Sorting Algorithms and Their Efficiency

Chapter 11

Basic Sorting Algorithms

- Sorting:
  - Organize a collection of data into either ascending or descending order
- Internal sort
  - Collection of data fits in memory
- External sort
  - Collection of data does *not* all fit in memory
  - Must reside on secondary storage

Basic Sorting Algorithms

- The Selection Sort
- The Insertion Sort
- The Bubble Sort

The Selection Sort

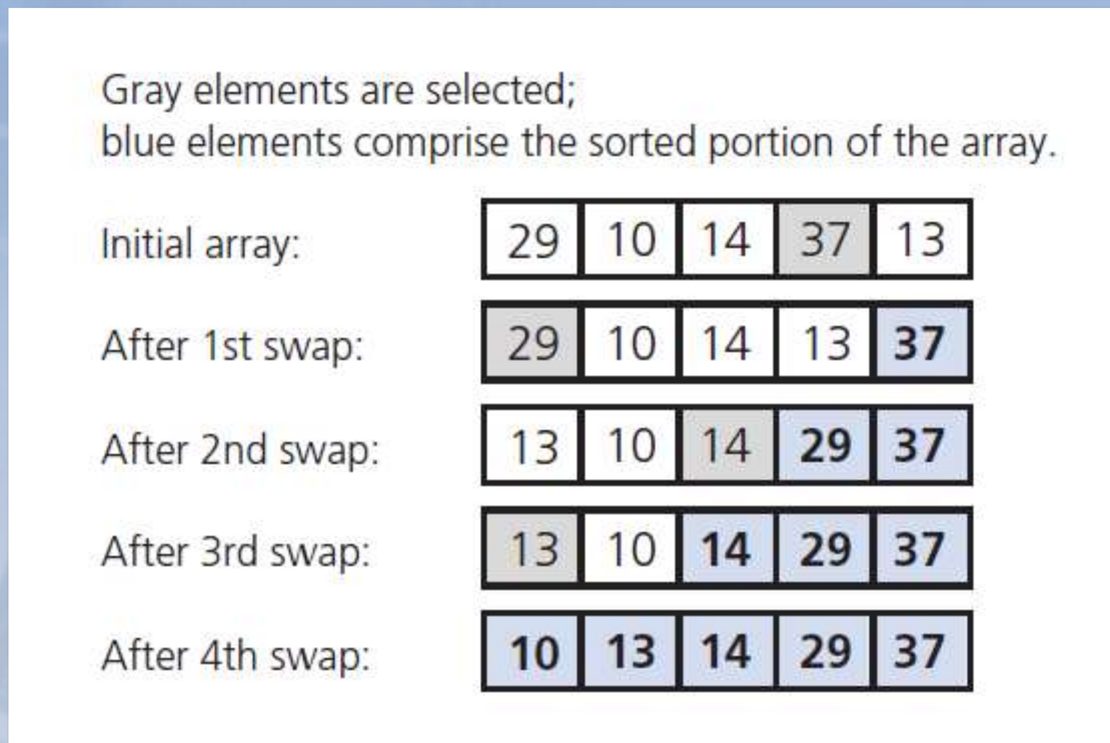

FIGURE 11-1 A selection sort of an array of f ive integers

The Selection Sort

```
/** Finds the largest item in an array.
 @pre  The size of the array is >= 1.
 @post  The arguments are unchanged.
 @param theArray  The given array.
 @param size  The number of elements in theArray.
 @return  The index of the largest entry in the array. */
template <class ItemType>
int findIndexOfLargest(const ItemType theArray[], int size);

/** Sorts  the items in an array into ascending order.
 @pre  None.
 @post  The array is sorted into ascending order; the size of the array
    is unchanged.
 @param theArray  The array to sort.
 @param n  The size of theArray. */
template <class ItemType>
void selectionSort(ItemType theArray[], int n)
{
   // last = index of the last item in the subarray of items yet
   //        to be sorted;
   // largest = index of the largest item found
```

LISTING 11-1 An implementation of the selection sort

The Selection Sort

```
    for (int last = n - 1; last >= 1; last--)
    {
       // At this point, theArray[last+1..n-1] is sorted, and its
       // entries are greater than those in theArray[0..last].
       // Select the largest entry in theArray[0..last]
       int largest = findIndexOfLargest(theArray, last+1);

       // Swap the largest entry, theArray[largest], with
       // theArray[last]
       std::swap(theArray[largest], theArray[last]);
    }  // end for
}  // end selectionSort

template <class ItemType>
```

LISTING 11-1 An implementation of the selection sort

The Selection Sort

```cpp
template <class ItemType>
int findIndexOfLargest(const ItemType theArray[], int size)
{
   int indexSoFar = 0; // Index of largest entry found so far
   for (int currentIndex = 1; currentIndex < size; currentIndex++)
   {
      // At this point, theArray[indexSoFar] >= all entries in
      // theArray[0..currentIndex - 1]
      if (theArray[currentIndex] > theArray[indexSoFar])
         indexSoFar = currentIndex;
   }  // end for

   return indexSoFar; // Index of largest entry
}  // end findIndexOfLargest
```

LISTING 11-1 An implementation of the selection sort

The Selection Sort

- Analysis
  - Selection sort is $O(n^2)$
  - Appropriate only for small *n,*
  - $O(n^2)$ grows rapidly
- Could be a good choice when
  - Data moves are costly,
  - But comparisons are not

The Bubble Sort

- Compares adjacent items
  - Exchanges them if out of order
  - Requires several passes over the data
- When ordering successive pairs
  - Largest item bubbles to end of the array

