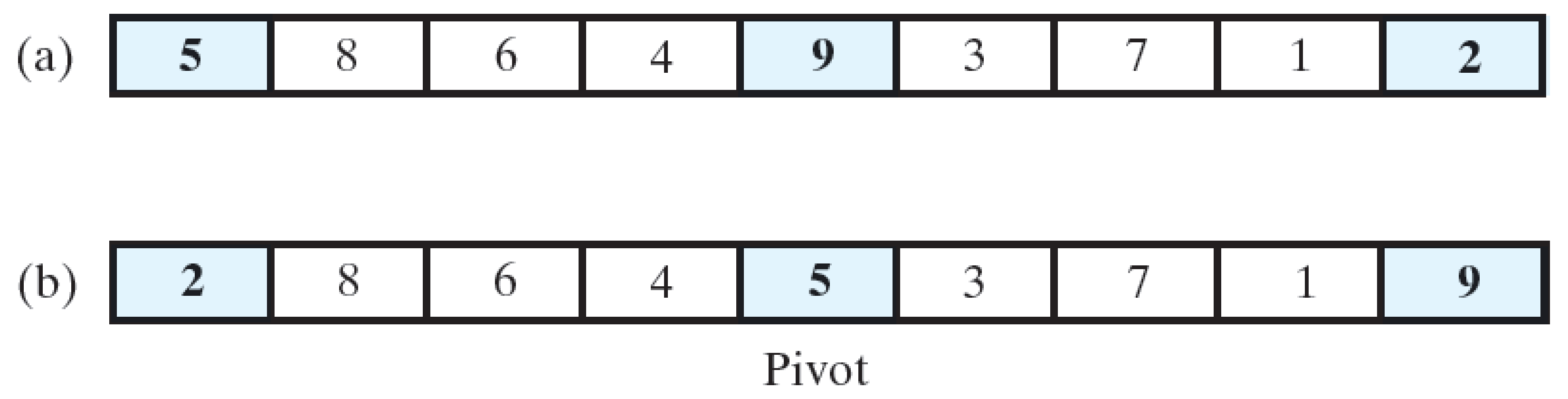


Figure 9-6 A partitioning strategy for quick sort



(a)	**5**	8	6	4	**9**	3	7	1	**2**
(b)	**2**	8	6	4	**5**	3	7	1	**9**

Figure 9-7 Median-of-three pivot selection: (a) The original array; (b) the array with its first, middle, and last entries sorted

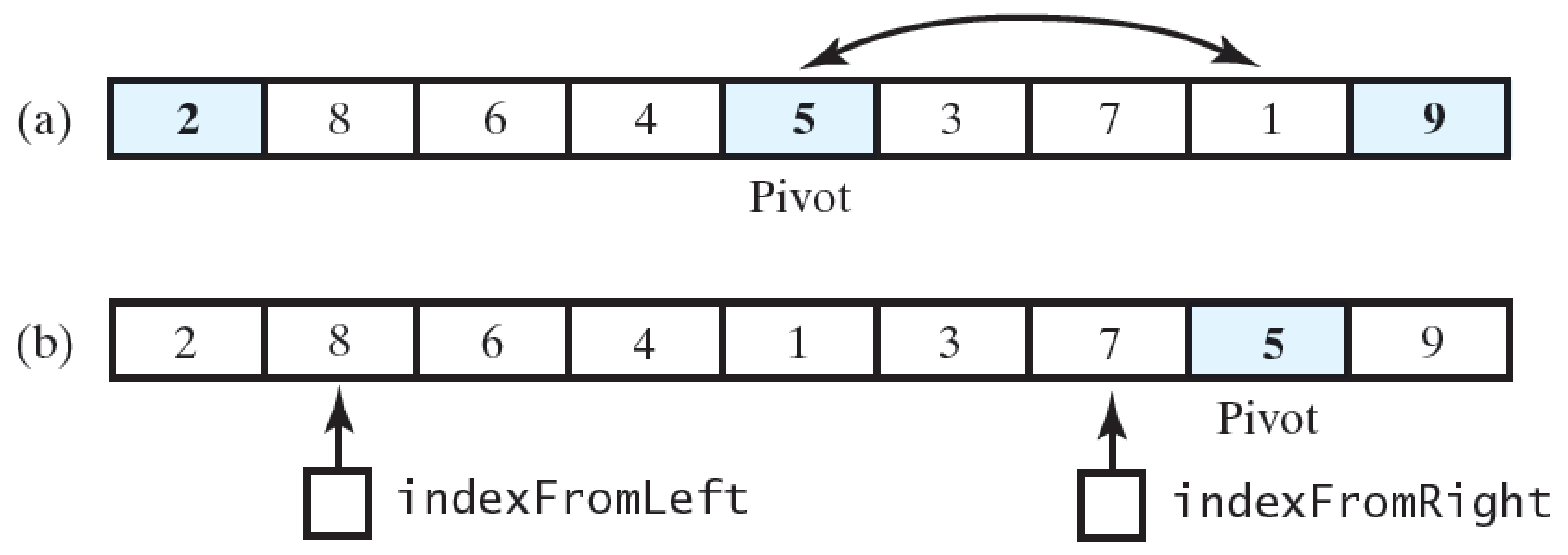
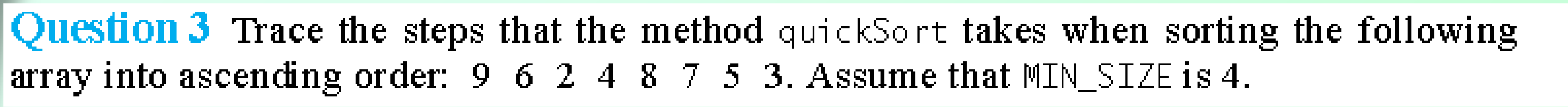


