



CS/COE 0445 - Data Structures

Week 9-10: Sorting

Sherif Khattab

<http://www.cs.pitt.edu/~skhattab/cs0445>

Administrivia

- Assignment 3 due on Friday 3/16 @11:59pm

Agenda

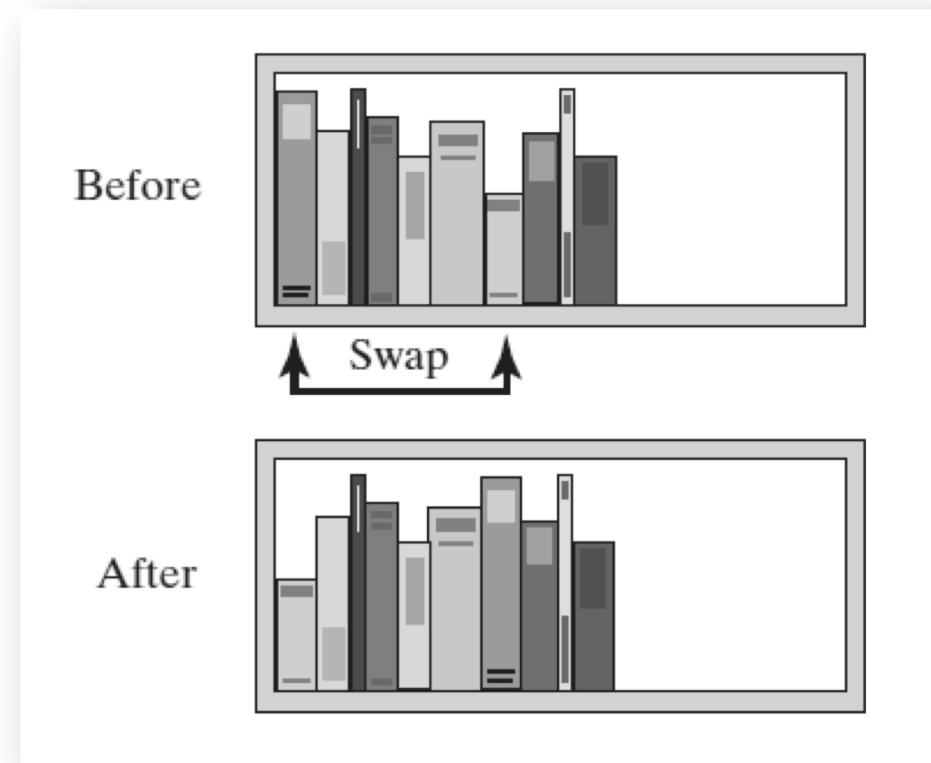
- Sorting
 - $O(n^2)$
 - SelectionSort
 - InsertionSort
 - $O(n \log n)$
 - MergeSort
 - QuickSort
 - $O(1)$ Sorting
 - Radix Sort

Sorting

- We seek algorithms to arrange items, a_i , such that
 $entry\ 1 \leq entry\ 2 \leq \dots \leq entry\ n$
- Sorting an array is usually easier than sorting a chain of linked nodes
- Efficiency of a sorting algorithm is significant

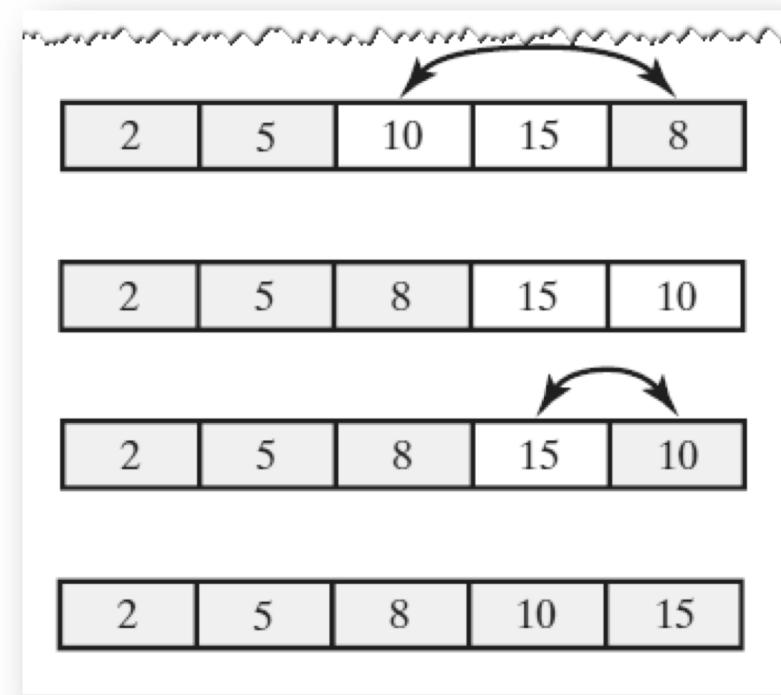
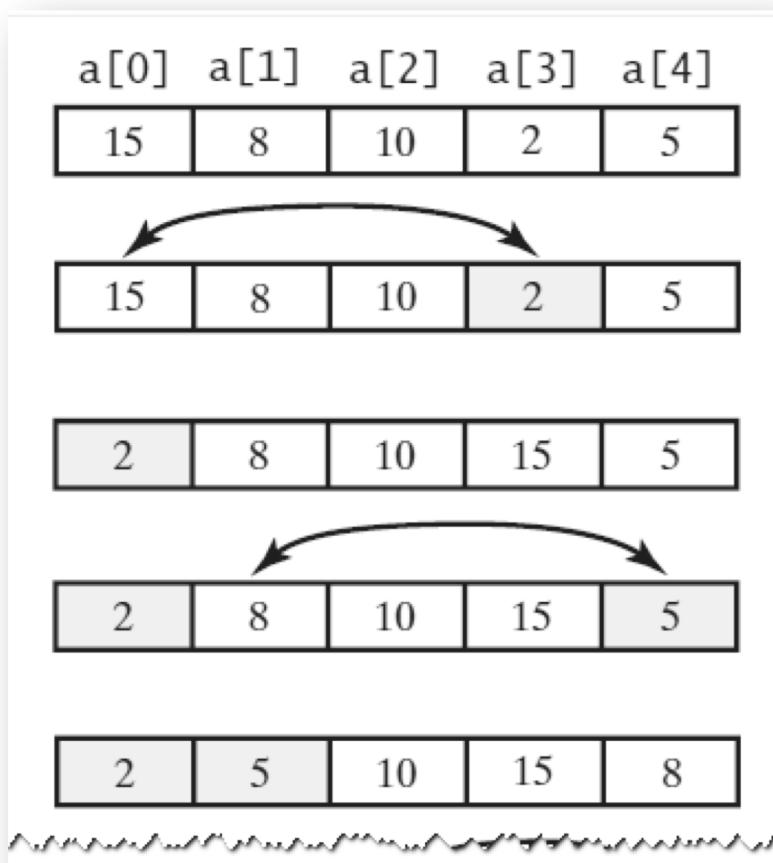
Selection Sort

- FIGURE 8-1 Before and after exchanging the shortest book and the first book



Selection Sort

- FIGURE 8-2 A selection sort of an array of integers into ascending order



Iterative Selection Sort

- This pseudocode describes an iterative algorithm for the selection sort

```
Algorithm selectionSort(a, n)
// Sorts the first n entries of an array a.

for (index = 0; index < n - 1; index++)
{
    indexOfNextSmallest = the index of the smallest value among
                          a[index], a[index + 1], . . . , a[n - 1]
    Interchange the values of a[index] and a[indexOfNextSmallest]
    // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[index], and these are the smallest
    // of the original array entries. The remaining array entries begin at a[index + 1].
}
```

Iterative Selection Sort

- LISTING 8-1 A class for sorting an array using selection sort

```
1  /**
2   * Class for sorting an array of Comparable objects from smallest to largest.
3  */
4  public class SortArray
5  {
6      /** Sorts the first n objects in an array into ascending order.
7       * @param a An array of Comparable objects.
8       * @param n An integer > 0. */
9      public static <T extends Comparable<? super T>>
10         void selectionSort(T[] a, int n)
11     {
12         for (int index = 0; index < n - 1; index++)
13         {
14             int indexOfNextSmallest = getIndexOfSmallest(a, index, n - 1);
15             swap(a, index, indexOfNextSmallest);
16             // Assertion: a[0] <= a[1] <= . . . <= a[index] <= all other a[i].
17         } // end for
18     } // end selectionSort
```

Iterative Selection Sort

- LISTING 8-1 A class for sorting an array using selection sort

```
14     indexOrNextSmallest = getIndexOfSmallest(a, index, "index", "i");
15         swap(a, index, indexOfNextSmallest);
16         // Assertion: a[0] <= a[1] <= . . . <= a[index] <= all other a[i].
17     } // end for
18 } // end selectionSort
19
20 // Finds the index of the smallest value in a portion of an array a.
21 // Precondition: a.length > last >= first >= 0.
22 // Returns the index of the smallest value among
23 // a[first], a[first + 1], . . . , a[last].
24 private static <T extends Comparable<? super T>>
25     int getIndexOfSmallest(T[] a, int first, int last)
26 {
27     T min = a[first];
```

Iterative Selection Sort

- LISTING 8-1 A class for sorting an array using selection sort

```
28     int indexOfMin = first;
29     for (int index = first + 1; index <= last; index++)
30     {
31         if (a[index].compareTo(min) < 0)
32         {
33             min = a[index];
34             indexOfMin = index;
35         } // end if
36         // Assertion: min is the smallest of a[first] through a[index].
37     } // end for
38     return indexOfMin;
```

Iterative Selection Sort

- LISTING 8-1 A class for sorting an array using selection sort

```
36     // Assertion: min is the smallest of a[first] through a[index].
37     } // end for
38     return indexOfMin;
39
40 } // end getIndexOfSmallest
41 // Swaps the array entries a[i] and a[j].
42
43 private static void swap(Object[] a, int i, int j)
44 {
45     Object temp = a[i];
46     a[i] = a[j];
47     a[j] = temp;
48 } // end swap
49 } // end SortArray
```