The Bubble Sort

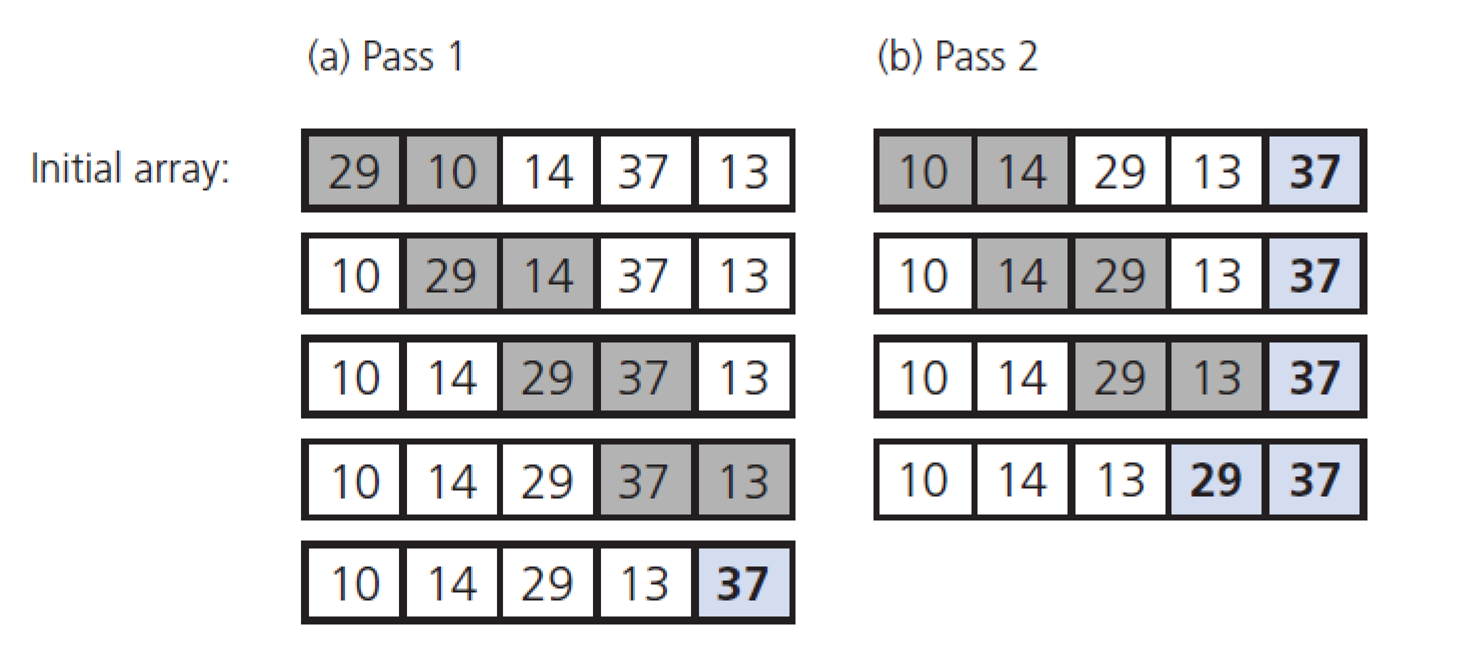

FIGURE 11-2 First two passes of a bubble sort of an array of five integers

The Bubble Sort

```
/** Sorts the items in an array into ascending order.
 @pre  None.
 @post  theArray is sorted into ascending order; n is unchanged.
 @param theArray  The given array.
 @param n  The size of theArray. */
template <class ItemType>
void bubbleSort(ItemType theArray[], int n)
{
   bool sorted = false;     // False when swaps occur
   int pass = 1;
   while (!sorted && (pass < n))
   {
      // At this point, theArray[n+1-pass..n-1] is sorted
      // and all of its entries are > the entries in theArray[0..n-pass]
      sorted = true;         // Assume sorted
      for (int index = 0; index < n - pass; index++)
      {
```

LISTING 11-2 An implementation of the bubble sort

The Bubble Sort

```
        for (int index = 0; index < n - pass; index++)
        {
           // At this point, all entries in theArray[0..index-1]
           // are <= theArray[index]
           int nextIndex = index + 1;
           if (theArray[index] > theArray[nextIndex])
           {
              // Exchange entries
              std::swap(theArray[index], theArray[nextIndex]);
              sorted = false; // Signal exchange
           }  // end if
        }  // end for
        // Assertion: theArray[0..n-pass-1] < theArray[n-pass]

        pass++;
     }  // end while
  }  // end bubbleSort
```

LISTING 11-2 An implementation of the bubble sort

The Bubble Sort

- Analysis
  - Worst case $O(n^2)$
  - Best case (array already in order) is $O(n)$

The Insertion Sort

- Take each item from unsorted region
  - Insert it into correct order in sorted region