Figure 9-8 (a) The array with its first, middle, and last entries sorted; (b) the array after positioning the pivot and just before partitioning

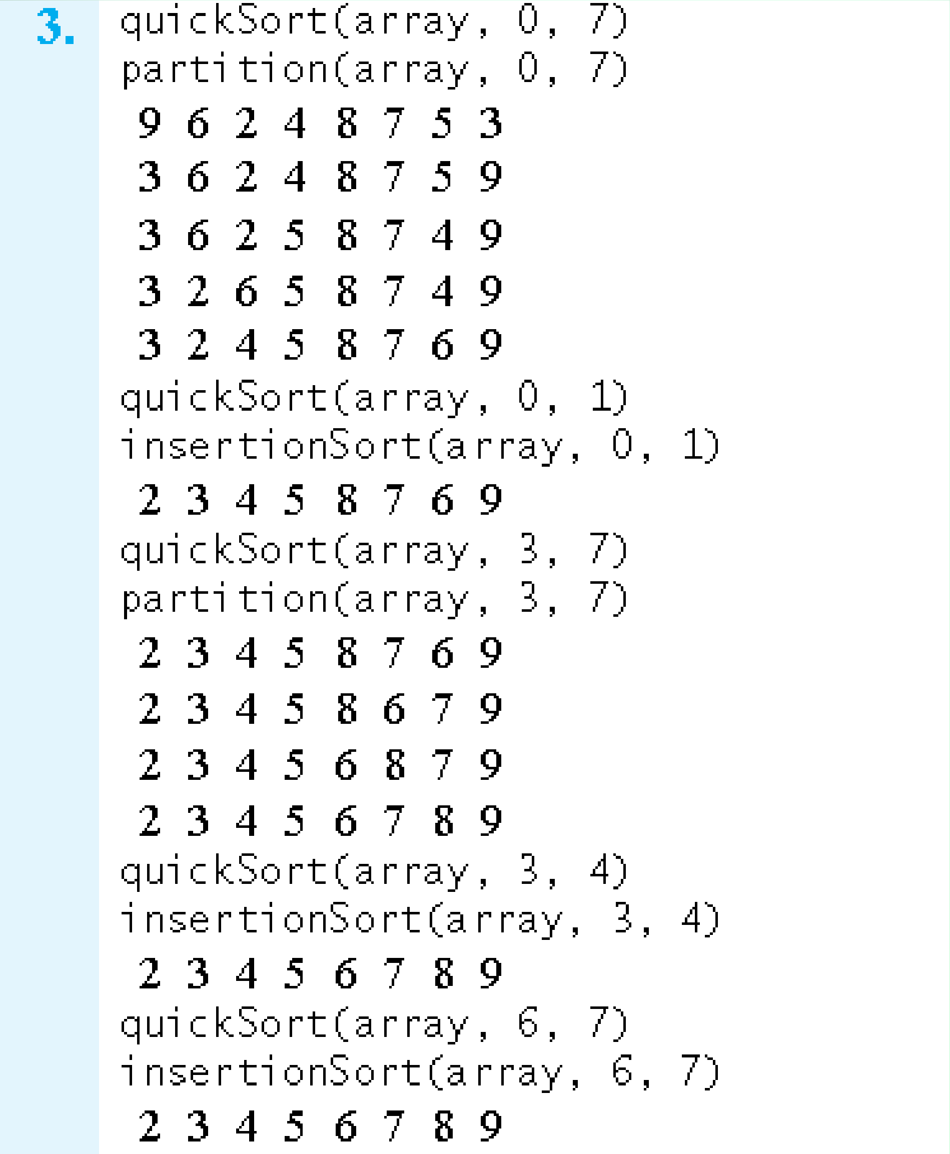
Java Code for Quick Sort

- Pivot selection code, Listing 9-B
- Partitioning code, Listing 9-C
- QuickSort code, Listing 9-D
- Java Class Library – Class **`Arrays`** uses quick sort for p
  - **`public stati`** **`e[] a)`**
  - **`public static void sort (type[] a, int first, int after)`**

