```
        @param table The array being sorted
        @param nextPos The position of element to insert
        @param gap The gap between elements in the subarray
    */
    private static <T extends Comparable<T>> void insert(T[] table,
                                                         int nextPos, int gap) {
        T nextVal = table[nextPos];
        // Element to insert.
        // Shift all values > nextVal in subarray down by gap.
        while ((nextPos > gap - 1) && (nextVal.compareTo
                                       (table [nextPos - gap]) < 0)) {
            // First element not shifted.
            table[nextPos] = table[nextPos - gap];
            // Shift down.
            nextPos -= gap;
            // Check next position in subarray.
        }
        table[nextPos] = nextVal;
        // Insert nextVal.
    }
}
```

EXERCISES FOR SECTION 8.5

SELF-CHECK

1. Trace the execution of Shell sort on the following array. Show the array after all sorts when the gap is 5, the gap is 2, and after the final sort when the gap is 1. List the number of comparisons and exchanges required when the gap is 5, the gap is 2 and when the gap is 1. Compare this with the number of comparisons and exchanges that would be required for a regular insertion sort.

   40 35 80 75 60 90 70 65 50 22

2. For the example of Shell sort shown in this section, determine how many comparisons and exchanges are required to insert all the elements for each gap value. Compare this with the number of comparisons and exchanges that would be required for a regular insertion sort.

PROGRAMMING

1. Eliminate method `insert` in Listing 8.3 and write its code inside the **`for`** statement.

2. Add statements to trace the progress of Shell sort. Display each value of `gap`, and display the array contents after all subarrays for that `gap` value have been sorted.

8.6 Merge Sort

The next algorithm that we will consider is called *merge sort*. A *merge* is a common data processing operation that is performed on two sequences of data (or data files) with the following characteristics:

- Both sequences contain items with a common `compareTo` method.
- The objects in both sequences are ordered in accordance with this `compareTo` method (i.e., both sequences are sorted).

The result of the merge operation is to create a third sequence that contains all of the objects from the first two sorted sequences. For example, if the first sequence is 3, 5, 8, 15 and the second sequence is **4**, **9**, **12**, **20**, the final sequence will be 3, **4**, 5, 8, **9**, **12**, 15, **20**. The algorithm for merging the two sequences follows.

Merge Algorithm

1. Access the first item from both sequences.
2. **`while`** not finished with either sequence
3. Compare the current items from the two sequences, copy the smaller current item to the output sequence, and access the next item from the input sequence whose item was copied.
4. Copy any remaining items from the first sequence to the output sequence.
5. Copy any remaining items from the second sequence to the output sequence.

The **`while`** loop (Step 2) merges items from both input sequences to the output sequence. The current item from each sequence is the one that has been most recently accessed but not yet copied to the output sequence. Step 3 compares the two current items and copies the smaller one to the output sequence. If input sequence A's current item is the smaller one, the next item is accessed from sequence A and becomes its current item. If input sequence B's current item is the smaller one, the next item is accessed from sequence B and becomes its current item. After the end of either sequence is reached, Step 4 or Step 5 copies the items from the other sequence to the output sequence. Note that either Step 4 or Step 5 is executed, but not both.

As an example, consider the sequences shown in Figure 8.4. Steps 2 and 3 will first copy the items from sequence A with the values **244** and **311** to the output sequence; then items from sequence B with values 324 and 415 will be copied; and then the item from sequence A with value **478** will be copied. At this point, we have copied all items in sequence A, so we exit the **`while`** loop and copy the remaining items from sequence B (499, 505) to the output (Steps 4 and 5).

Analysis of Merge

For two input sequences that contain a total of n elements, we need to move each element from its input sequence to its output sequence, so the time required for a merge is $O(n)$. How about the space requirements? We need to be able to store both initial sequences and the output sequence. So the array cannot be merged in place, and the *additional* space usage is $O(n)$.