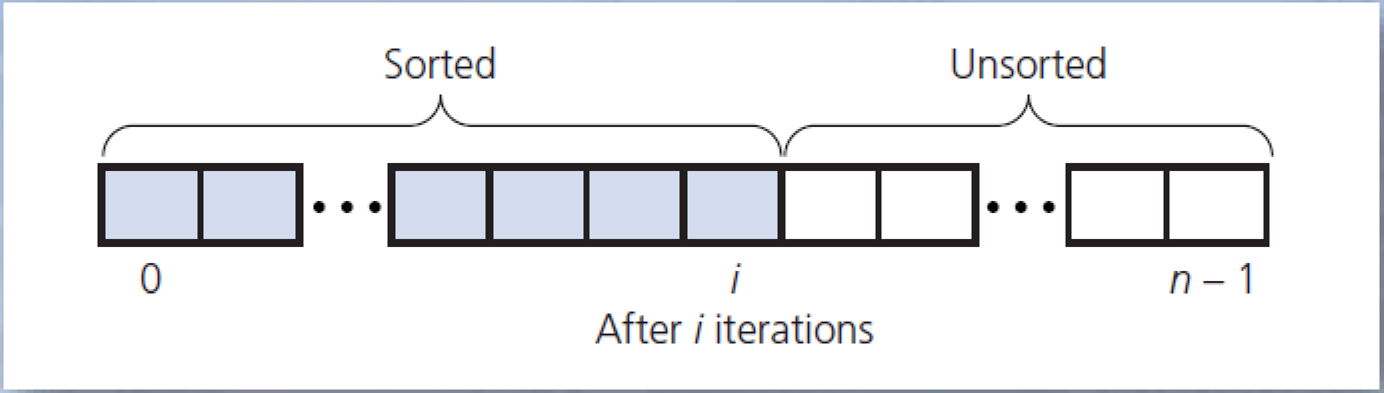

FIGURE 11-3 An insertion sort partitions the array into two regions

The Insertion Sort

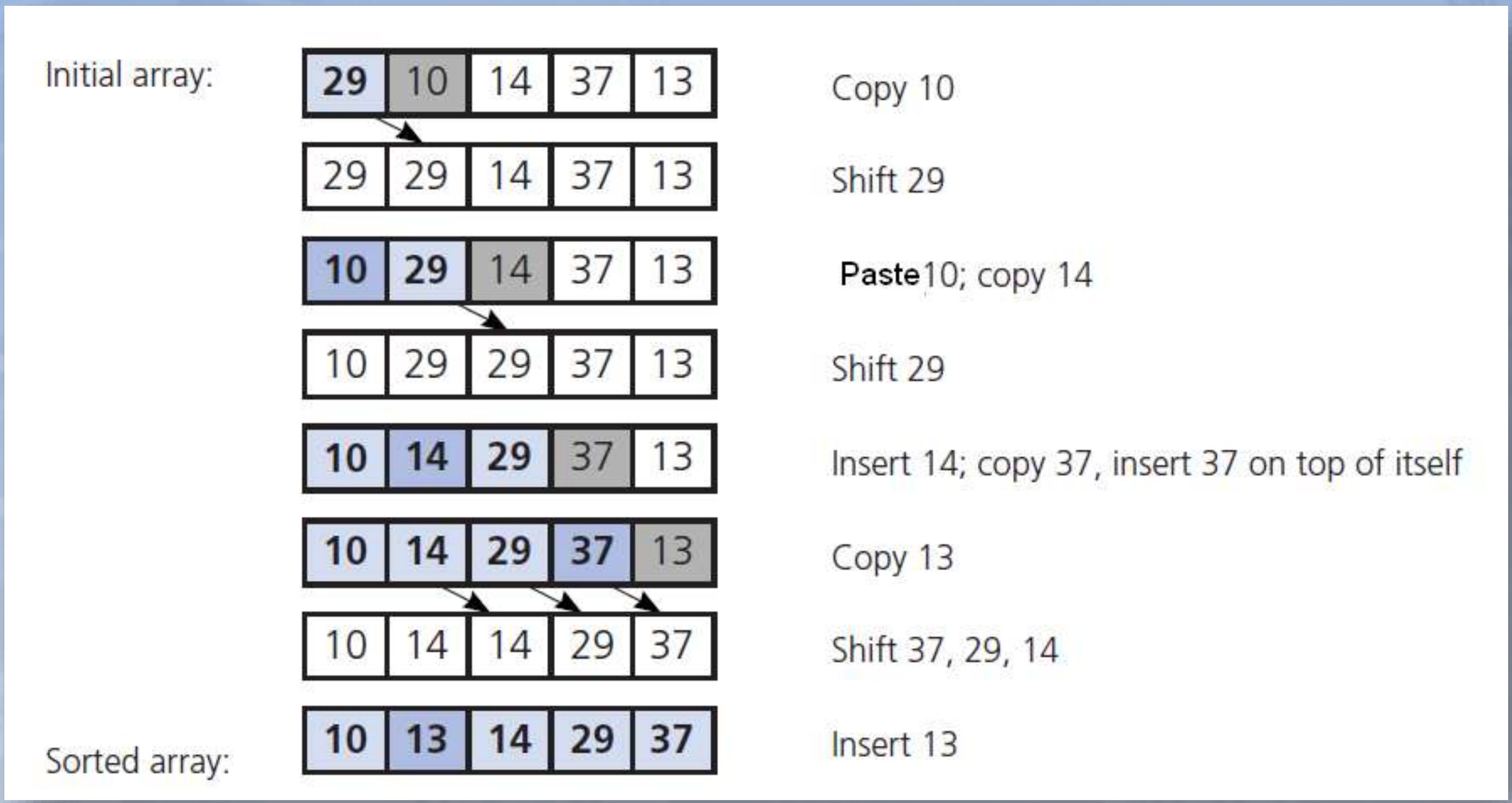

FIGURE 11-4 An insertion sort of an array of five integers

The Insertion Sort

```
/** Sorts the items in an array into ascending order.
 @pre  None.
 @post  theArray is sorted into ascending order; n is unchanged.
 @param theArray  The given array.
 @param n  The size of theArray. */
template<class ItemType>
void insertionSort(ItemType theArray[], int n)
{
   // unsorted = first index of the unsorted region,
   // loc = index of insertion in the sorted region,
   // nextItem = next item in the unsorted region.
   // Initially,  sorted region is theArray[0],
   //             unsorted region is theArray[1..n-1].
   // In general, sorted region is theArray[0..unsorted-1],
   //             unsorted region theArray[unsorted..n-1]
```

LISTING 11-3 An implementation of the insertion sort

The Insertion Sort

```
    for (int unsorted = 1; unsorted < n; unsorted++)
    {
        // At this point, theArray[0..unsorted-1] is sorted.
        // Find the right position (loc) in theArray[0..unsorted]
        // for theArray[unsorted], which is the first entry in the
        // unsorted region; shift, if necessary, to make room
        ItemType nextItem = theArray[unsorted];
        int loc = unsorted;
        while ((loc > 0) && (theArray[loc - 1] > nextItem))
        {
            // Shift theArray[loc - 1] to the right
            theArray[loc] = theArray[loc - 1];
            loc--;
        }  // end while
        // At this point, theArray[loc] is where nextItem belongs
        theArray[loc] = nextItem; // Insert nextItem into sorted region
    }  // end for
}  // end insertionSort
```

LISTING 11-3 An implementation of the insertion sort

The Insertion Sort

- Analysis
  - Worst case $O(n^2)$
  - Best case (array already in order) is $O(n)$
- Appropriate for small ($n < 25$) arrays
- Unsuitable for large arrays
  - Unless already sorted

Faster Sorting Algorithms

- The Merge Sort
- The Quick Sort
- The Radix Sort

Merge Sort

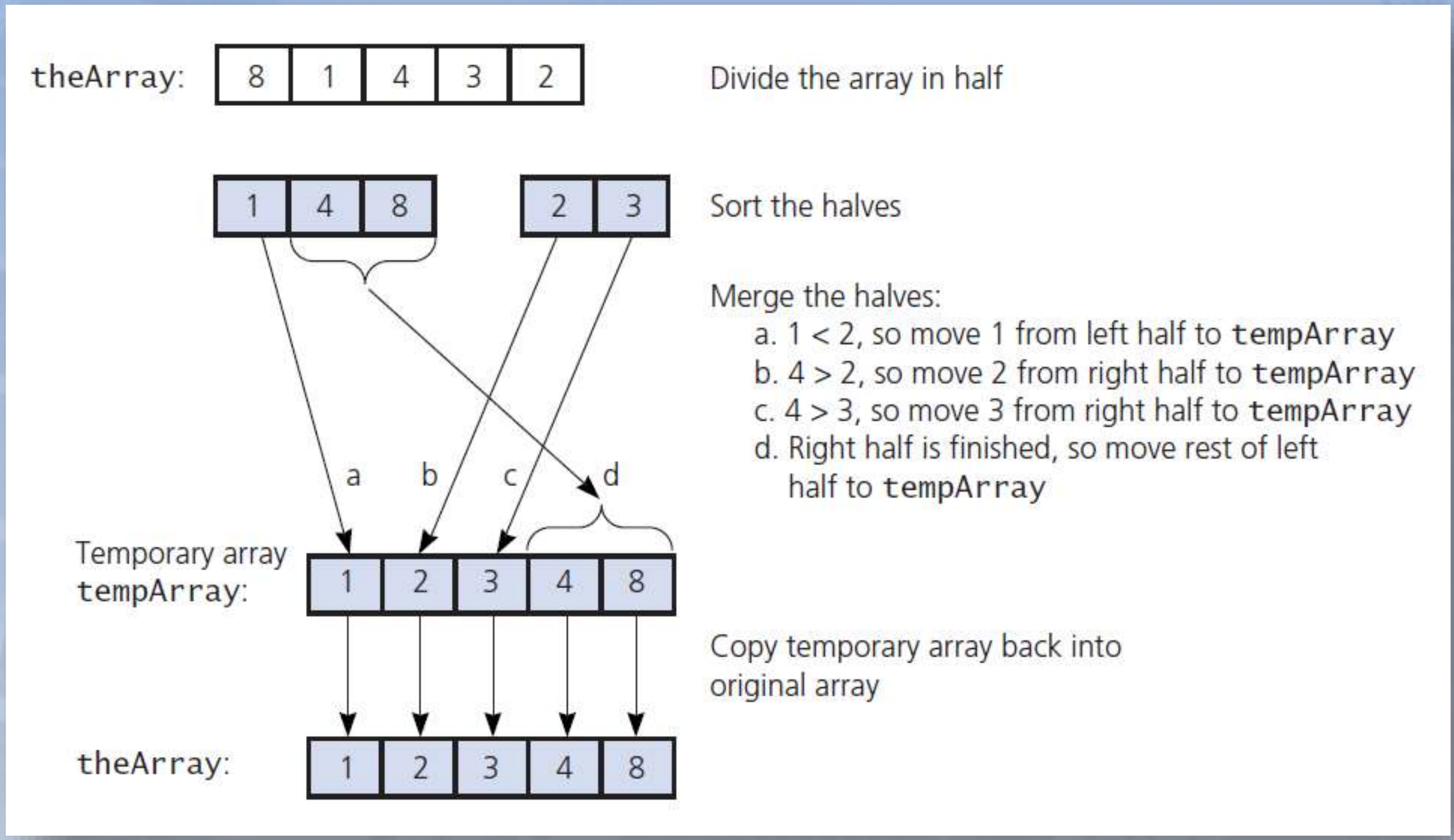

FIGURE 11-5 A merge sort with an auxiliary temporary array

Merge Sort

```
// Sorts theArray[first..last] by
//    1. Sorting the first half of the array
//    2. Sorting the second half of the array
//    3. Merging the two sorted halves
mergeSort(theArray: ItemArray, first: integer, last: integer)
{
   if (first < last)
   {
      mid = (first + last) / 2        // Get midpoint

      // Sort theArray[first..mid]
      mergeSort(theArray, first, mid)

      // Sort theArray[mid+1..last]
      mergeSort(theArray, mid + 1, last)

      // Merge sorted halves theArray[first..mid] and theArray[mid+1..last]
      merge(theArray, first, mid, last)
   }
   // If first >= last, there is nothing to do
}
```