Note: Code listing files must be in same folder as PowerPoint files for links to work



Question 3 Trace the steps that the method `quickSort` takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3. Assume that `MIN_SIZE` is 4.

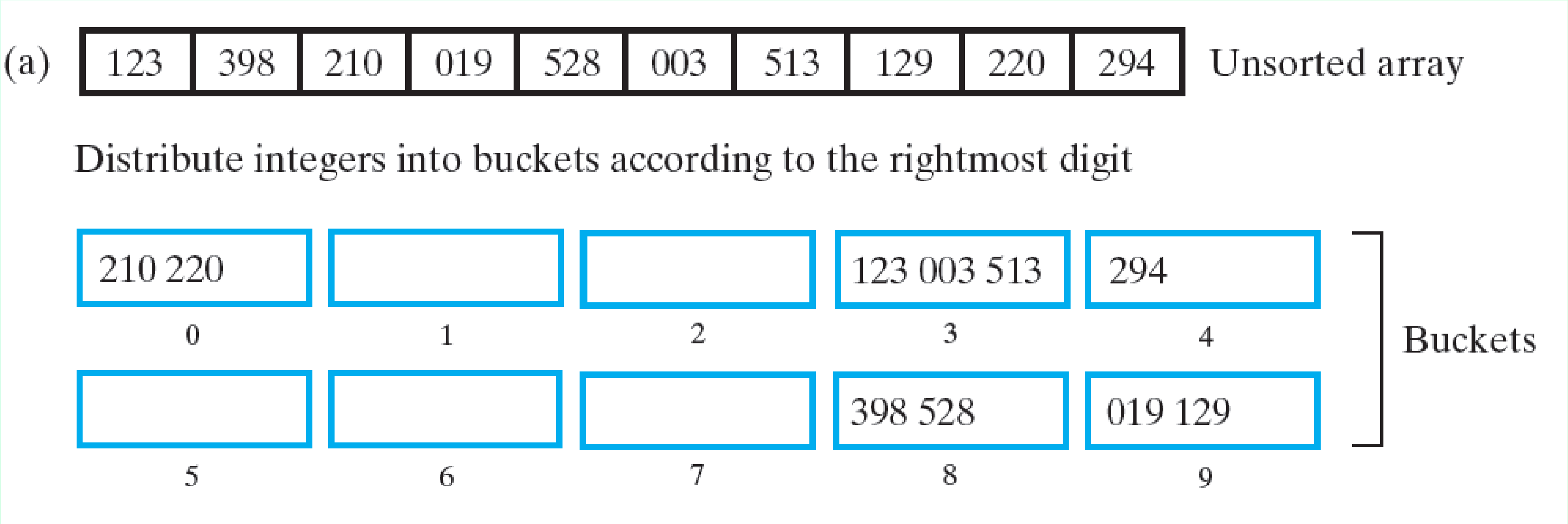


3.

```
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

- Previously seen sorts on objects that can be compared
- Radix sort does not use comparison