Recursive Selection Sort

- Recursive selection sort algorithm

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

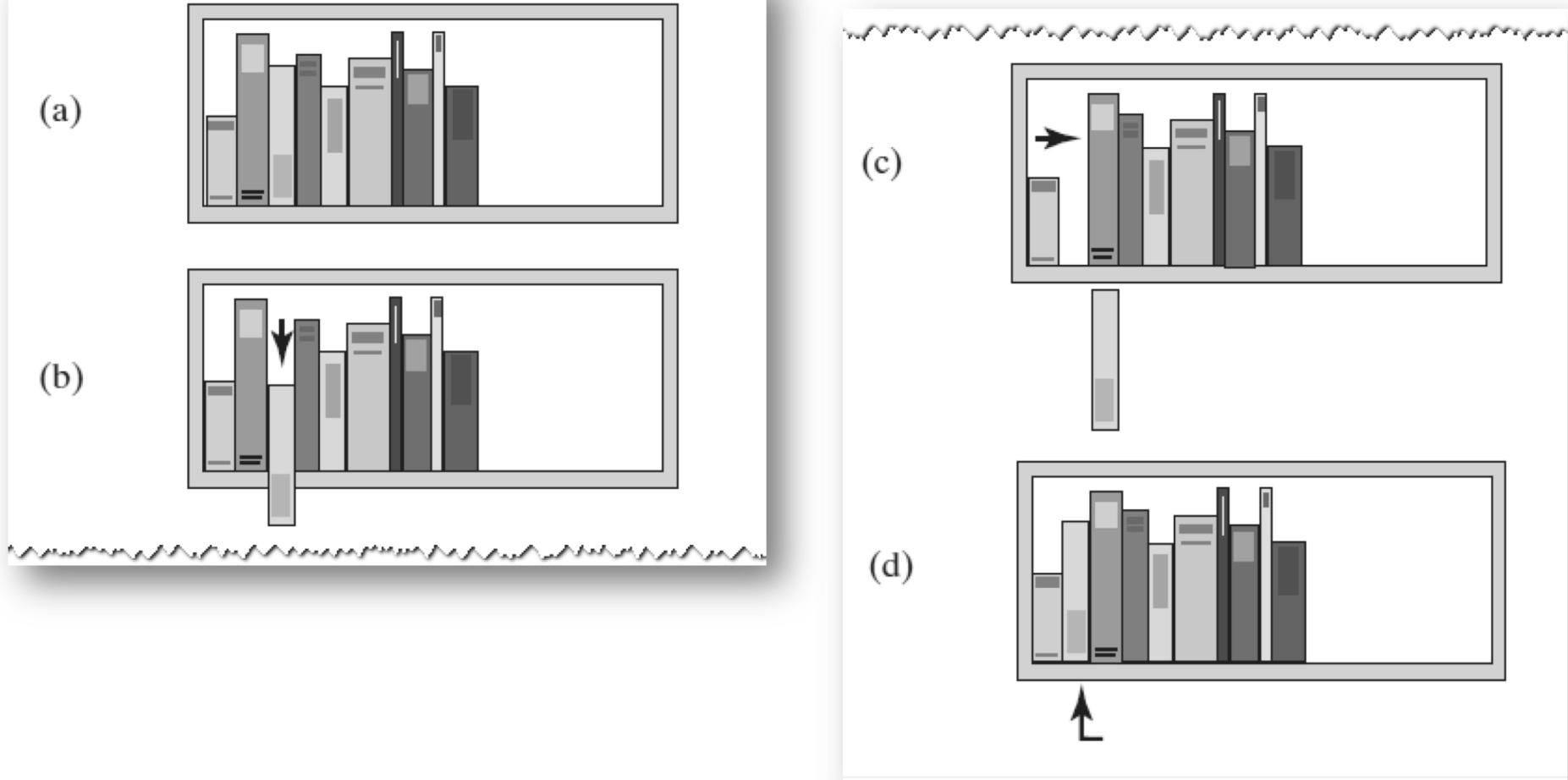
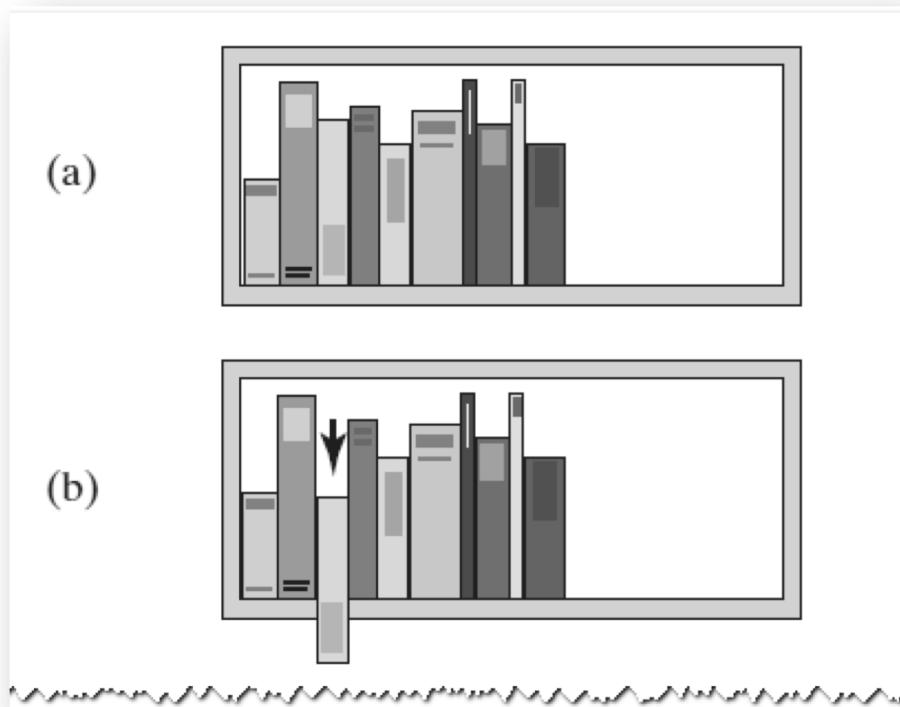
if (first < last)
{
    indexOfNextSmallest = the index of the smallest value among
                           a[first], a[first + 1], . . . , a[last]
    Interchange the values of a[first] and a[indexOfNextSmallest]
    // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[first] and these are the smallest
    // of the original array entries. The remaining array entries begin at a[first + 1].
    selectionSort(a, first + 1, last)
}
```

Efficiency of Selection Sort

- Selection sort is $O(n^2)$ regardless of the initial order of the entries.
 - Requires $O(n^2)$ comparisons
 - Does only $O(n)$ swaps

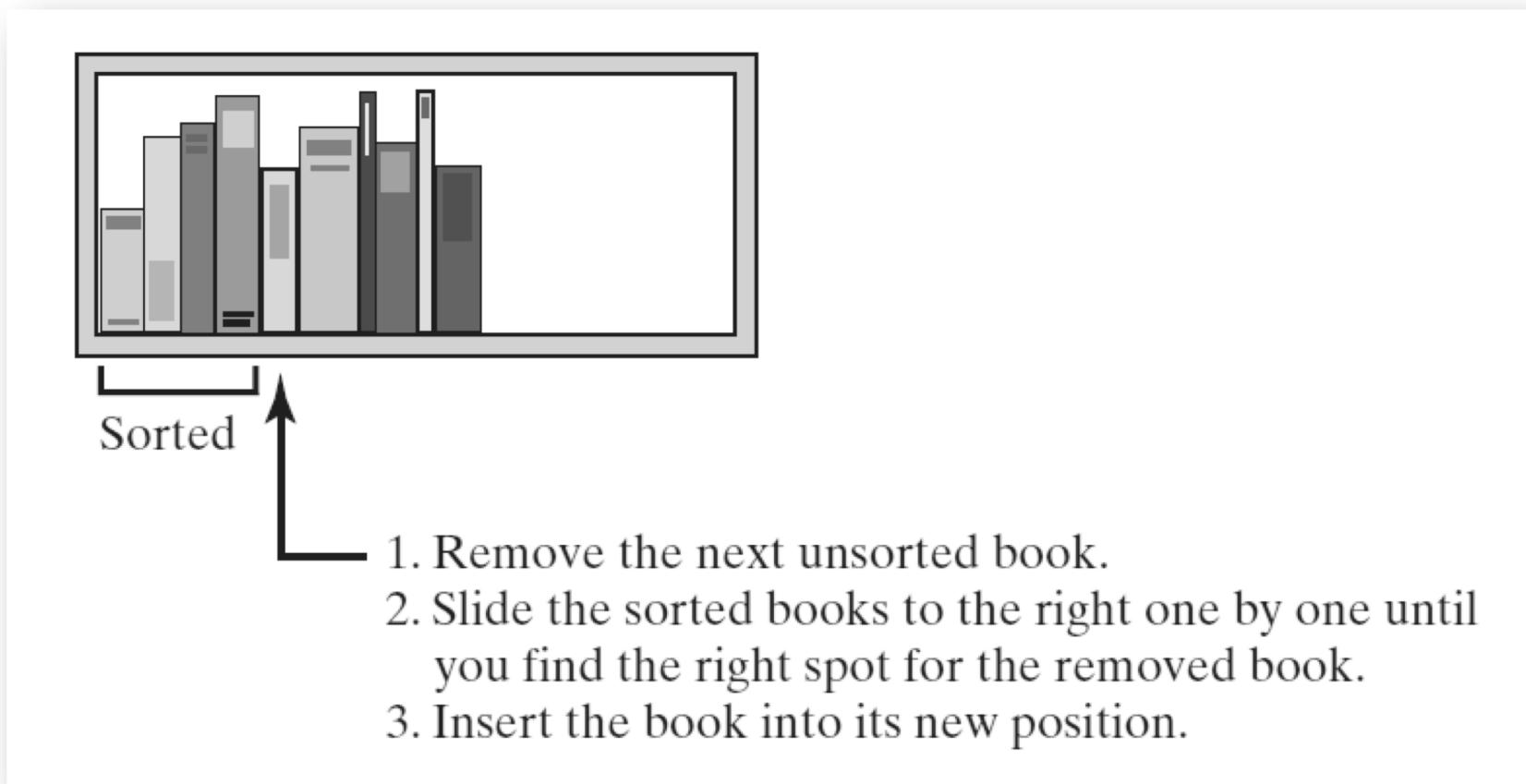
Insertion Sort

- FIGURE 8-3 The placement of the third book during an insertion sort



Insertion Sort

- FIGURE 8-4 An insertion sort of books



Iterative Insertion Sort

- Iterative algorithm describes an insertion sort of the entries at indices **first** through **last** of the array **a**

```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] iteratively.

for (unsorted = first + 1 through last)
{
    nextToInsert = a[unsorted]
    insertInOrder(nextToInsert, a, first, unsorted - 1)
}
```

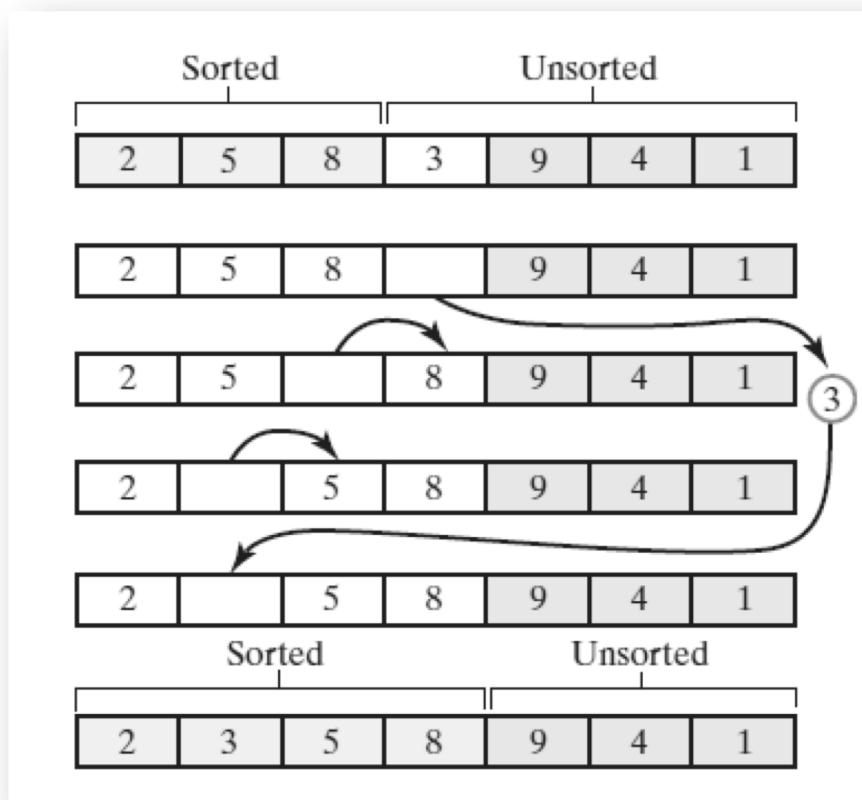
Iterative Insertion Sort

- Pseudocode of method, **insertInOrder**, to perform the insertions.

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted entries a[begin] through a[end].
index = end          // Index of last entry in the sorted portion
// Make room, if needed, in sorted portion for another entry
while ( (index >= begin) and (anEntry < a[index]) )
{
    a[index + 1] = a[index] // Make room
    index--
}
// Assertion: a[index + 1] is available.
a[index + 1] = anEntry // Insert
```

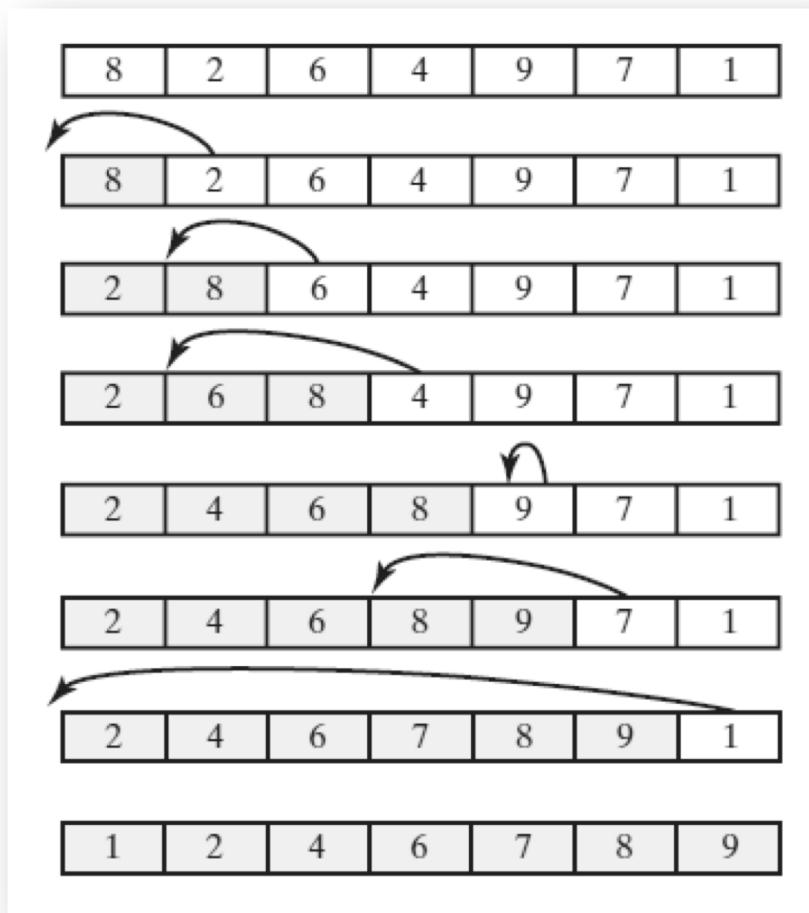
Iterative Insertion Sort

- FIGURE 8-5 Inserting the next unsorted entry into its proper location within the sorted portion of an array during an insertion sort



Iterative Insertion Sort

- FIGURE 8-6 An insertion sort of an array of integers into ascending order



Recursive Insertion Sort

- This pseudocode describes a recursive insertion sort.