Pseudocode for the merge sort

Merge Sort

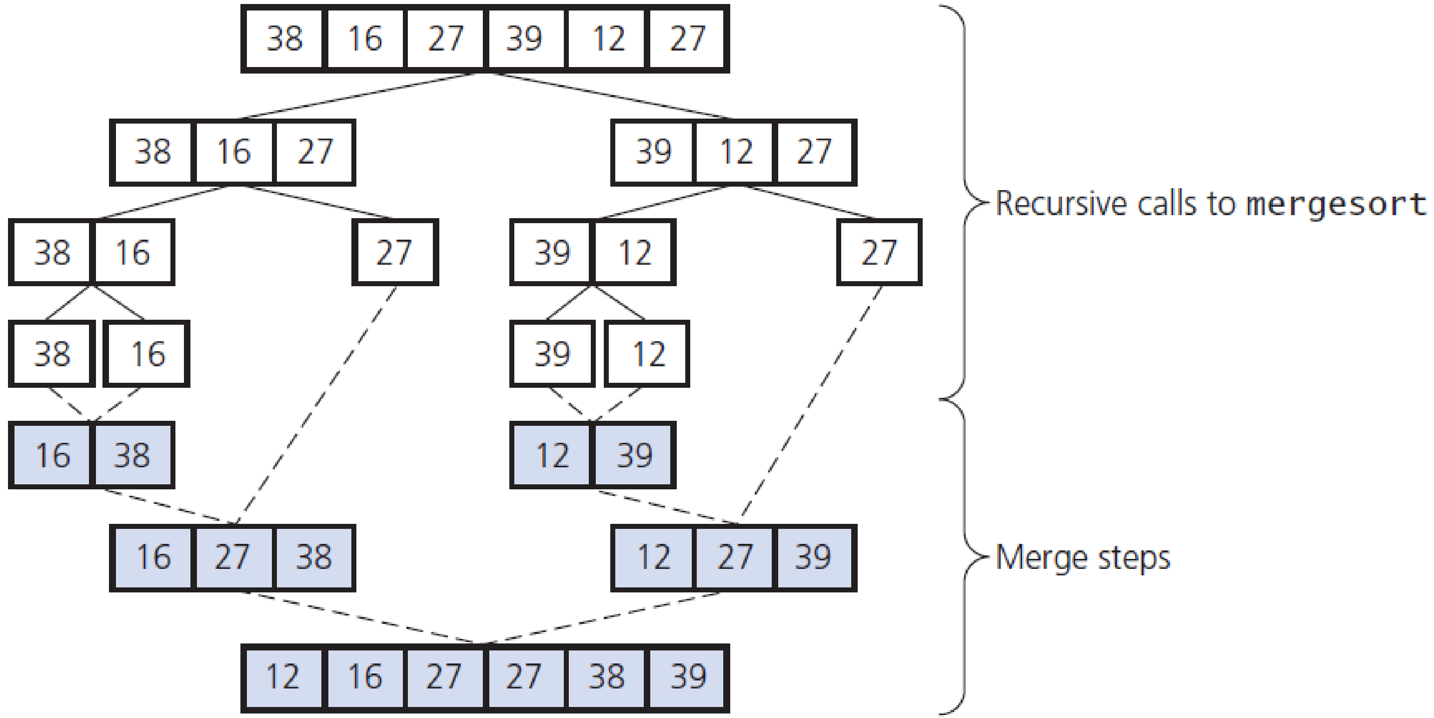

FIGURE 11-6 A merge sort of an array of six integers

Merge Sort

```
const int MAX_SIZE = maximum-number-of-items-in-array;

/** Merges two sorted array segments theArray[first..mid] and
    theArray[mid+1..last] into one sorted array.
 @pre  first <= mid <= last. The subarrays theArray[first..mid] and
    theArray[mid+1..last] are each sorted in increasing order.
 @post  theArray[first..last] is sorted.
 @param theArray  The given array.
 @param first  The index of the beginning of the first segment in
    theArray.
 @param mid  The index of the end of the first segment in theArray;
    mid + 1 marks the beginning of the second segment.
 @param last  The index of the last element in the second segment in
    theArray.
```

LISTING 11-4 An implementation of merge and mergeSort

Merge Sort

```
 @note  This function merges the two subarrays into a temporary
    array and copies the result into the original array theArray. */
template <class ItemType>
void merge(ItemType theArray[], int first, int mid, int last)
{
   ItemType tempArray[MAX_SIZE]; // Temporary array

   // Initialize the local indices to indicate the subarrays
   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

   // While both subarrays are not empty, copy the
   // smaller item into the temporary array
   int index = first1;           // Next available location in tempArray
   while ((first1 <= last1) && (first2 <= last2))
   {
```

LISTING 11-4 An implementation of merge and mergeSort

Merge Sort

```
while ((first1 <= last1) && (first2 <= last2))
{
   // At this point, tempArray[first..index-1] is in order
   if (theArray[first1] <= theArray[first2])
   {
      tempArray[index] = theArray[first1];
      first1++;
   }
   else
   {
      tempArray[index] = theArray[first2];
      first2++;
   }   // end if
   index++;
}  // end while
// Finish off the first subarray, if necessary
while (first1 <= last1)
```