  - Looks for matches in certain categories
  - Places items in “buckets”
- Origin is from punched card sorting machines



(a)

123	398	210	019	528	003	513	129	220	294

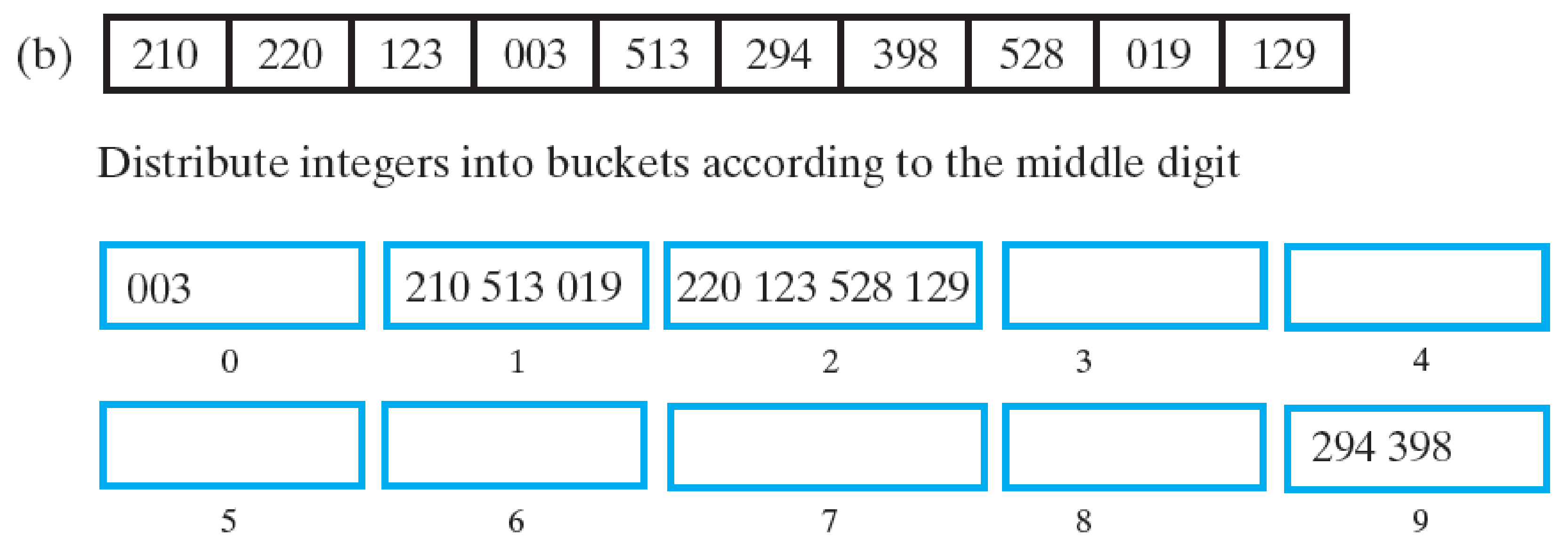
Unsorted array

Distribute integers into buckets according to the rightmost digit

210 220			123 003 513	294
0	1	2	3	4
			398 528	019 129
5	6	7	8	9

Buckets

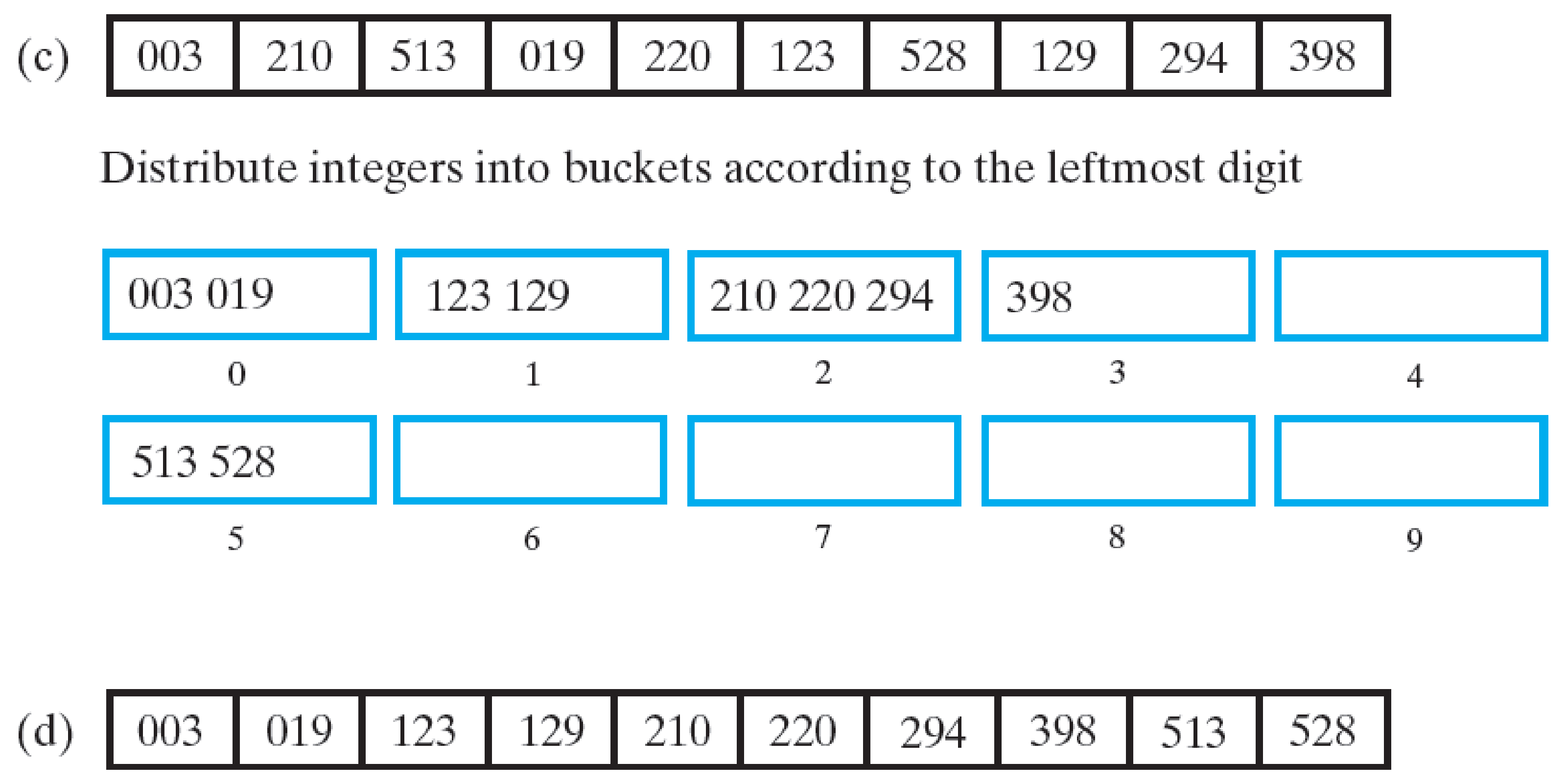