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

if (the array contains more than one entry)
{
    Sort the array entries a[first] through a[last - 1]
    Insert the last entry a[last] into its correct sorted position within the rest of the array
}
```

Recursive Insertion Sort

- Implementing the algorithm in Java

```
public static <T extends Comparable<? super T>>
    void insertionSort(T[] a, int first, int last)
{
    if (first < last)
    {
        // Sort all but the last entry
        insertionSort(a, first, last - 1);

        // Insert the last entry in sorted order
        insertInOrder(a[last], a, first, last - 1);
    } // end if
} // end insertionSort
```

Recursive Insertion Sort

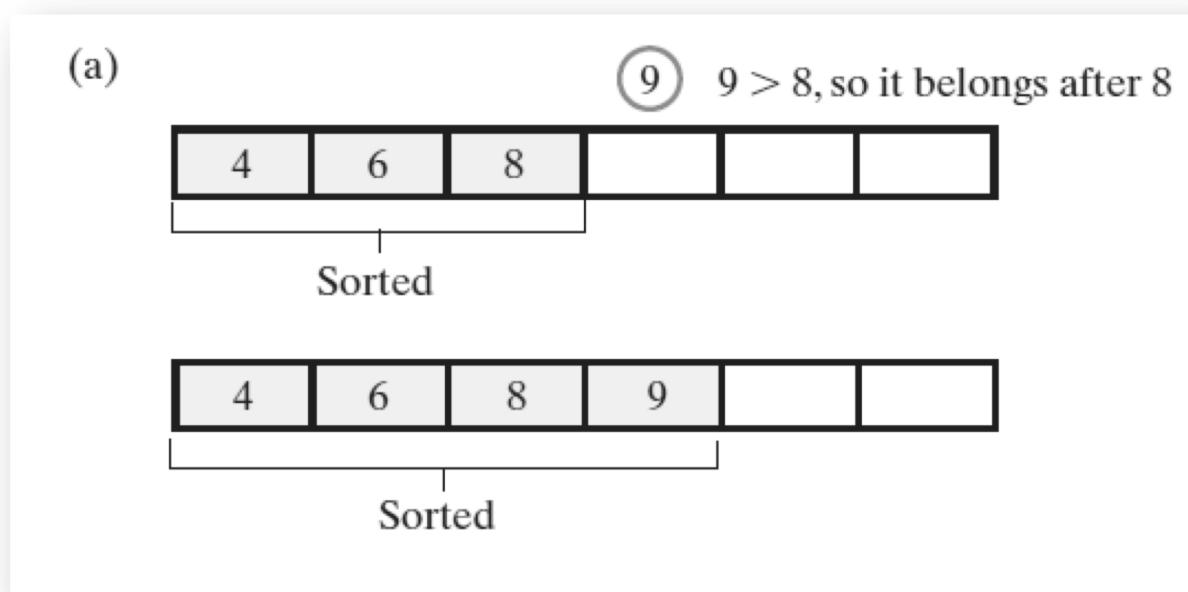
- First draft of **insertInOrder** algorithm.

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// First draft.

if (anEntry >= a[end])
    a[end + 1] = anEntry
else
{
    a[end + 1] = a[end]
    insertInOrder(anEntry, a, begin, end - 1)
}
```

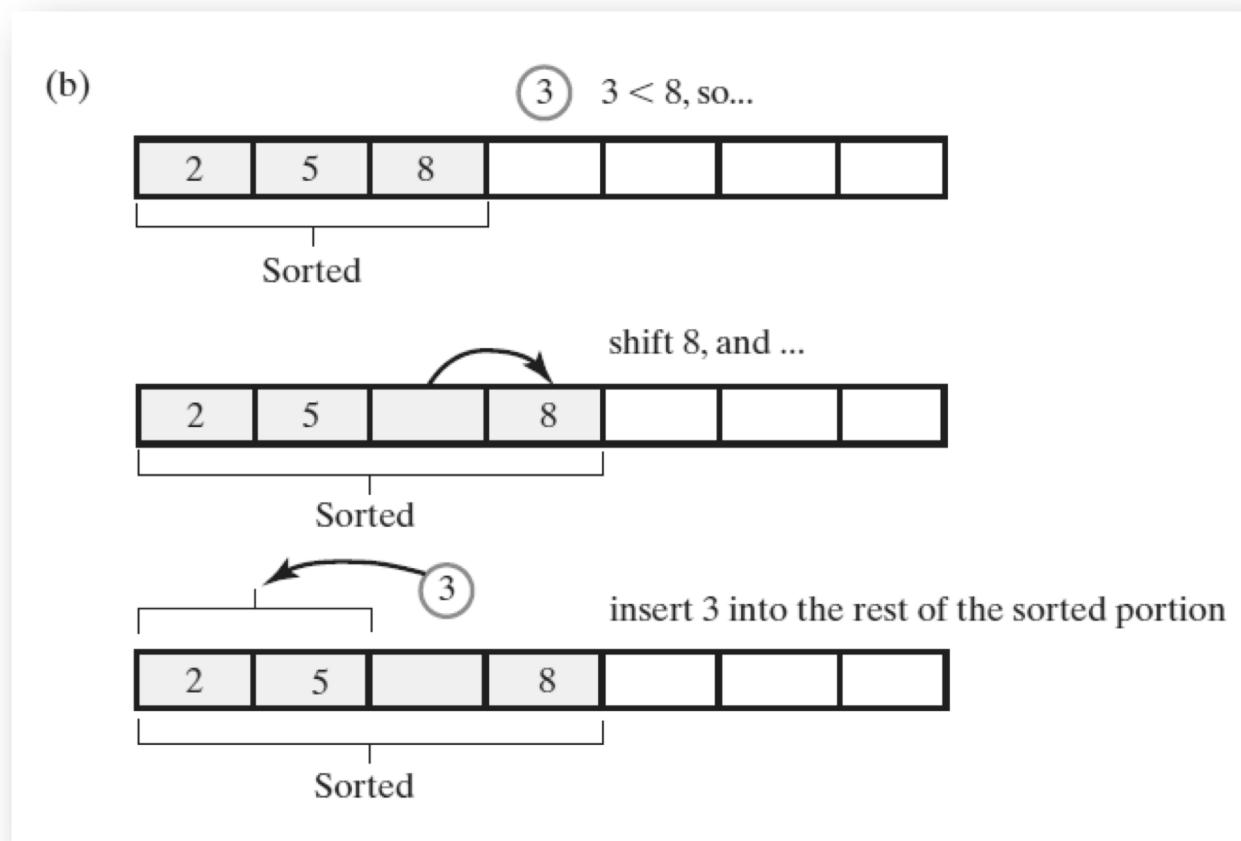
Recursive Insertion Sort

- FIGURE 8-8 Inserting the first unsorted entry into the sorted portion of the array. (a) The entry is greater than or equal to the last sorted entry



Recursive Insertion Sort

- FIGURE 8-8 Inserting the first unsorted entry into the sorted portion of the array. (b) the entry is smaller than the last sorted entry



Recursive Insertion Sort

- The algorithm **insertInOrder**: final draft.

Note: insertion sort efficiency (worst case) is $O(n^2)$

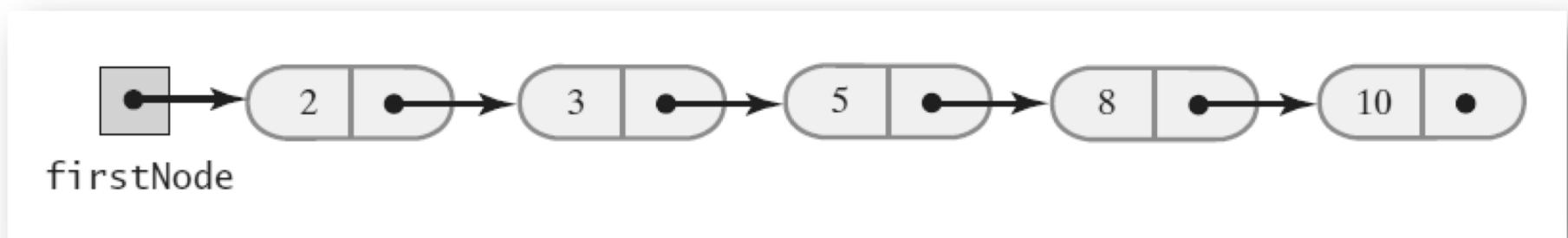
```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// Revised draft.

if (anEntry >= a[end])
    a[end + 1] = anEntry

else if (begin < end)
{
    a[end + 1] = a[end]
    insertInOrder(anEntry, a, begin, end - 1)
}
else // begin == end and anEntry < a[end]
{
    a[end + 1] = a[end]
    a[end] = anEntry
}
```

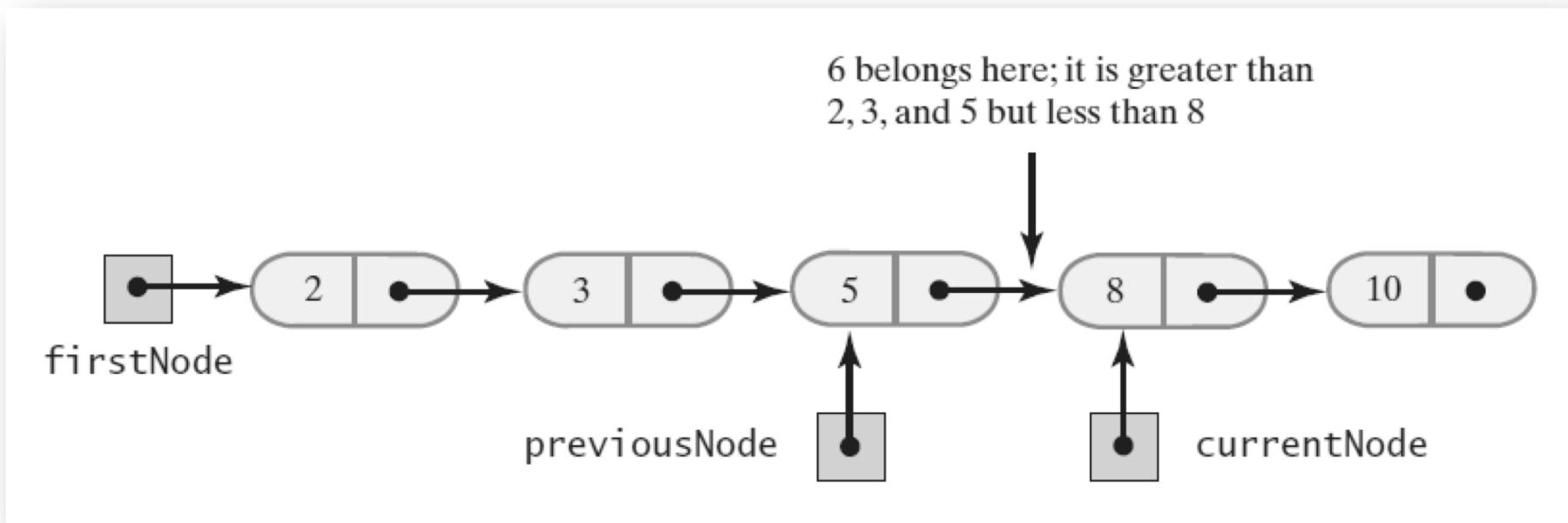
Insertion Sort of a Chain of Linked Nodes