LISTING 11-4 An implementation of merge and mergeSort

Merge Sort

```
   while (first1 <= last1)
   {
      // At this point, tempArray[first..index-1] is in order
      tempArray[index] = theArray[first1];
      first1++;
      index++;
   }  // end while
   // Finish off the second subarray, if necessary
   while (first2 <= last2)
   {
      // At this point, tempArray[first..index-1] is in order
      tempArray[index] = theArray[first2];
      first2++;
      index++;
   }  // end for

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

LISTING 11-4 An implementation of merge and mergeSort

Merge Sort

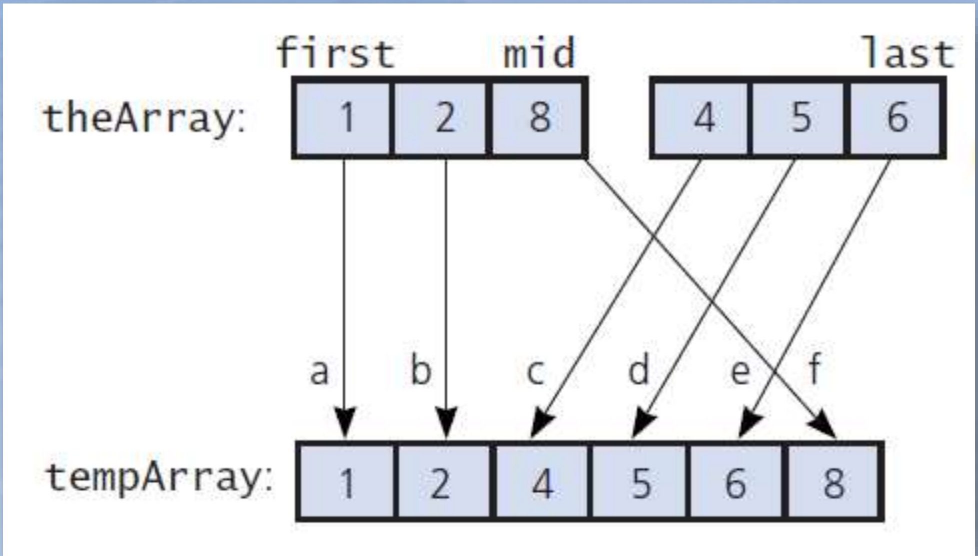

FIGURE 11-7 A worst-case instance of the merge step in a merge sort

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`

Merge Sort

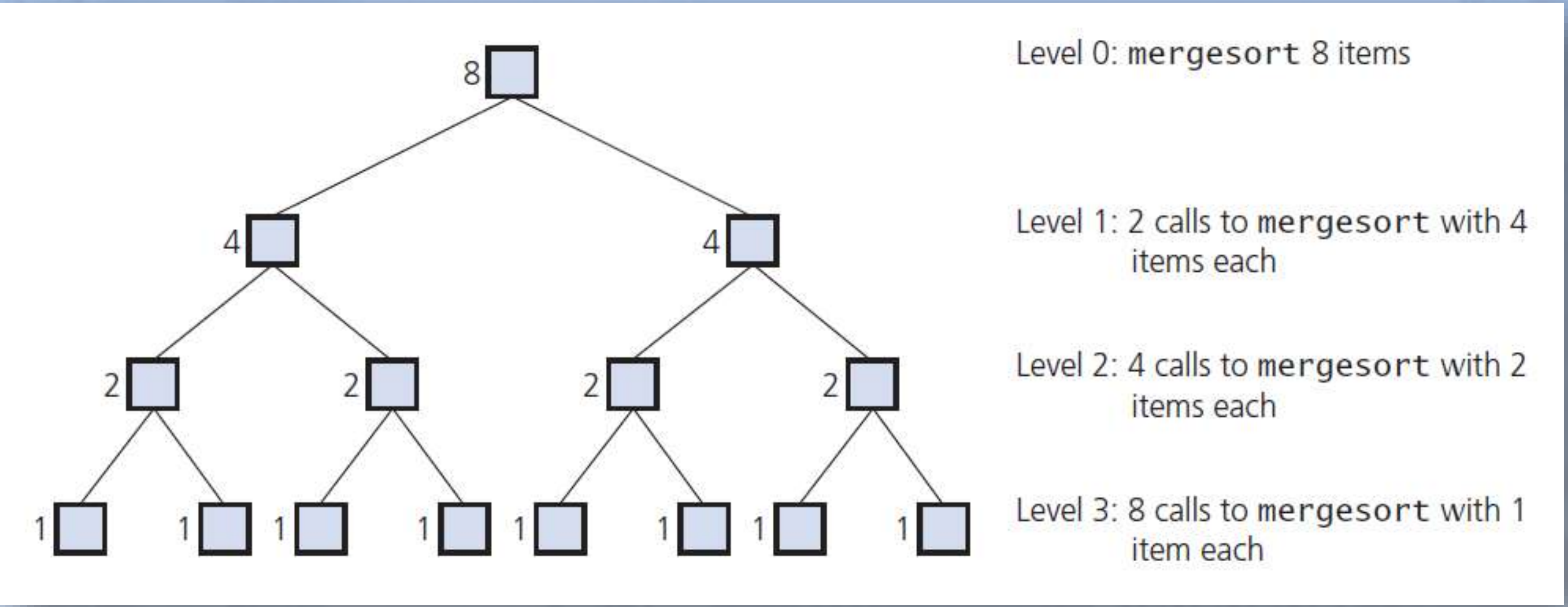

FIGURE 11-8 Levels of recursive calls to mergeSort, given an array of eight items

The Quick Sort

- Another divide-and-conquer algorithm
- Partitions an array into items that are
  - Less than or equal to the pivot and
  - Those that are greater than or equal to the pivot
- Partitioning places pivot in its correct position within the array
  - Place chosen pivot in theArray[last] before partitioning

The Quick Sort

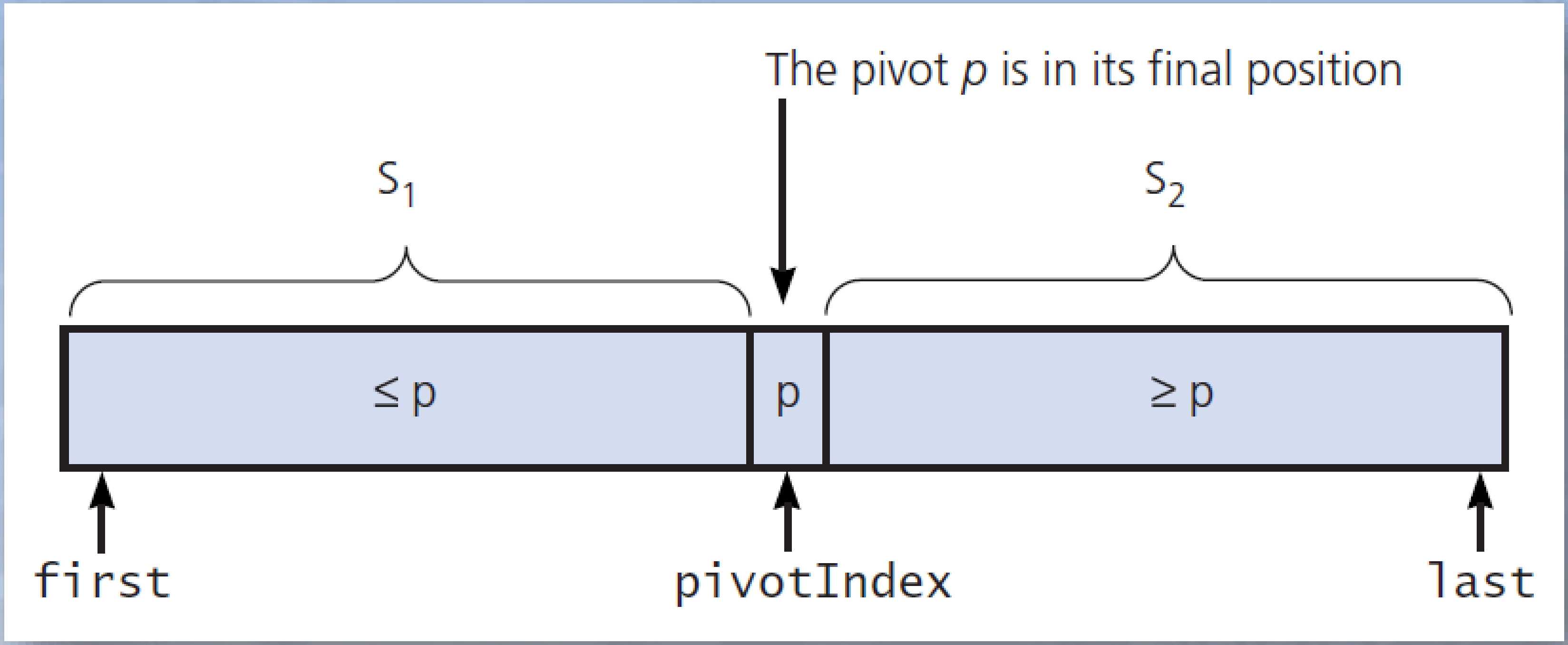

FIGURE 11-9 A partition about a pivot

The Quick Sort