Figure 9-9 Radix sort: (a) Original array and buckets after first distribution;



Distribute integers into buckets according to the middle digit

0	1	2	3	4	5	6	7	8	9
003	210 513 019	220 123 528 129							294 398

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



(c)

003	210	513	019	220	123	528	129	294	398

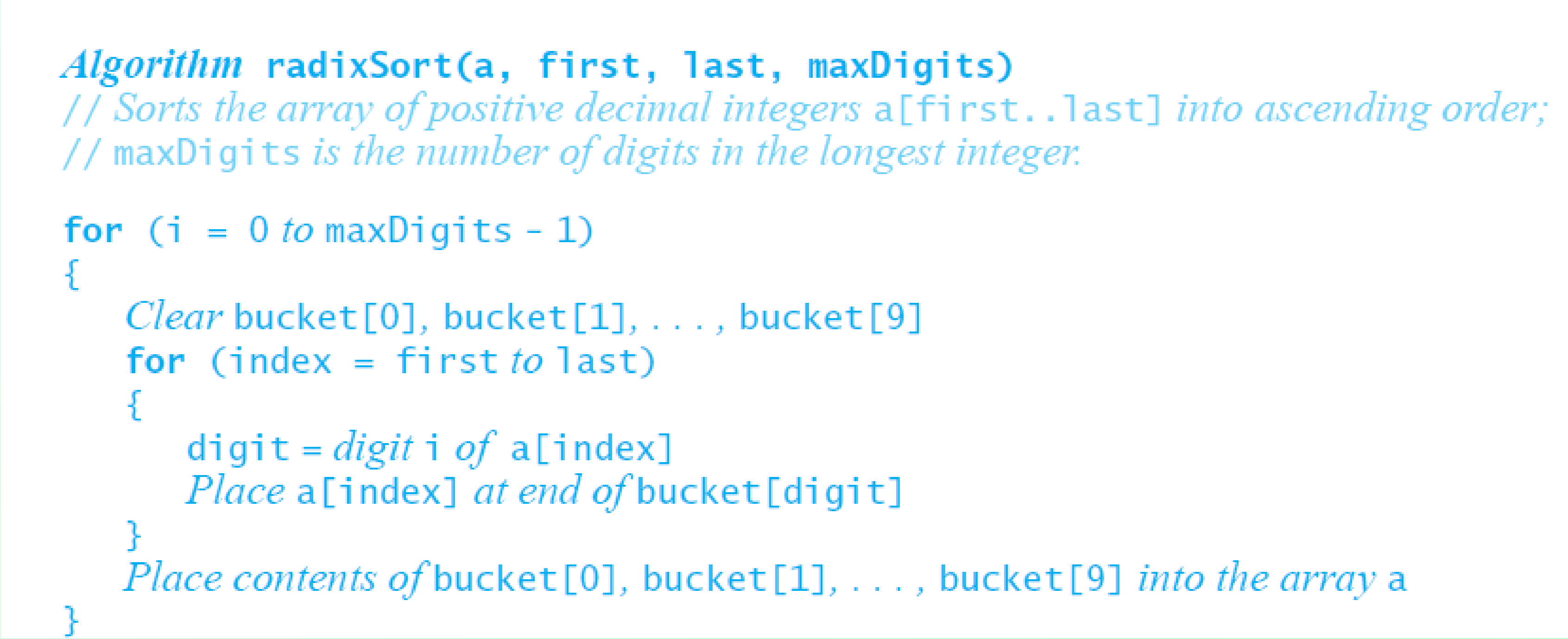
Distribute integers into buckets according to the leftmost digit

003 019	123 129	210 220 294	398	