- FIGURE 8-8 A chain of integers sorted into ascending order



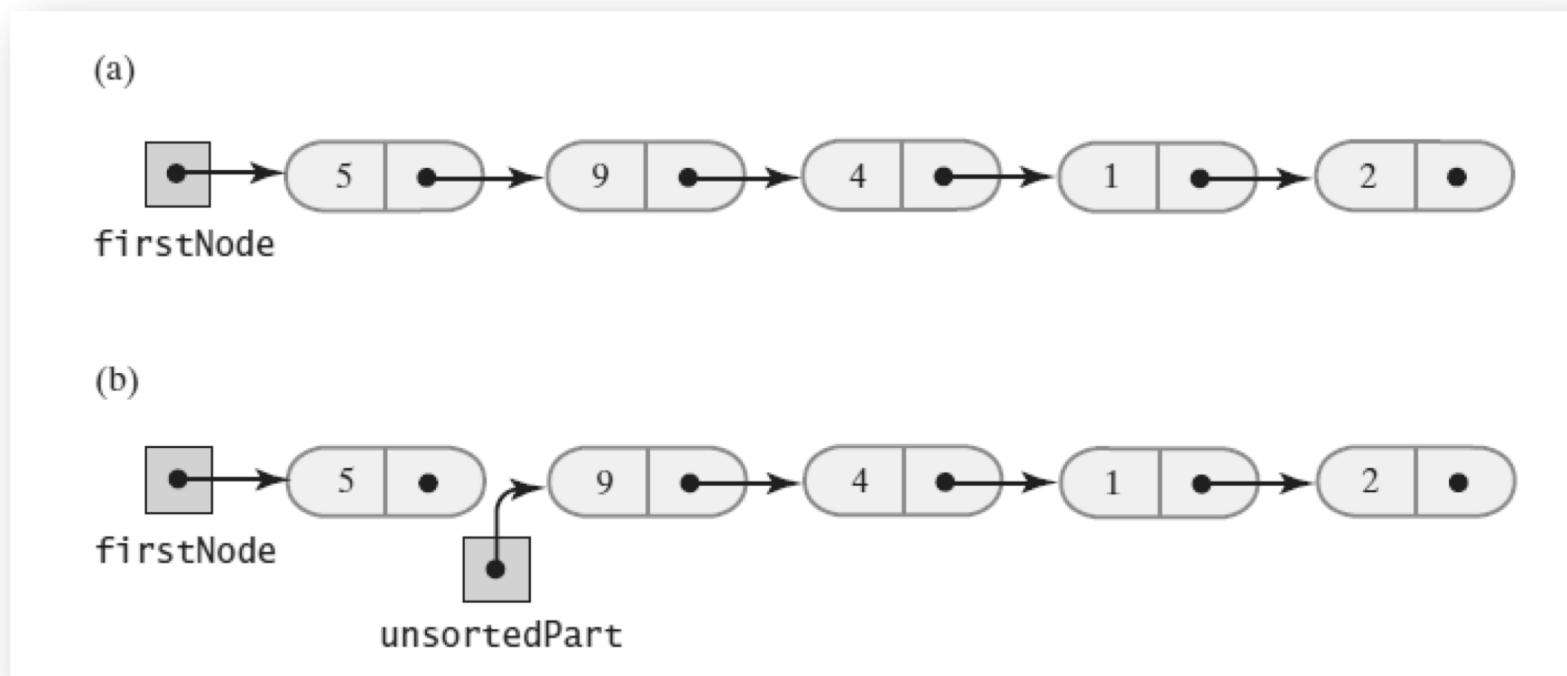
Insertion Sort of a Chain of Linked Nodes

- FIGURE 8-9 During the traversal of a chain to locate the insertion point, save a reference to the node before the current one



Insertion Sort of a Chain of Linked Nodes

- FIGURE 8-10 Breaking a chain of nodes into two pieces as the first step in an insertion sort: (a) the original chain;
(b) the two pieces



Insertion Sort of a Chain of Linked Nodes

- Add a sort method to a class **LinkedGroup**
- that uses a linked chain to represent a certain collection

```
public class LinkedGroup<T extends Comparable<? super T>>
{
    private Node firstNode;
    int length; // Number of objects in the group
    . . .
```

Insertion Sort of a Chain of Linked Nodes

- This class has an inner class **Node** that has set and get methods

```
private void insertInOrder(Node nodeToInsert)
{
    T item = nodeToInsert.getData();
    Node currentNode = firstNode;
    Node previousNode = null;

    // Locate insertion point
    while ( (currentNode != null) &&
            (item.compareTo(currentNode.getData()) > 0) )
    {
        previousNode = currentNode;
        currentNode = currentNode.getNextNode();
    } // end while

    // Make the insertion
```

Insertion Sort of a Chain of Linked Nodes

- This class has an inner class **Node** that has set and get methods

```
    } // end while

    // Make the insertion
    if (previousNode != null)
    { // Insert between previousNode and currentNode
        previousNode.setNextNode(nodeToInsert);
        nodeToInsert.setNextNode(currentNode);
    }
    else // Insert at beginning
    {
        nodeToInsert.setNextNode(firstNode);
        firstNode = nodeToInsert;
    } // end if
} // end insertInOrder
```

Insertion Sort of a Chain of Linked Nodes

- The method to perform the insertion sort.

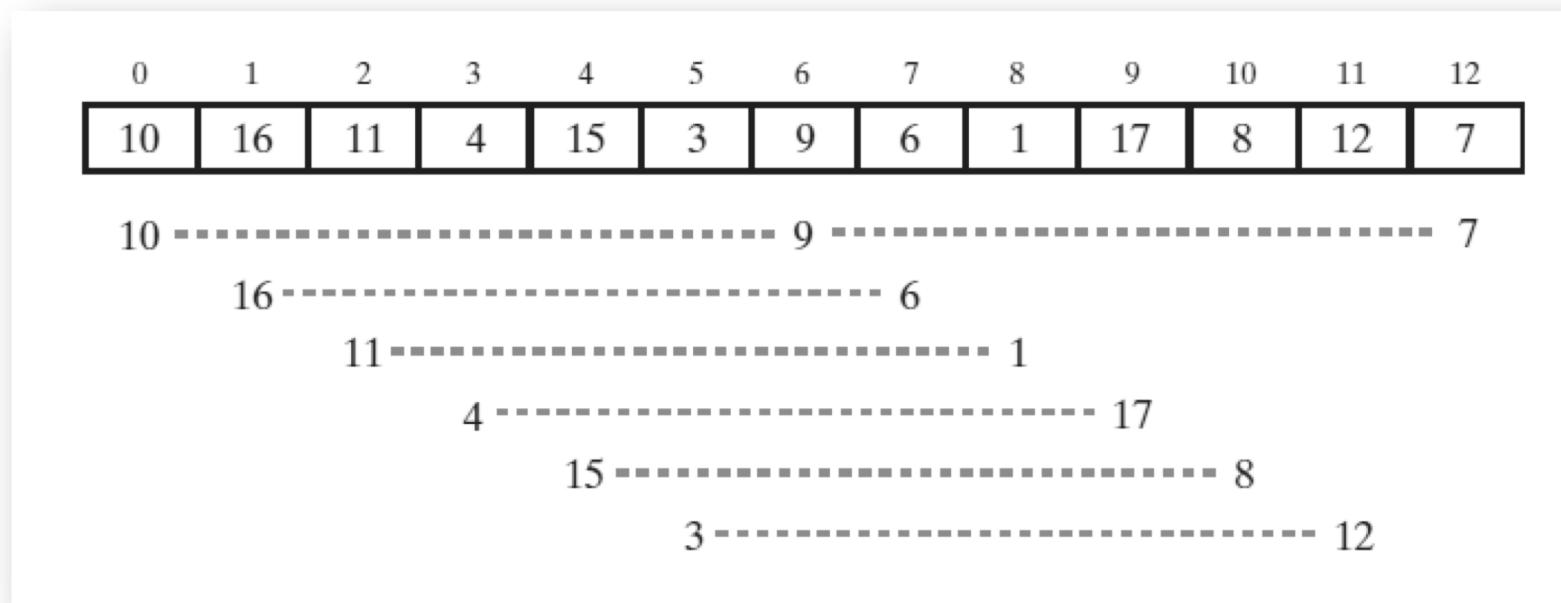
```
public void insertionSort()
{
    // If zero or one item is in the chain, there is nothing to do
    if (length > 1)
    {
        assert firstNode != null;
        // Break chain into 2 pieces: sorted and unsorted
        Node unsortedPart = firstNode.getNextNode();
        assert unsortedPart != null;
        firstNode.setNextNode(null);
        while (unsortedPart != null)
        {
            Node nodeToInsert = unsortedPart;
            unsortedPart = unsortedPart.getNextNode();
            insertInOrder(nodeToInsert);
        } // end while
    } // end if
} // end insertionSort
```

Shell Sort

- Algorithms seen so far are simple but inefficient for large arrays at $O(n^2)$
- Note, the more sorted an array is, the less work **insertInOrder** must do
- Improved insertion sort developed by Donald Shell

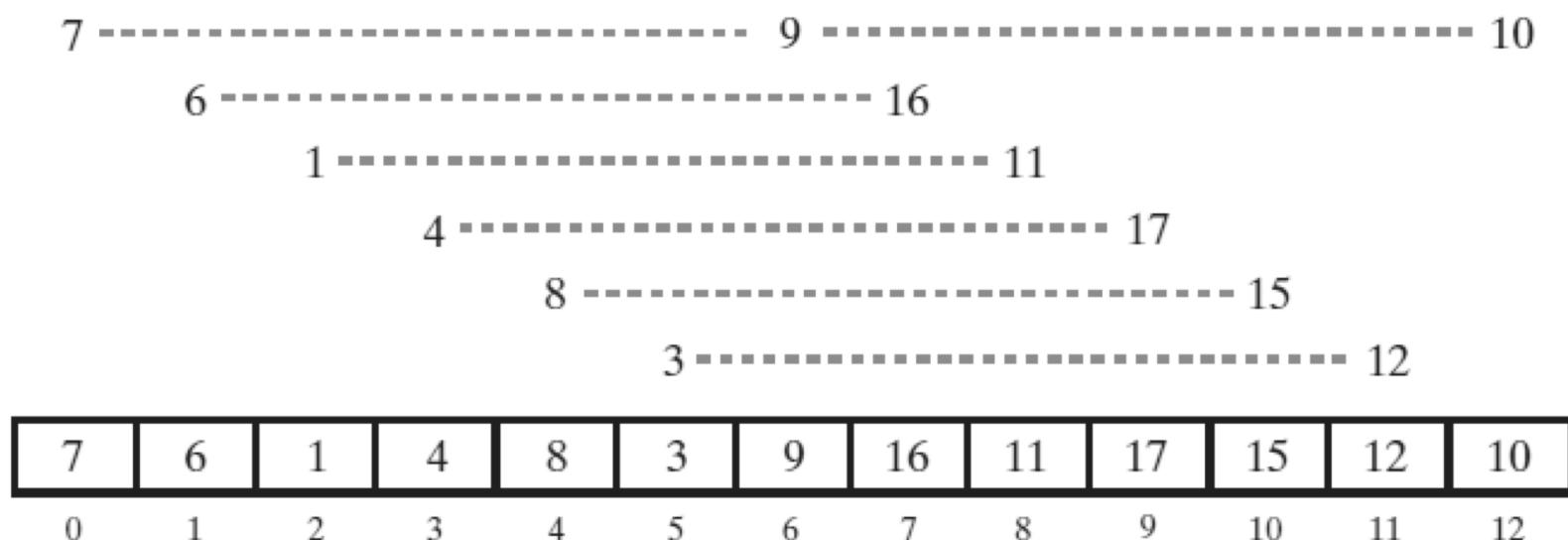
Shell Sort

- FIGURE 8-11 An array and the subarrays formed by grouping entries whose indices are 6 apart.