```
// Sorts theArray[first..last].
quickSort(theArray: ItemArray, first: integer, last: integer): void
{
  if (first < last)
  {
    Choose a pivot item p from theArray[first..last]
    Partition the items of theArray[first..last] about p
    // The partition is theArray[first..pivotIndex..last]

    quickSort(theArray, first, pivotIndex - 1) // Sort S1

    quickSort(theArray, pivotIndex + 1, last)  // Sort S2
  }
  // If first >= last, there is nothing to do
}
```

First draft of pseudocode for the quick sort algorithm

The Quick Sort

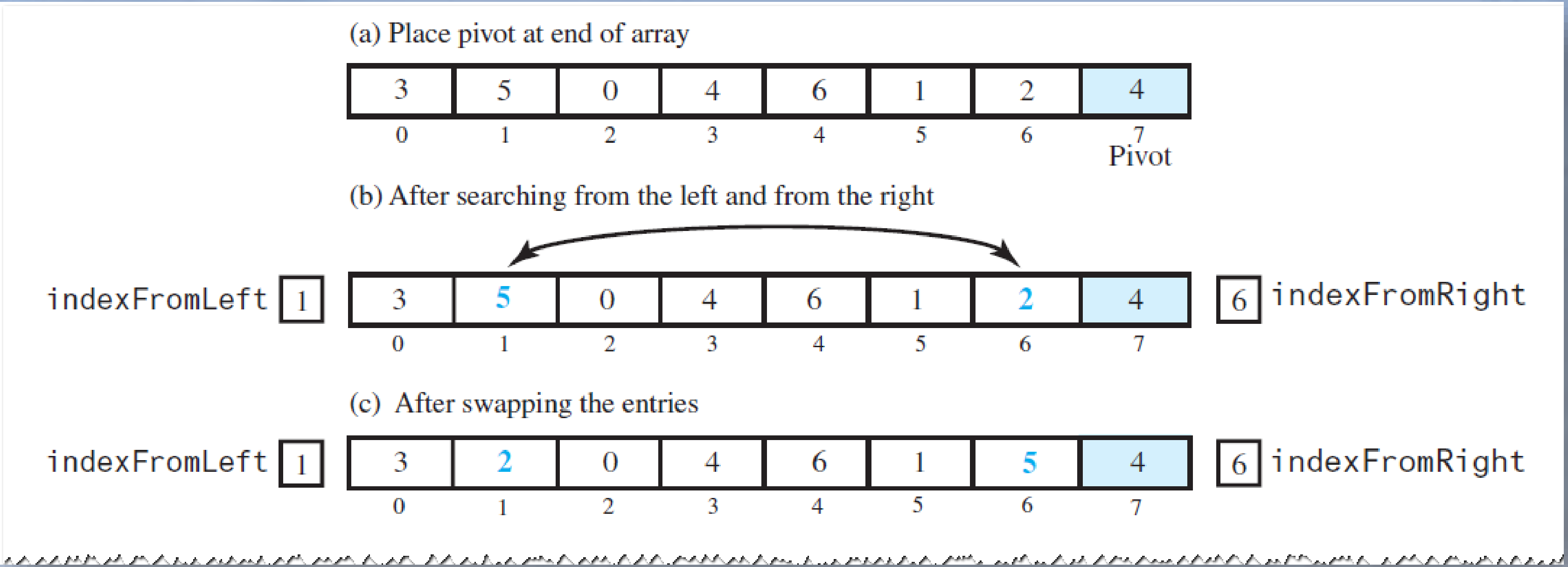

FIGURE 11-10 A partitioning of an array during a quick sort

The Quick Sort

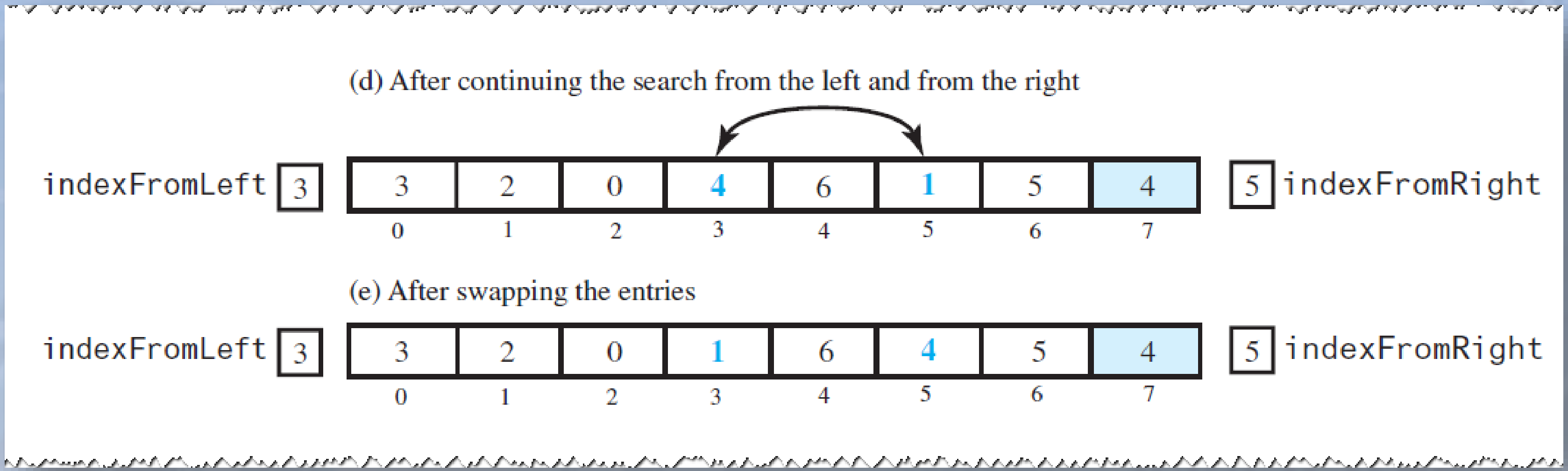

FIGURE 11-10 A partitioning of an array during a quick sort

The Quick Sort

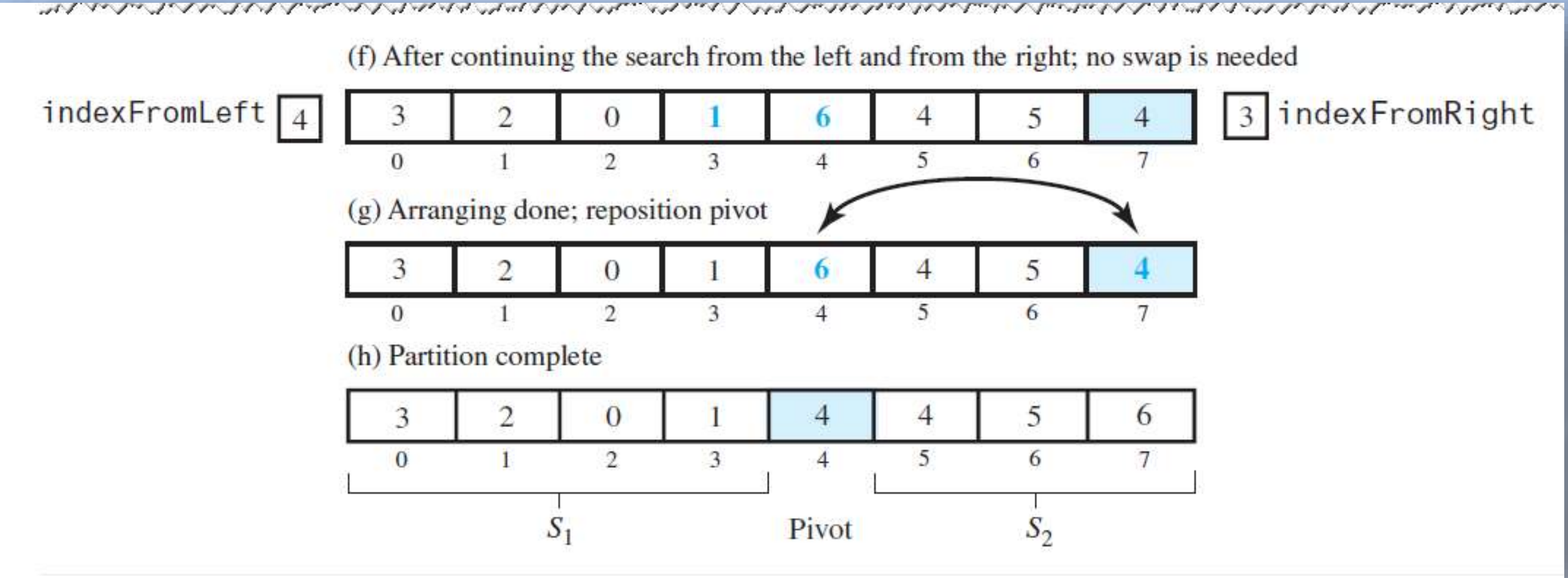

FIGURE 11-10 A partitioning of an array during a quick sort

The Quick Sort

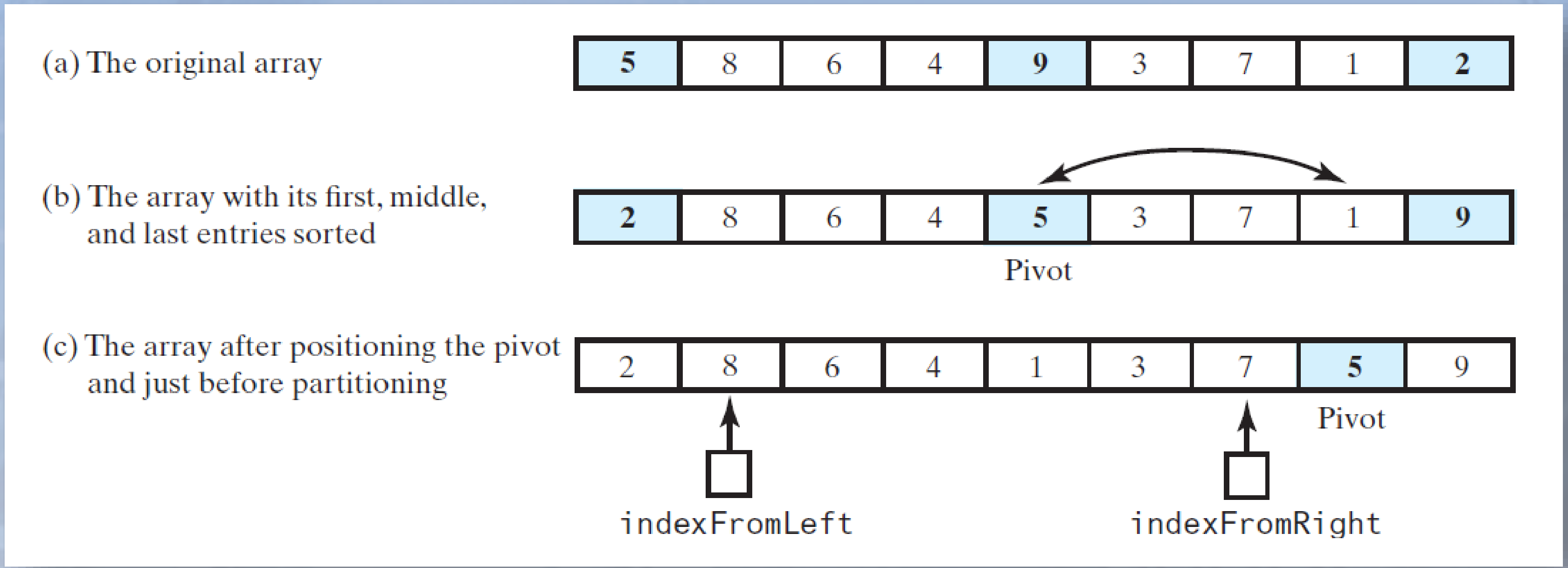

FIGURE 11-11 Median-of-three pivot selection

The Quick Sort