0	1	2	3	4
513 528				
5	6	7	8	9

(d)

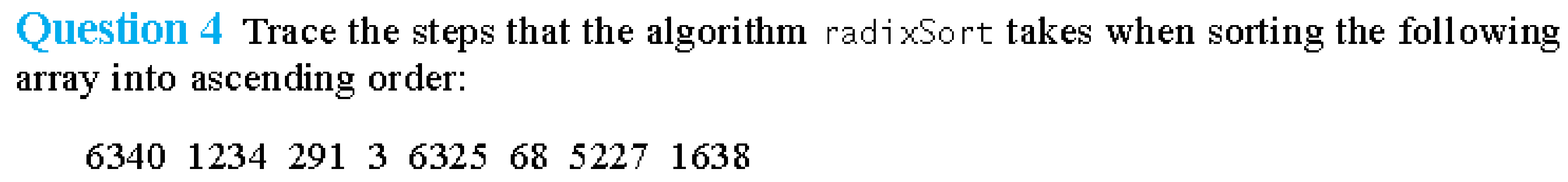
003	019	123	129	210	220	294	398	513	528

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



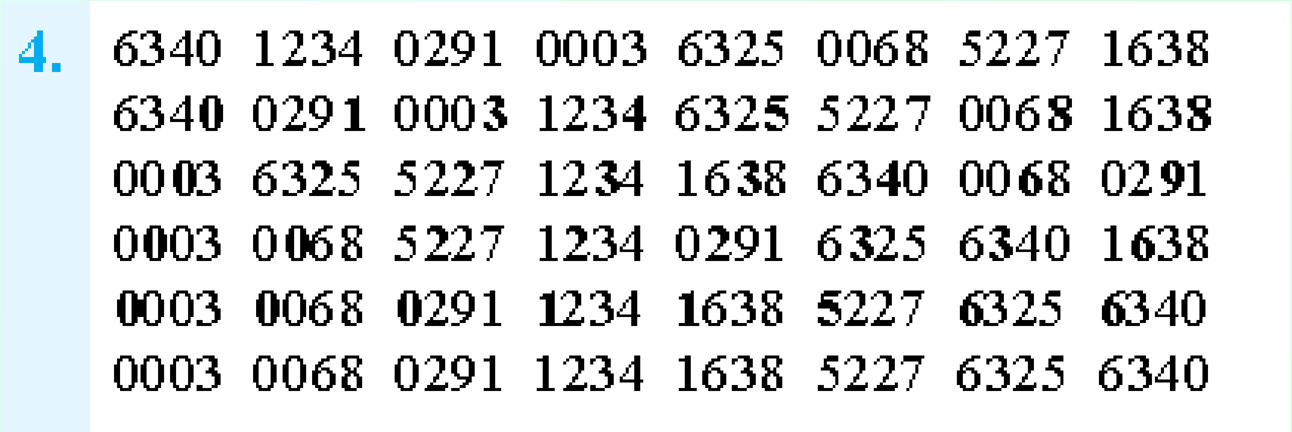
```
Algorithm radixSort(a, first, last, maxDigits)
// Sorts the array of positive decimal integers a[first..last] into ascending order;
// maxDigits is the number of digits in the longest integer.