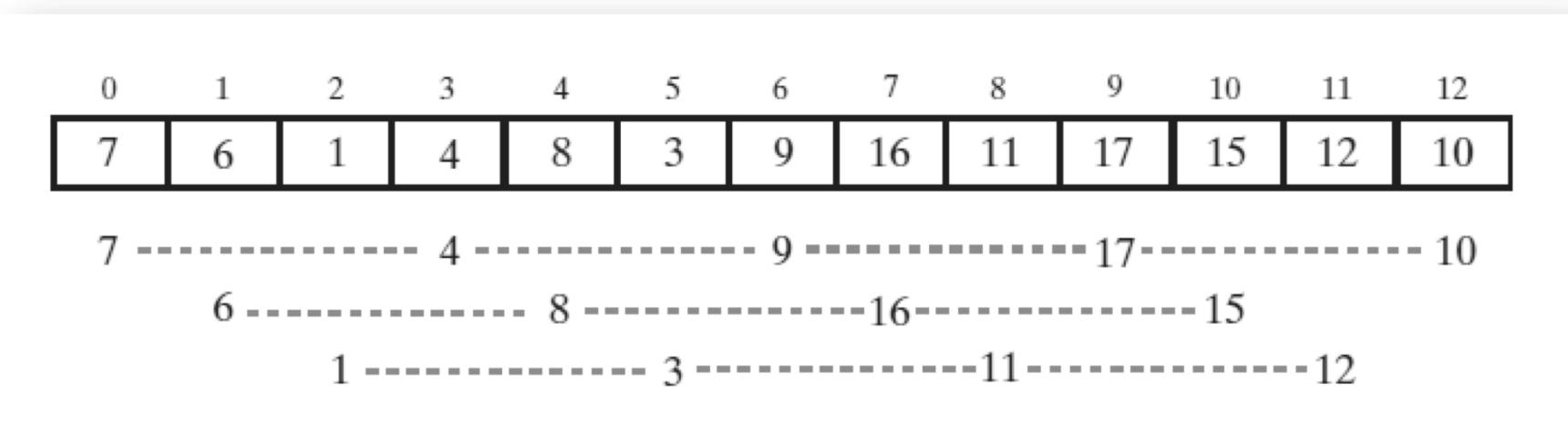
Shell Sort

- FIGURE 8-12 The subarrays of Figure 8-11 after each is sorted, and the array that contains them



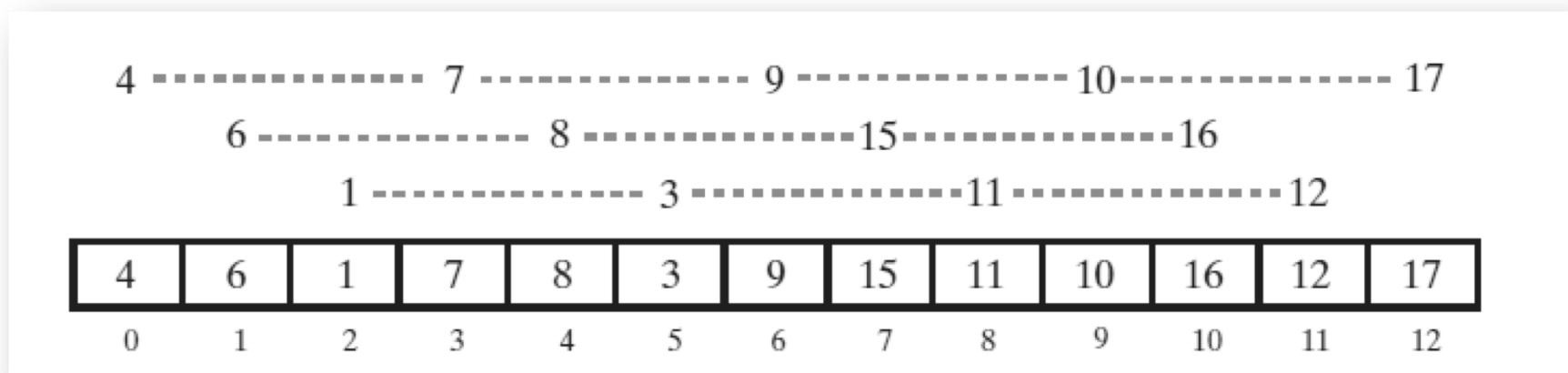
Shell Sort

- FIGURE 8-13 The subarrays of the array in Figure 8-12 formed by grouping entries whose indices are 3 apart



Shell Sort

- FIGURE 8-14 The subarrays of Figure 8-13 after each is sorted, and the array that contains them



Shell Sort

- Algorithm that sorts array entries whose indices are separated by an increment of **space**.

```
Algorithm incrementalInsertionSort(a, first, last, space)
// Sorts equally spaced entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length;
// space is the difference between the indices of the entries to sort.

for (unsorted = first + space through last at increments of space)
{
    nextToInsert = a[unsorted]
    index = unsorted - space
    while ( (index >= first) and (nextToInsert.compareTo(a[index]) < 0) )
    {
        a[index + space] = a[index]
        index = index - space
    }
    a[index + space] = nextToInsert
}
```

Shell Sort

- Algorithm to perform a Shell sort will invoke **incrementalInsertionSort** and supply any sequence of spacing factors. Efficiency (worst) can be $O(n^{1.5})$

```
Algorithm shellSort(a, first, last)
// Sorts the entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length.

n = number of array entries
space = n / 2
while (space > 0)
{
    for (begin = first through first + space - 1)
    {
        incrementalInsertionSort(a, begin, last, space)
    }
    space = space / 2
}
```

Comparing the Algorithms

- FIGURE 8-16 The time efficiencies of three sorting algorithms, expressed in Big Oh notation

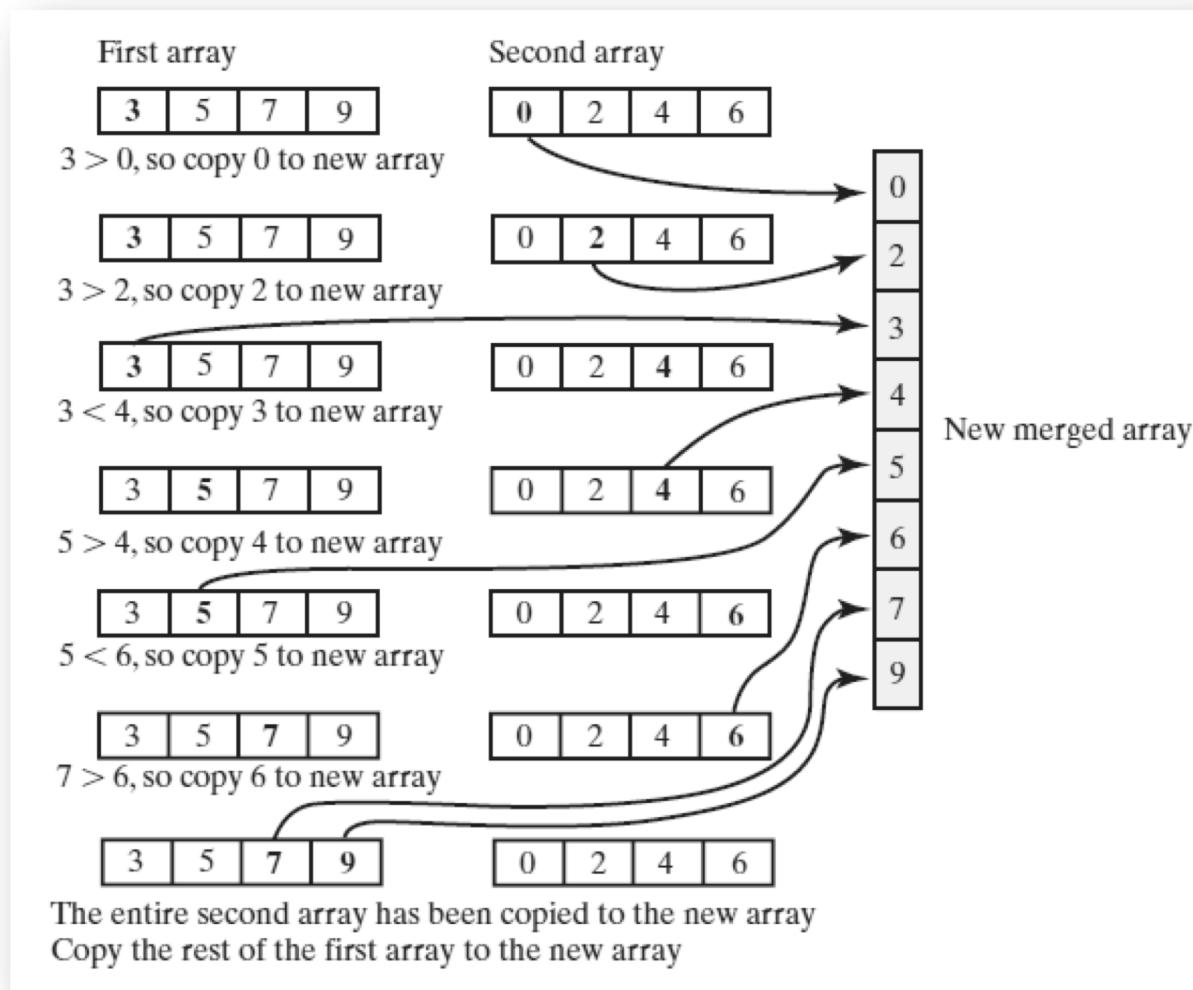
	Best Case	Average Case	Worst Case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n)$	$O(n^{1.5})$	$O(n^2)$ or $O(n^{1.5})$

Merge Sort

- Divides an array into halves
- Sorts the two halves,
 - Then merges them into one sorted array.
- The algorithm for merge sort is usually stated recursively.
- Major programming effort is in the merge process

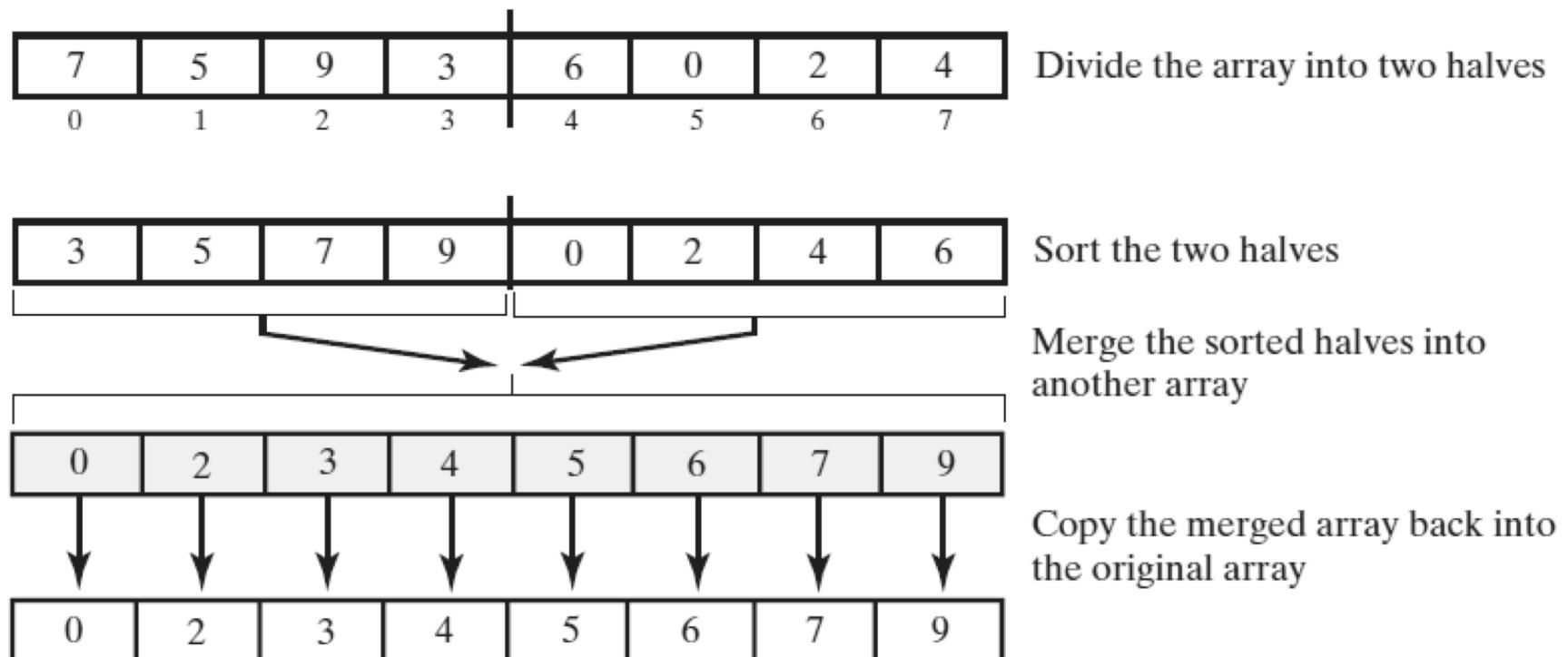
Merging Arrays

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



Recursive Merge Sort

- FIGURE 9-2 The major steps in a merge sort



Recursive Merge Sort

- Recursive algorithm for merge sort.