```
// Arranges the first, middle, and last entries in an array into ascending order.
sortFirstMiddleLast(theArray: ItemArray, first: integer, mid: integer,
                    last: integer): void
{
   if (theArray[first] > theArray[mid])
      Interchange theArray[first] and theArray[mid]

   if (theArray[mid] > theArray[last])
      Interchange theArray[mid] and theArray[last]

   if (theArray[first] > theArray[mid])
      Interchange theArray[first] and theArray[mid]
}
```

Adjusting the partition algorithm.

The Quick Sort

```
// Partitions theArray[first..last].
partition(theArray: ItemArray, first: integer, last: integer): integer
{
   // Choose pivot and reposition it
   mid = first + (last - first) / 2
   sortFirstMiddleLast(theArray, first, mid, last)
   Interchange theArray[mid] and theArray[last - 1]
   pivotIndex = last - 1
   pivot = theArray[pivotIndex]

   // Determine the regions S1 and S2
   indexFromLeft = first + 1
   indexFromRight = last - 2

   done = false
   while (not done)
   {
      // Locate first entry on left that is ≥ pivot
```

Pseudocode describes the partitioning algorithm for an array of at least four entries

The Quick Sort

```
done = false
while (not done)
{
   // Locate first entry on left that is ≥ pivot
   while (theArray[indexFromLeft] < pivot)
      indexFromLeft = indexFromLeft + 1