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 Pseudocode

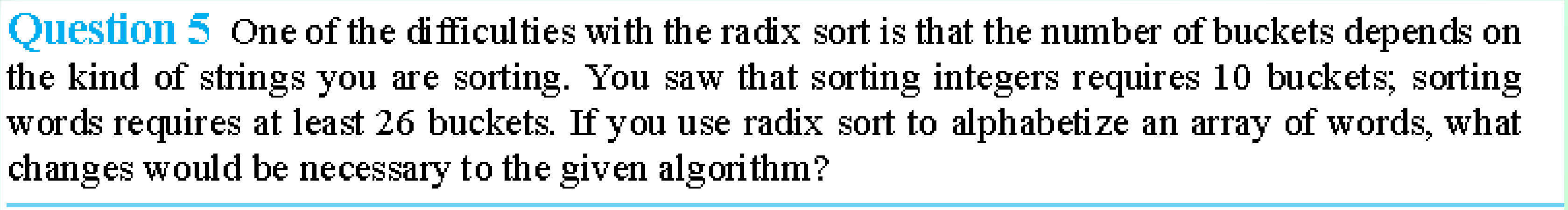


Question 4 Trace the steps that the algorithm `radixSort` takes when sorting the following array into ascending order:

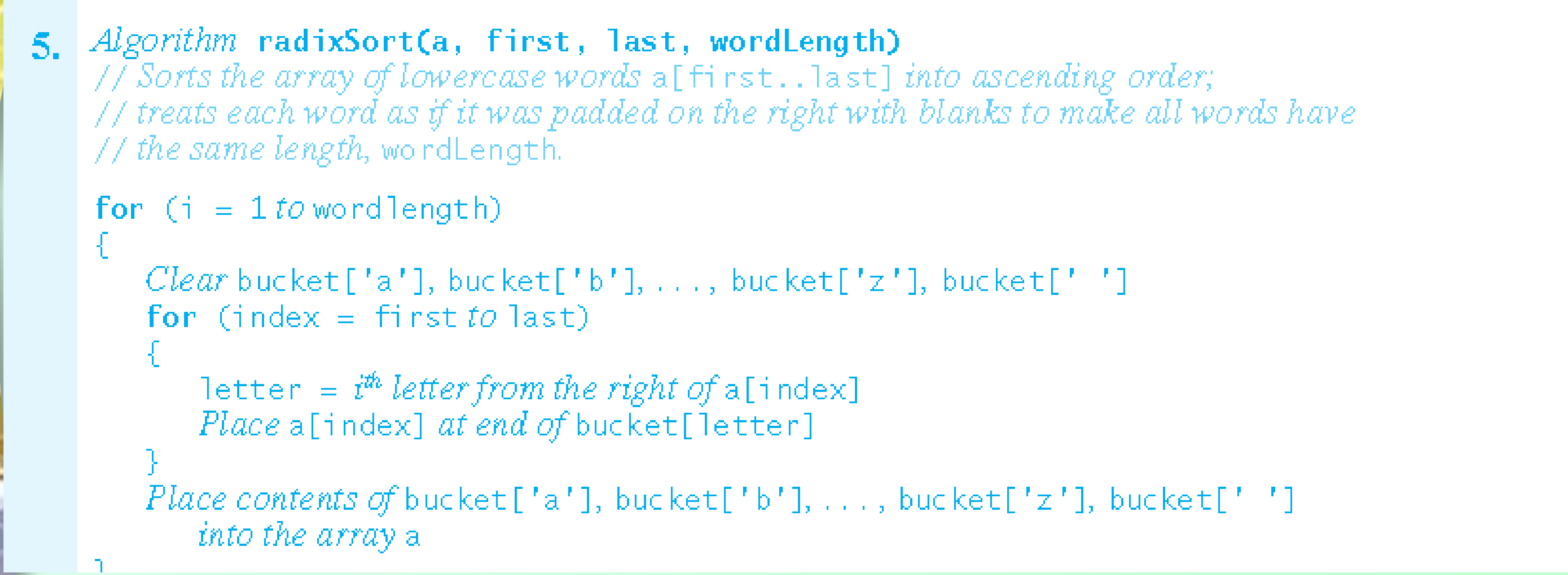
6340 1234 291 3 6325 68 5227 1638



4\. 6340 1234 0291 0003 6325 0068 5227 1638
6340 0291 0003 1234 6325 5227 0068 1638
0003 6325 5227 1234 1638 6340 0068 0291
0003 0068 5227 1234 0291 6325 6340 1638
0003 0068 0291 1234 1638 5227 6325 6340
0003 0068 0291 1234 1638 5227 6325 6340