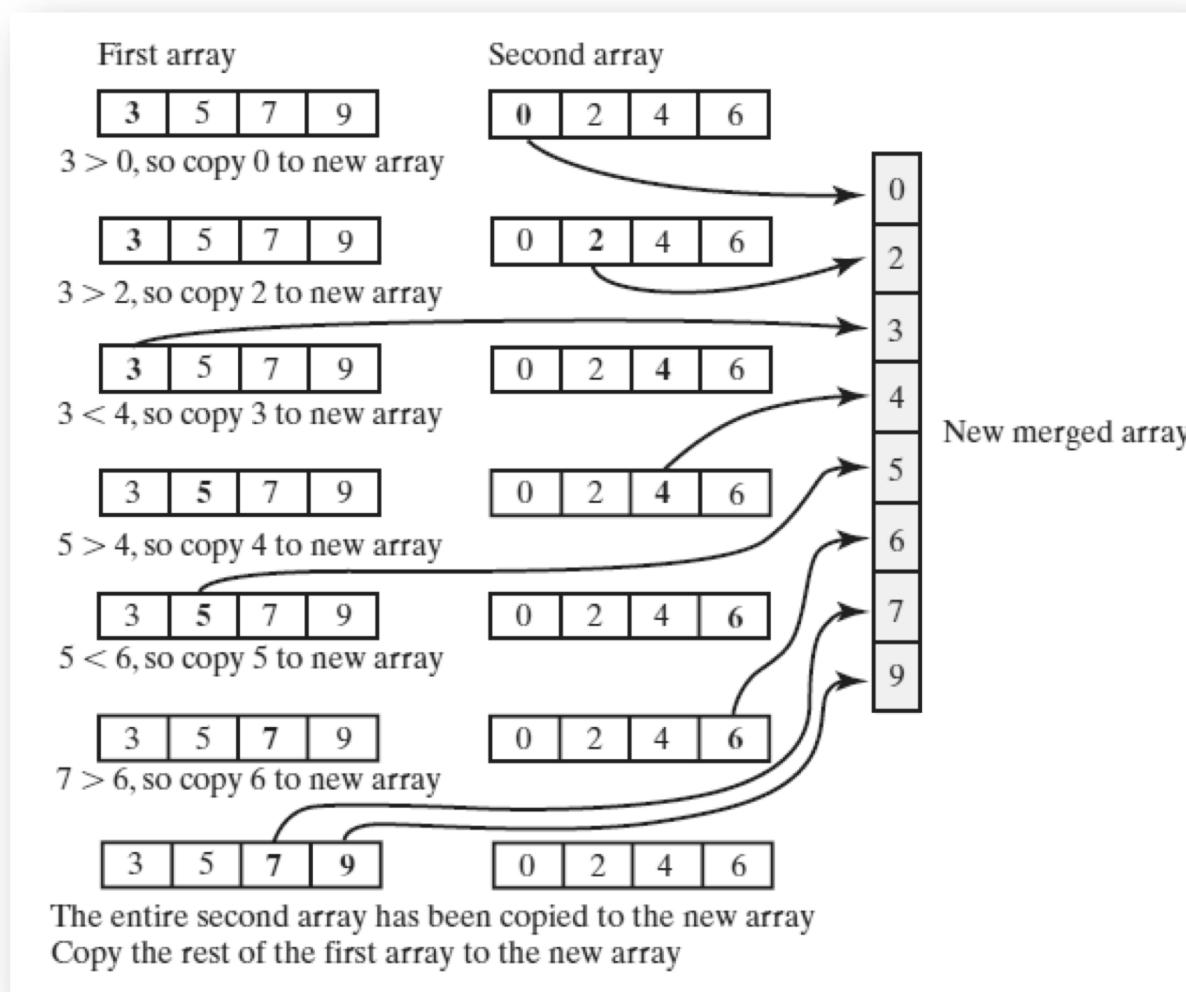
```
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.
if (first < last)
{
    mid = approximate midpoint between first and last
    mergeSort(a, tempArray, first, mid)
    mergeSort(a, tempArray, mid + 1, last)
    Merge the sorted halves a[first..mid] and a[mid + 1..last] using the array tempArray
}
```

Merge Sort

- Divides an array into halves
- Sorts the two halves,
 - Then merges them into one sorted array.
- The algorithm for merge sort is usually stated recursively.
- Major programming effort is in the merge process

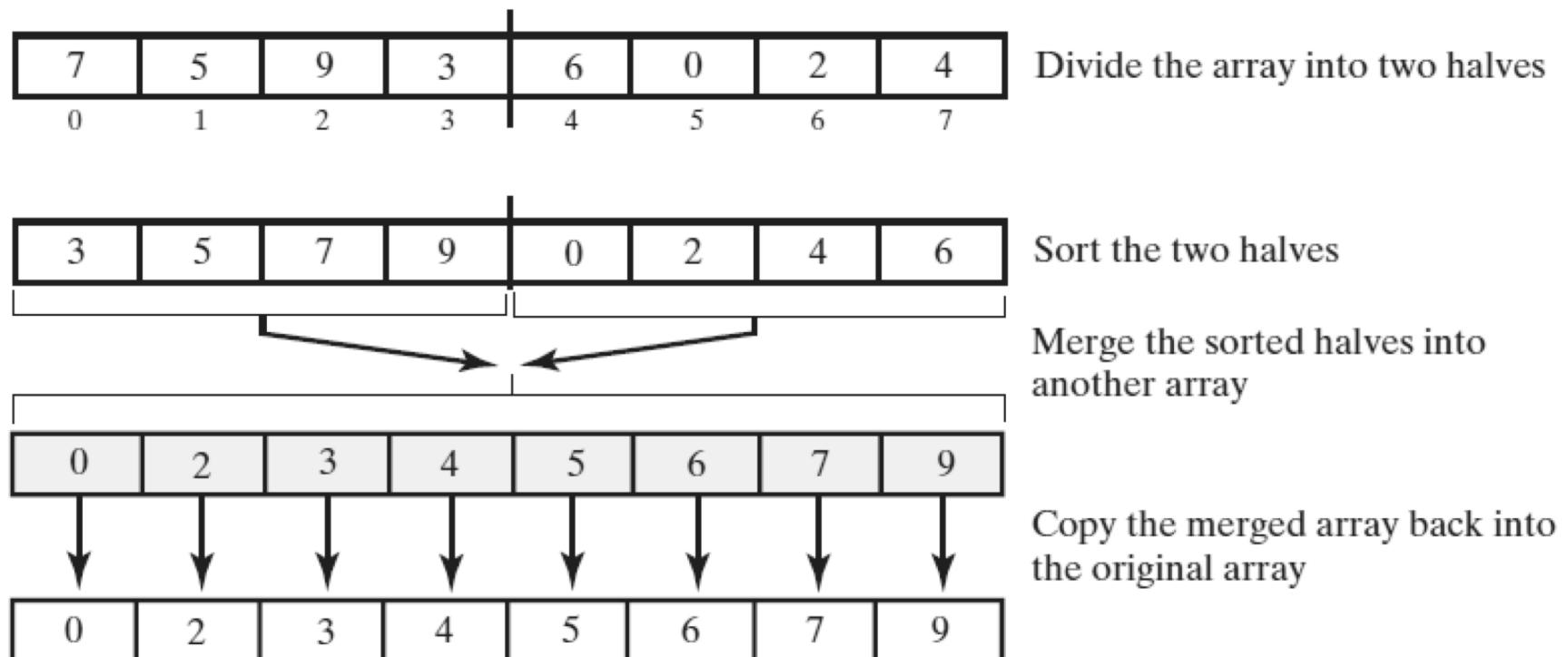
Merging Arrays

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



Recursive Merge Sort

- FIGURE 9-2 The major steps in a merge sort



Recursive Merge Sort

- Recursive algorithm for merge sort.

```
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.
if (first < last)
{
    mid = approximate midpoint between first and last
    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
}
```

Recursive Merge Sort

- Pseudocode which describes the merge step.

```
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++
    }
}
```

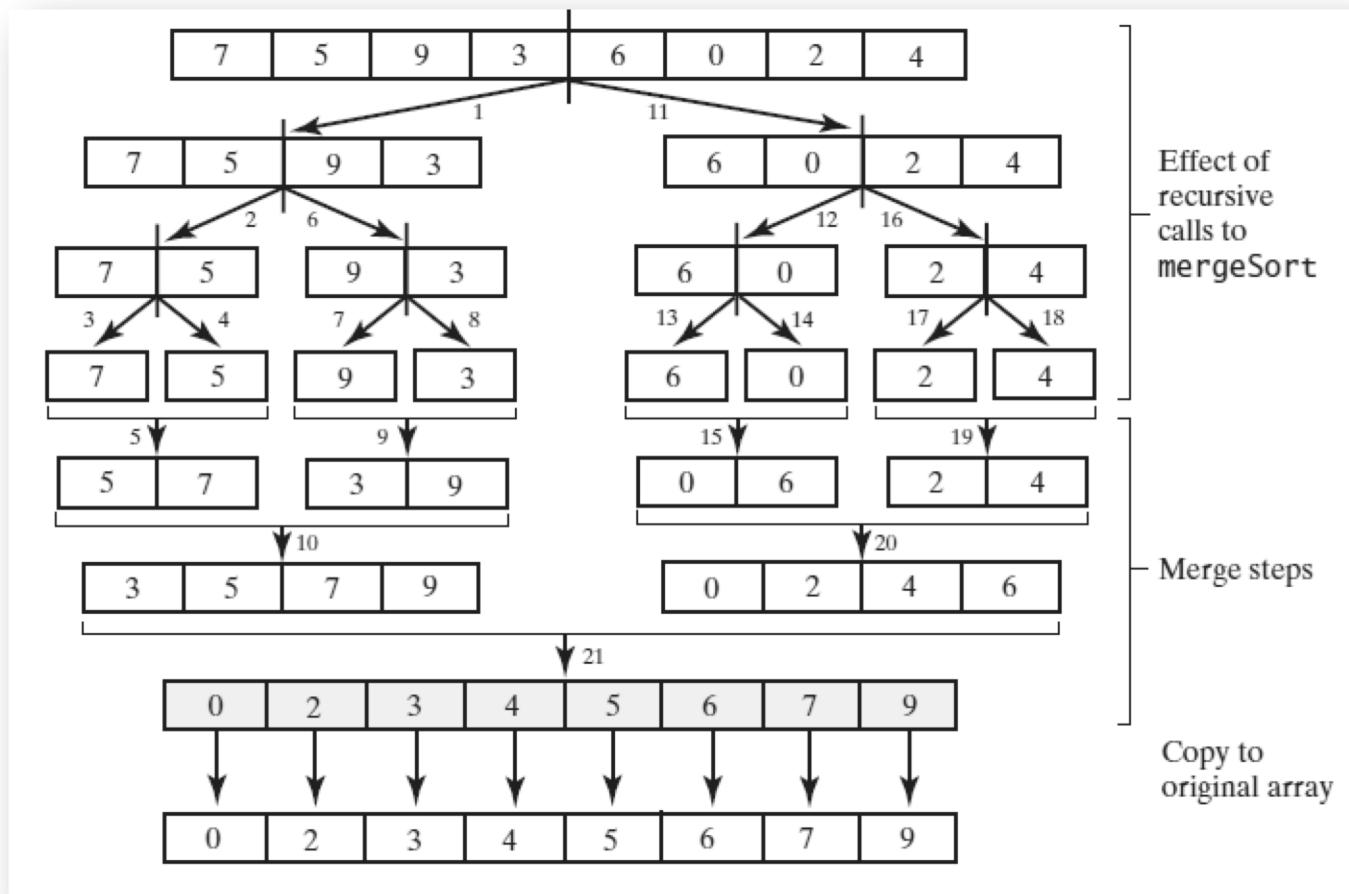
Recursive Merge Sort

- Pseudocode which describes the merge step.

```
        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
```

Recursive Merge Sort

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



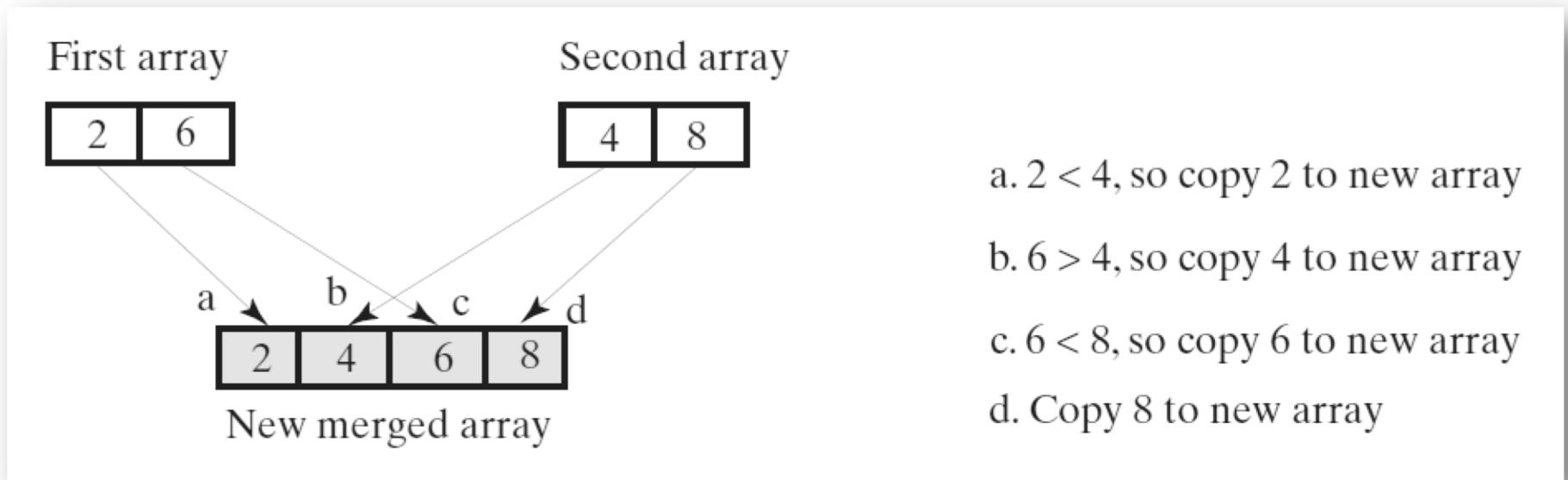
Recursive Merge Sort

- Be careful to allocate the temporary array only once.

```
public static <T extends Comparable<? super T>>
    void mergeSort(T[] a, int first, int last)
{
    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    T[] tempArray = (T[])new Comparable<?>[a.length]; // Unchecked cast
    mergeSort(a, tempArray, first, last);
} // end mergeSort
```

Efficiency of Merge Sort

- FIGURE 9-4 A worst-case merge of two sorted arrays.
- Efficiency is $O(n \log n)$.