   // Locate first entry on right that is ≤ pivot
   while (theArray[indexFromRight] > pivot)
      indexFromRight = indexFromRight - 1

   if (indexFromLeft < indexFromRight)
   {
      Interchange theArray[indexFromLeft] and theArray[indexFromRight]
      indexFromLeft = indexFromLeft + 1
      indexFromRight = indexFromRight - 1
   }
   else
      done = true
}
```

Pseudocode describes the partitioning algorithm for an array of at least four entries

The Quick Sort

```
            indexFromRight = indexFromRight - 1
        }
        else
            done = true
    }
    // Place pivot in proper position between S1 and S2, and mark its new location
    Interchange theArray[pivotIndex] and theArray[indexFromLeft]
    pivotIndex = indexFromLeft

    return pivotIndex
}
```

Pseudocode describes the partitioning algorithm for an array of at least four entries

The Quick Sort

```
/** Sorts an array into ascending order. Uses the quick sort with
    median-of-three pivot selection for arrays of at least MIN_SIZE
    entries, and uses the insertion sort for other arrays.
 @pre  theArray[first..last] is an array.
 @post  theArray[first..last] is sorted.
 @param theArray  The given array.
 @param first  The index of the first element to consider in theArray.
 @param last  The index of the last element to consider in theArray. */
template <class ItemType>
void quickSort(ItemType theArray[], int first, int last)
{
   if ((last - first + 1) < MIN_SIZE)
   {
      insertionSort(theArray, first, last);
   }
```

LISTING 11-5 A function that performs a quick sort

The Quick Sort

```
   }
   else
   {
      // Create the partition: S1 | Pivot | S2
      int pivotIndex = partition(theArray, first, last);

      // Sort subarrays S1 and S2
      quickSort(theArray, first, pivotIndex - 1);
      quickSort(theArray, pivotIndex + 1, last);
   }  // end if
}  // end quickSort
```

LISTING 11-5 A function that performs a quick sort

The Quick Sort

- Analysis
  - Partitioning is an O(n) task
  - There are either $\log_2 n$ or $1 + \log_2 n$ levels of recursive calls to quickSort.
- We conclude
  - Worst case O(n^2)
  - Average case O($n \log n$)

The Quick Sort

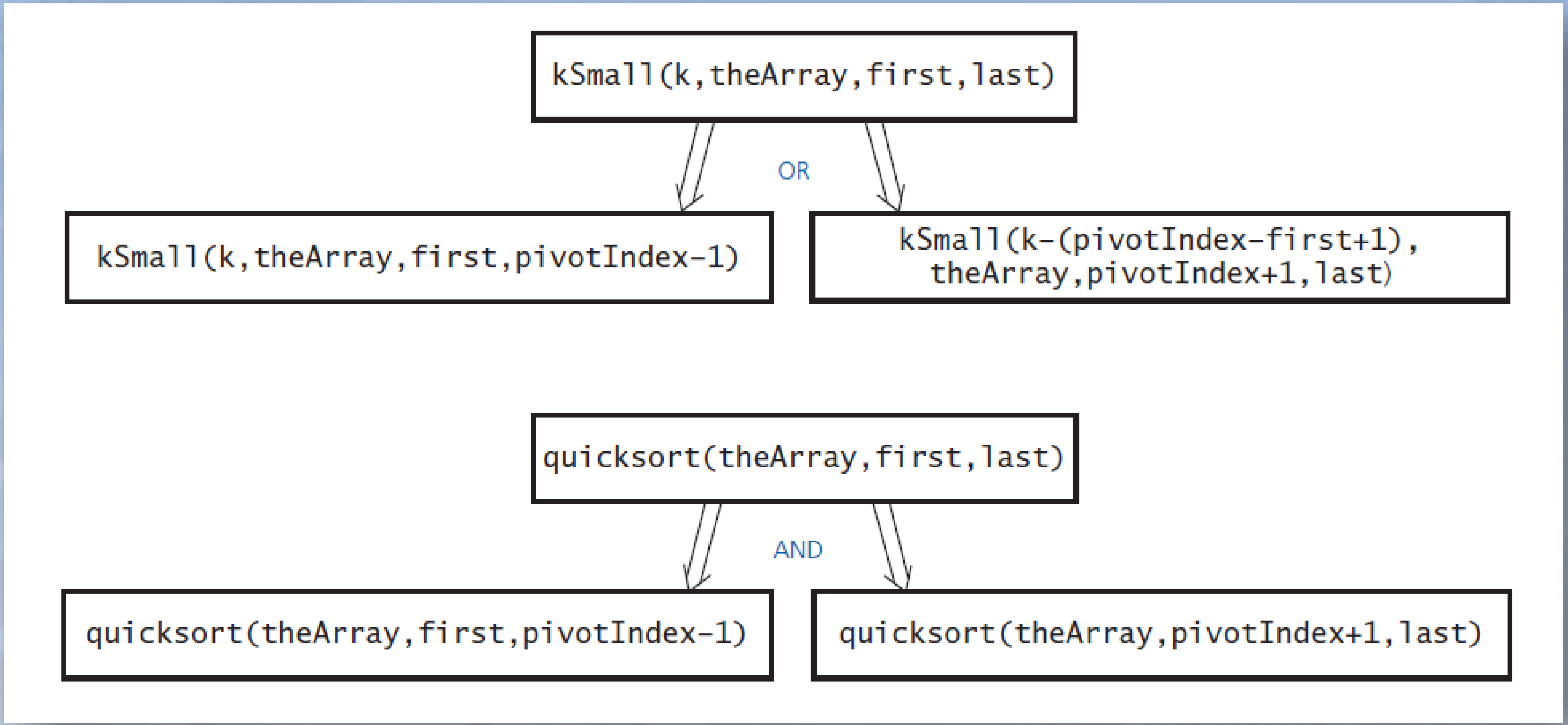

FIGURE 11-12 kSmall versus quickSort

The Radix Sort

- Different from other sorts
  - Does not compare entries in an array
- Begins by organizing data (say strings) according to least significant letters
  - Then combine the groups
- Next form groups using next least signficant letter

The Radix Sort

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

FIGURE 11-13 A radix sort of eight integers

The Radix Sort

```
// Sorts n d-digit integers in the array theArray.
radixSort(theArray: ItemArray, n: integer, d: integer): void
{
   for (j = d down to 1)
   {
      Initialize 10 groups to empty
      Initialize a counter for each group to 0
      for (i = 0 through n - 1)
      {
         k = jth digit of theArray[i]
         Place theArray[i] at the end of group k
         Increase kth counter by 1
      }
      Replace the items in theArray with all the items in group 0,
        followed by all the items in group 1, and so on.
   }
}
```

Pseudocode for algorithm for a radix sort of *n* decimal integers of *d* digits each:

The Radix Sort

- Analysis
  - Requires *n* moves each time it forms groups
  - *n* moves to combine again into one group
  - Performs these $2 \times n$ moves *d* times
  - Thus requires $2 \times n \times d$ moves
- Radix sort is of order O(*n*)
- More appropriate for chain of linked lists than for an array

A Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Merge sort	$n \times \log n$	$n \times \log n$
Quick sort	n^2	$n \times \log n$
Radix sort	n	n
Tree sort	n^2	$n \times \log n$
Heap sort	$n \times \log n$	$n \times \log n$

FIGURE 11-14 Approximate growth rates of time required for eight sorting algorithms

End

Chapter 11