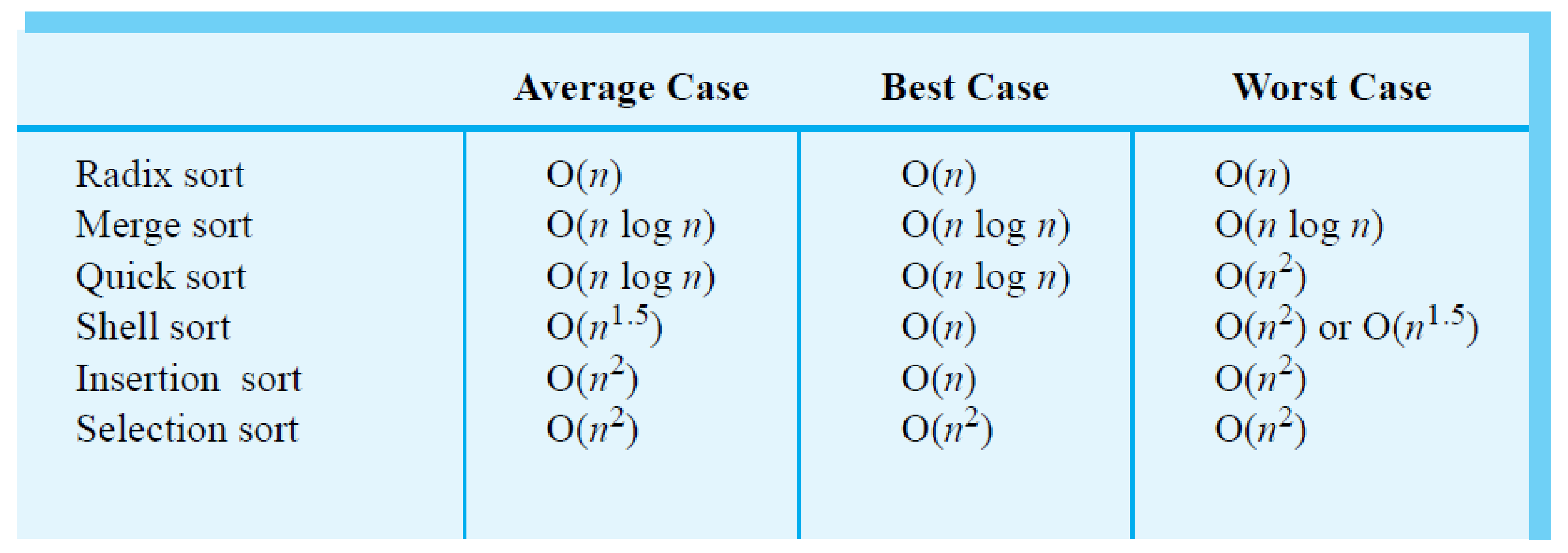


Question 5 One of the difficulties with the radix sort is that the number of buckets depends on the kind of strings you are sorting. You saw that sorting integers requires 10 buckets; sorting words requires at least 26 buckets. If you use radix sort to alphabetize an array of words, what changes would be necessary to the given algorithm?



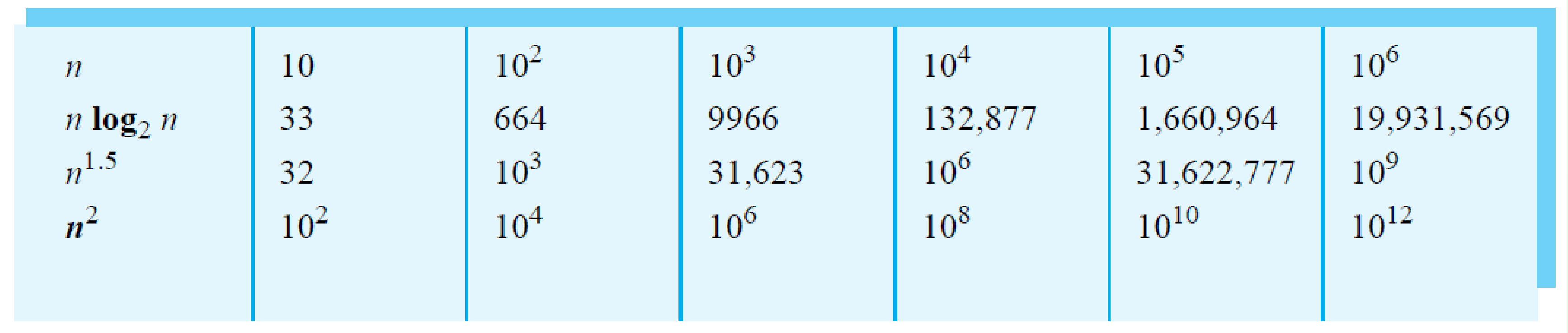
5.

```
Algorithm radixSort(a, first, last, wordLength)
// Sorts the array of lowercase words a[first..last] into ascending order;
// treats each word as if it was padded on the right with blanks to make all words have
// the same length, wordLength.

for (i = 1 to wordlength)
{
    Clear bucket['a'], bucket['b'], . . . , bucket['z'], bucket[' ']
    for (index = first to last)
    {
        letter = ith letter from the right of a[index]
        Place a[index] at end of bucket[letter]
    }
    Place contents of bucket['a'], bucket['b'], . . . , bucket['z'], bucket[' ']
        into the array a
}
```



	Average Case	Best 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	$O(n^{1.5})$	$O(n)$	$O(n^2)$ or $O(n^{1.5})$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Figure 9-10 The time efficiency of various sorting algorithms, expressed in Big Oh notation



n	10	10^2	10^3	10^4	10^5	10^6
$n \log_2 n$	33	664	9966	132,877	1,660,964	19,931,569
$n^{1.5}$	32	10^3	31,623	10^6	31,622,777	10^9
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}

Figure 9-11 A comparison of growth-rate functions as n increases

End

Chapter 9