Iterative Merge Sort

- Less simple than recursive version.
 - Need to control the merges.
- Will be more efficient of both time and space.
 - But, trickier to code without error.

Iterative Merge Sort

- Starts at beginning of array
 - Merges pairs of individual entries to form two-entry subarrays
 - Returns to the beginning of array and merges pairs of the two-entry subarrays to form four-entry subarrays
 - And so on
- After merging all pairs of subarrays of a particular length, might have entries left over.

Merge Sort in the Java Class Library

- Class **Arrays** in the package **java.util** defines versions of a static method **sort**

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

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

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 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

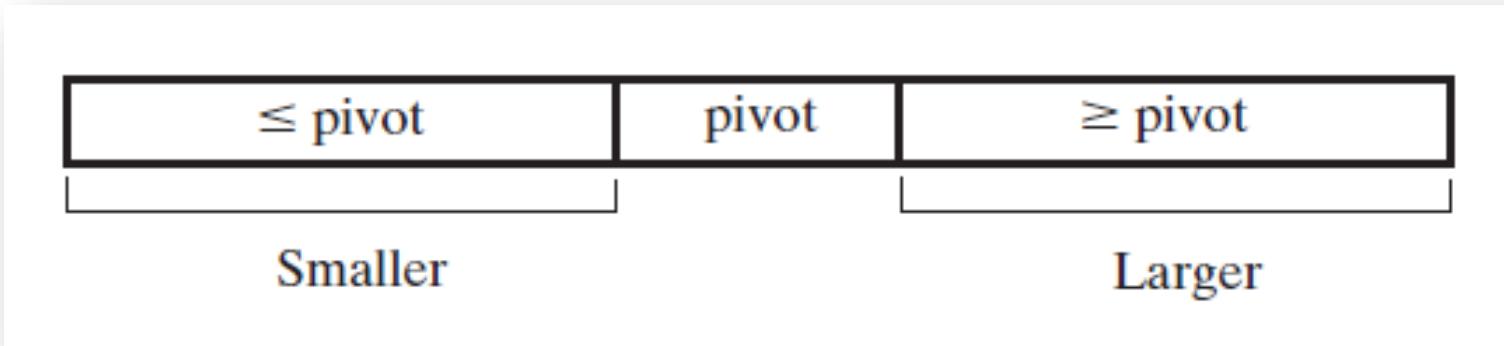
Quick Sort

- Algorithm that describes our sorting strategy:

```
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
}
```

Quick Sort

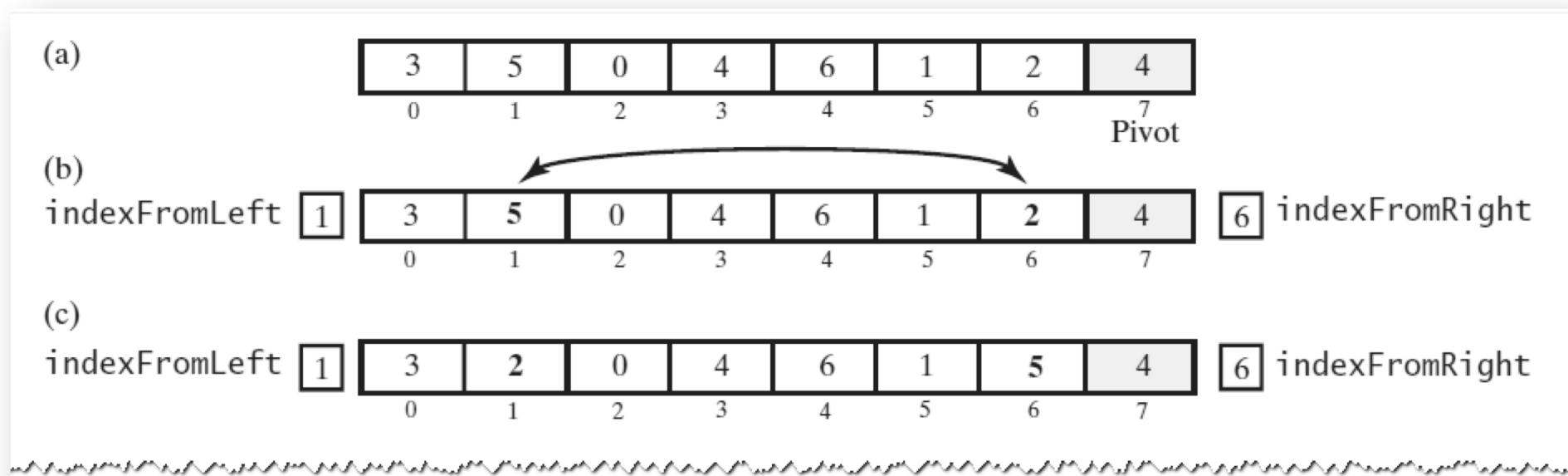
- FIGURE 9-5 A partition of an array during a quick sort



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

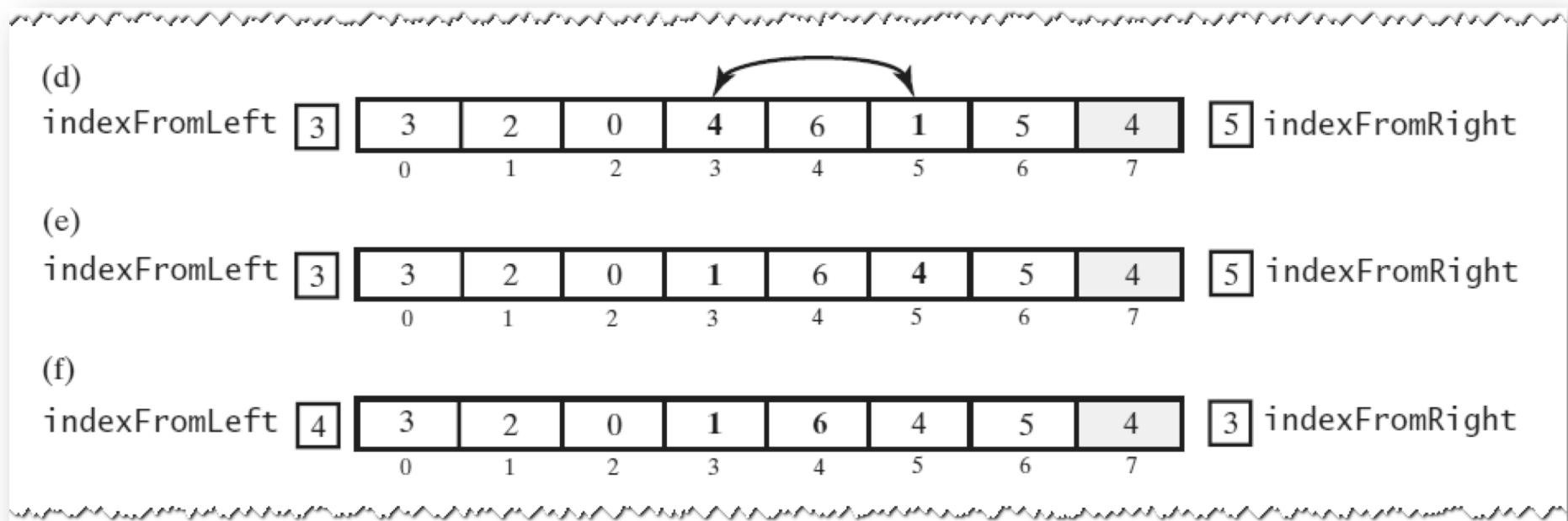
Creating the Partition

- FIGURE 9-6 A partitioning strategy for quick sort



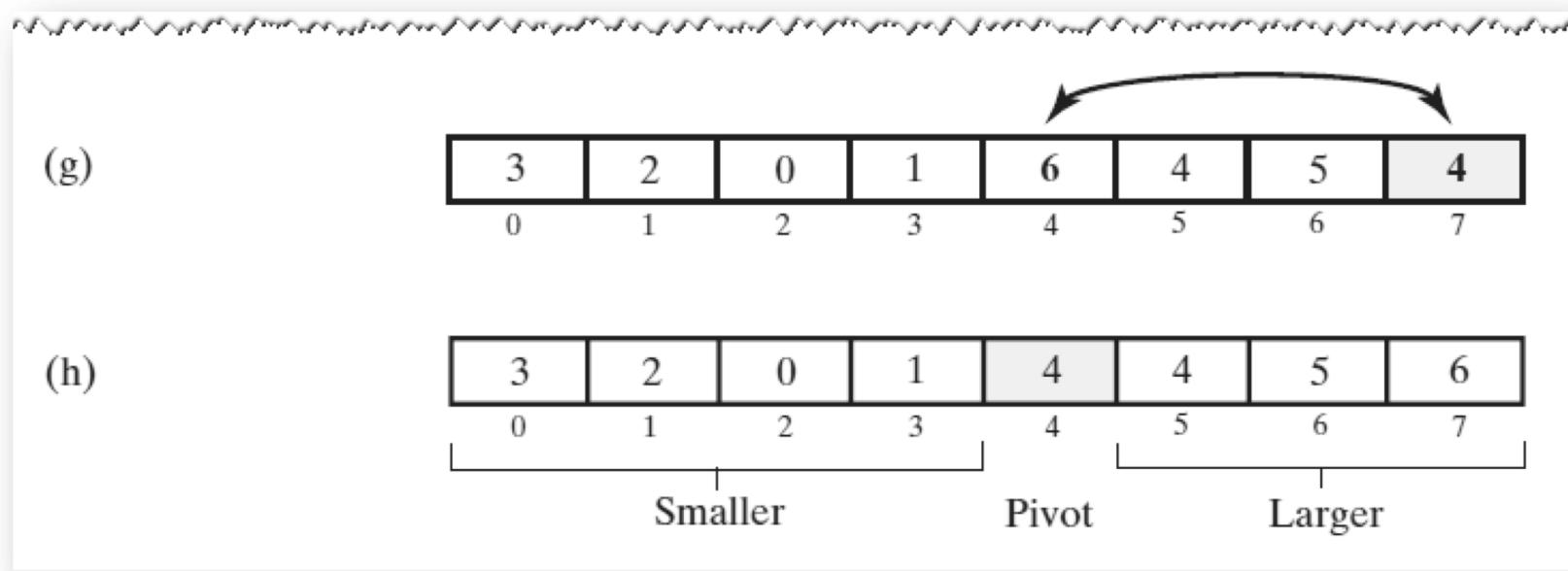
Creating the Partition

- FIGURE 9-6 A partitioning strategy for quick sort



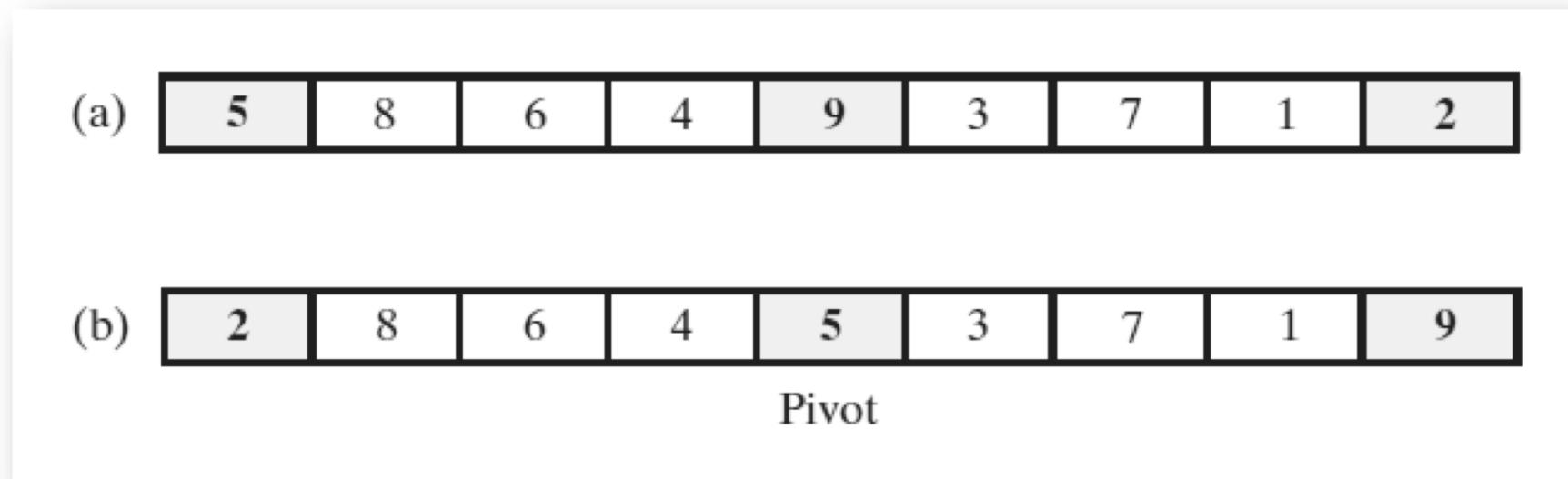
Creating the Partition

- FIGURE 9-6 A partitioning strategy for quick sort



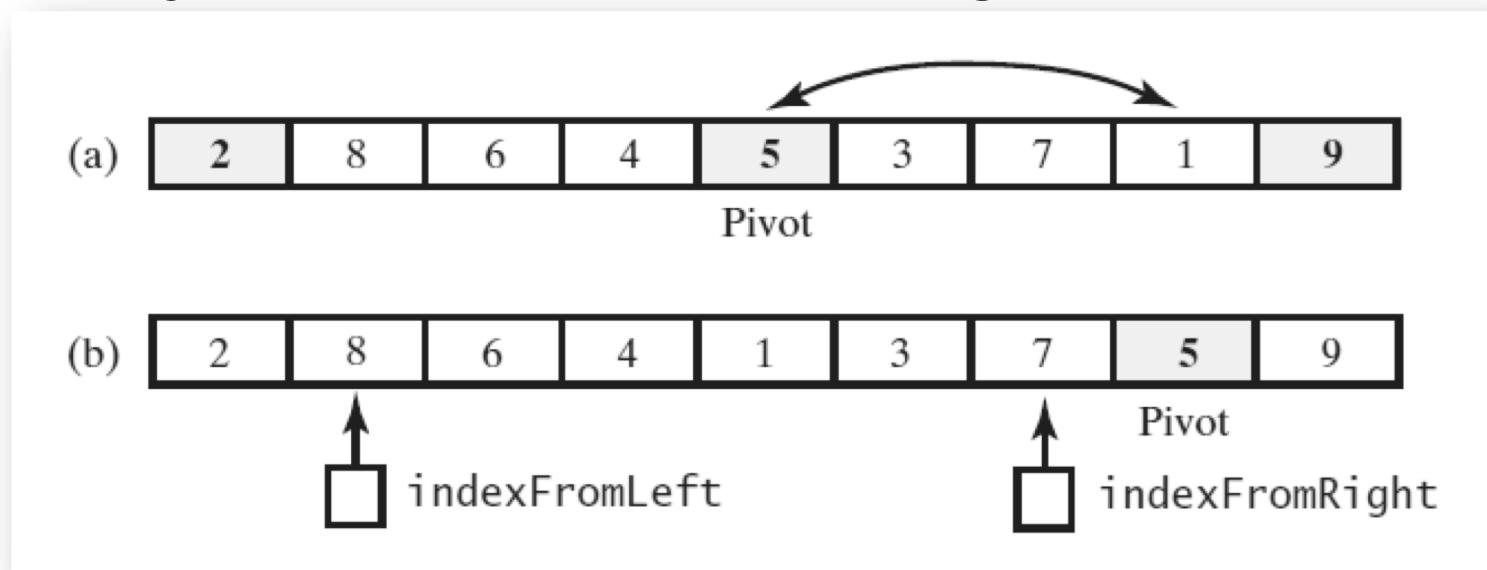
Creating the Partition

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



Adjusting the Partition Algorithm

- 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



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.
// 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
```

Adjusting the Partition Algorithm

```
// 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)
```

Adjusting the Partition Algorithm

```
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)
```

Adjusting the Partition Algorithm

```
//      a[indexFromRight] <= pivotValue
if (indexFromLeft < indexFromRight)
{
    Exchange a[indexFromLeft] and a[indexFromRight]
    indexFromLeft++
    indexFromRight--

}
else
    done = true
}

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

- Above method implements quick sort.

```
/** Sorts an array into ascending order. Uses quick sort with
 * median-of-three pivot selection for arrays of at least
 * MIN_SIZE entries, and uses insertion sort for smaller arrays. */
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
```

Quick Sort in the Java Class Library

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

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

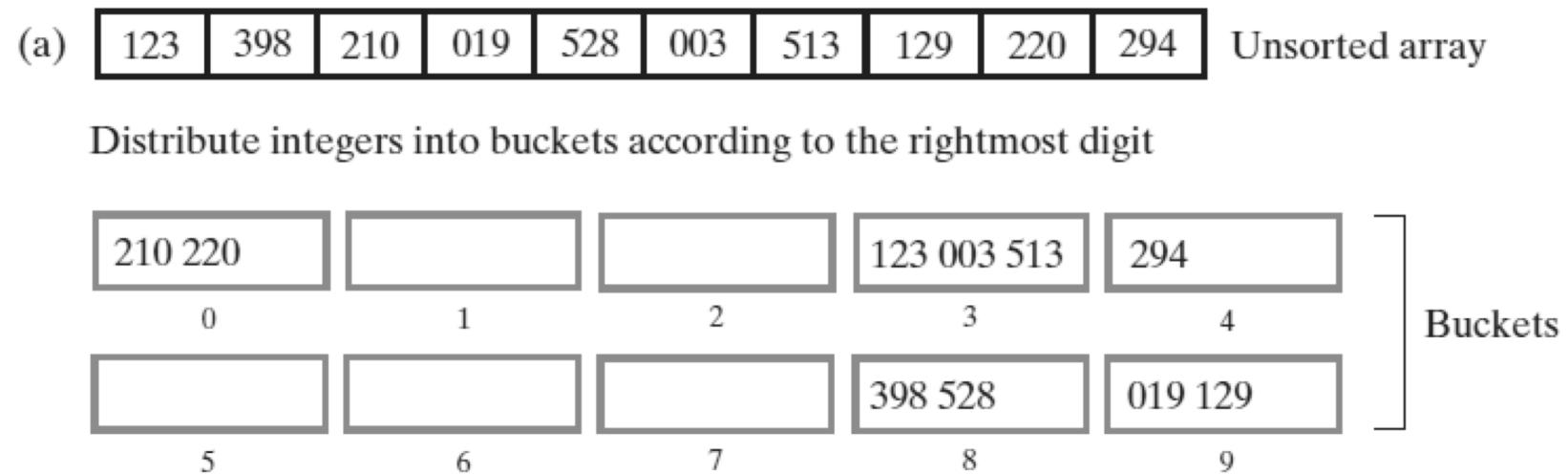
```
public static void sort(type[] a, int first, int after)
```

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.

Radix Sort

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



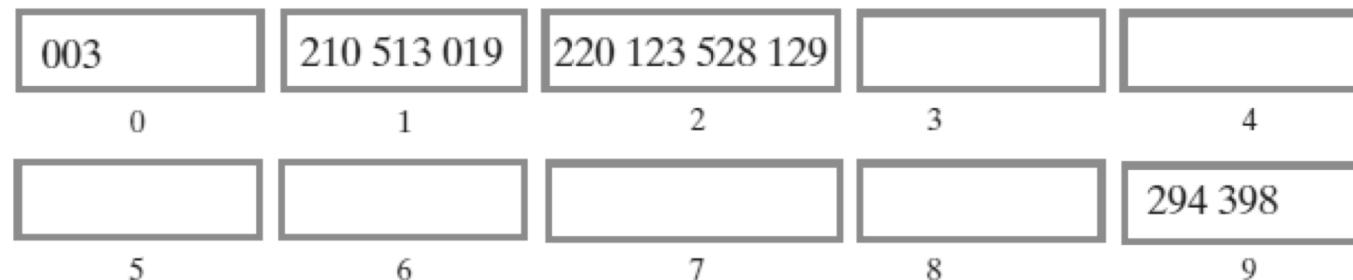
Radix Sort

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

(b)

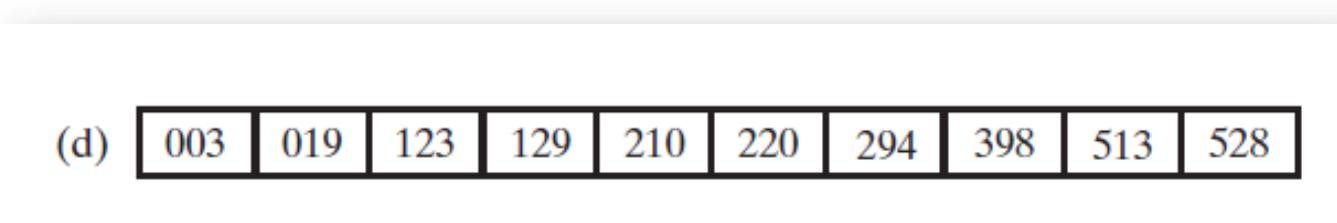
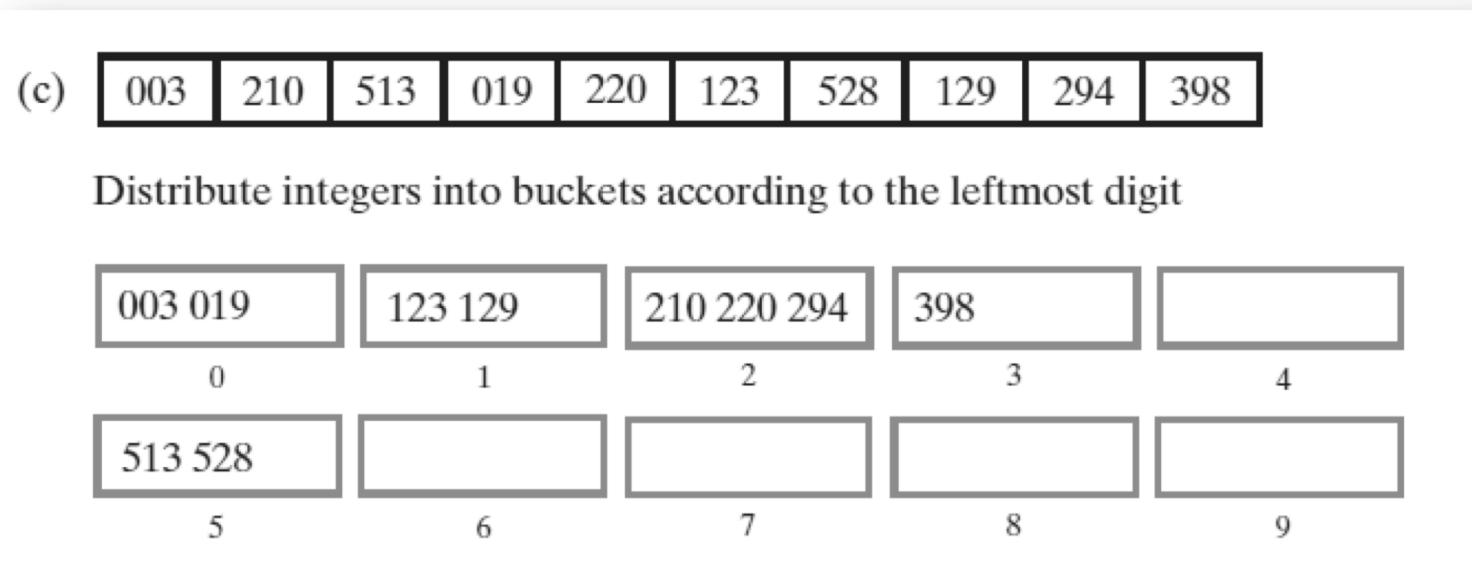
210	220	123	003	513	294	398	528	019	129
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Distribute integers into buckets according to the middle digit



Radix Sort

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



Pseudocode for Radix Sort

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

```
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
}
```

Comparing the Algorithms

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

	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	$O(n^2)$	$O(n^2)$	$O(n^2)$