Code for Merge

Listing 8.4 shows the merge algorithm applied to arrays of `Comparable` objects. Algorithm Steps 4 and 5 are implemented as **`while`** loops at the end of the method.

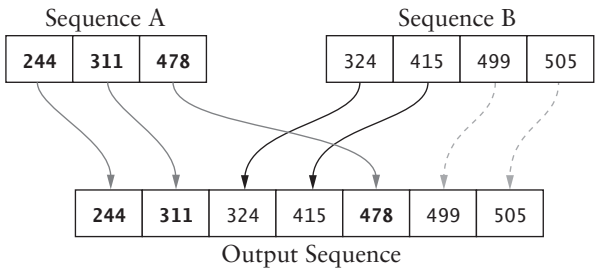


FIGURE 8.4
Merge Operation

LISTING 8.4
`Merge` Method

```
/** Merge two sequences.
    @pre leftSequence and rightSequence are sorted.
    @post outputSequence is the merged result and is sorted.
    @param outputSequence The destination
    @param leftSequence The left input
    @param rightSequence The right input
 */
private static <T extends Comparable<T>> void merge(T[] outputSequence,
                                                    T[] leftSequence,
                                                    T[] rightSequence) {
    int i = 0;
    // Index into the left input sequence.
    int j = 0;
    // Index into the right input sequence.
    int k = 0;
    // Index into the output sequence.
    // While there is data in both input sequences
    while (i < leftSequence.length && j < rightSequence.length) {
        // Find the smaller and
        // insert it into the output sequence.
        if (leftSequence[i].compareTo(rightSequence[j]) < 0) {
            outputSequence[k++] = leftSequence[i++];
        } else {
            outputSequence[k++] = rightSequence[j++];
        }
    }
    // assert: one of the sequences has more items to copy.
    // Copy remaining input from left sequence into the output.
    while (i < leftSequence.length) {
        outputSequence[k++] = leftSequence[i++];
    }
    // Copy remaining input from right sequence into output.
    while (j < rightSequence.length) {
        outputSequence[k++] = rightSequence[j++];
    }
}
```

PROGRAM STYLE

By using the postincrement operator on the index variables, you can both extract the current item from one sequence and append it to the end of the output sequence in one statement. The statement:

```
outputSequence[k++] = leftSequence[i++];
```

is equivalent to the following three statements, executed in the order shown:

```
outputSequence[k] = leftSequence[i];
k++;
i++;
```

Both the single statement and the group of three statements maintain the invariant that the indexes reference the current item.

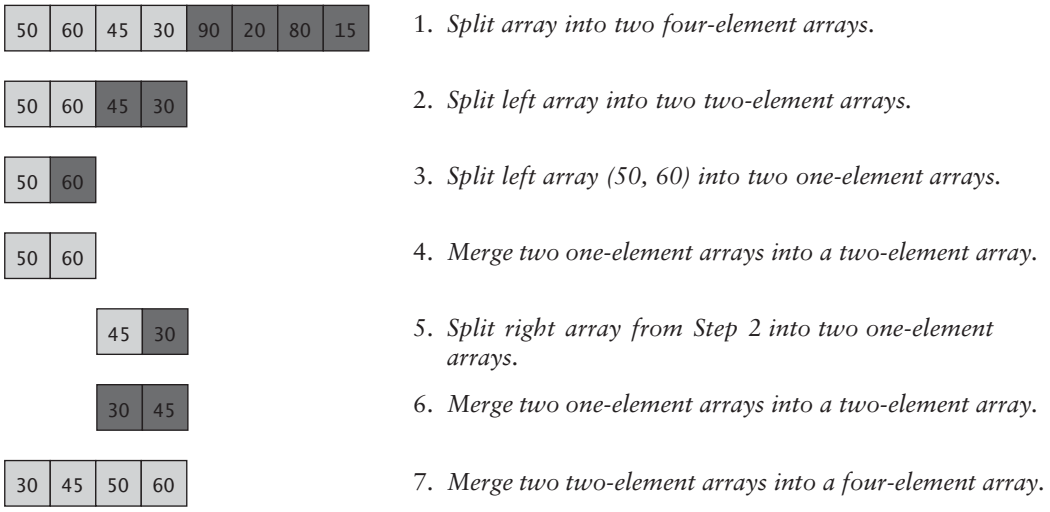


FIGURE 8.5
Trace of Merge Sort

Algorithm for Merge Sort

We can modify merging to serve as an approach to sorting a single, unsorted array as follows:

1. Split the array into two halves.
2. Sort the left half.
3. Sort the right half.
4. Merge the two.

What sort algorithm should we use to do Steps 2 and 3? We can use the merge sort algorithm we are developing! The base case will be a table of size 1, which is already sorted, so there is nothing to do for the base case. We write the algorithm next, showing its recursive step.

Algorithm for Merge Sort

1. `if` the `tableSize` is > 1
2. Set `halfSize` to `tableSize` divided by 2.
3. Allocate a table called `leftTable` of size `halfSize`.
4. Allocate a table called `rightTable` of size `tableSize - halfSize`.
5. Copy the elements from `table[0 ... halfSize - 1]` into `leftTable`.
6. Copy the elements from `table[halfSize ... tableSize]` into `rightTable`.
7. Recursively apply the merge sort algorithm to `leftTable`.
8. Recursively apply the merge sort algorithm to `rightTable`.
9. Apply the merge method using `leftTable` and `rightTable` as the input and the original table as the output.

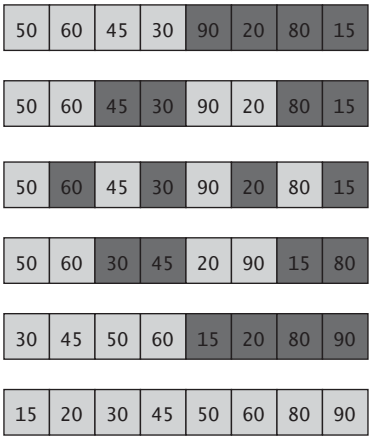
Trace of Merge Sort Algorithm

Figure 8.5 illustrates the merge sort. Each recursive call to method `sort` with an array argument that has more than one element splits the array argument into a left array and a right array, where each new array is approximately half the size of the array argument. We then

sort each of these arrays, beginning with the left half, by recursively calling method `sort` with the left array and right array as arguments. After returning from the sort of the left array and right array at each level, we merge these two halves together back into the space occupied by the array that was split. The left subarray in each recursive call (in gray) will be sorted before the processing of its corresponding right subarray (shaded dark) begins. Lines 4 and 6 merge two one-element arrays to form a sorted two-element array. At line 7, the two sorted two-element arrays (50, 60 and 30, 45) are merged into a sorted four-element array. Next, the right subarray shaded dark on line 1 will be sorted in the same way. When done, the sorted subarray (15, 20, 80, 90) will be merged with the sorted subarray on line 7.

Analysis of Merge Sort

In Figure 8.5, the size of the arrays being sorted decreases from 8 to 4 (line 1) to 2 (line 2) to 1 (line 3). After each pair of subarrays is sorted, the pair will be merged to form a larger sorted array. Rather than showing a time sequence of the splitting and merging operations, we summarize them as follows:



Lines 1 through 3 show the splitting operations, and lines 4 through 6 show the merge operations. Line 4 shows the two-element arrays formed by merging two-element pairs, line 5 shows the four-element arrays formed by merging two-element pairs, and line 6 shows the sorted array. Because each of these lines involves a movement of n elements from smaller-size arrays to larger arrays, the effort to do each merge is $O(n)$. The number of lines that require merging (three in this case) is $\log n$ because each recursive step splits the array in half. So the total effort to reconstruct the sorted array through merging is $O(n \log n)$.

Recall from our discussion of recursion that whenever a recursive method is called, a copy of the local variables is saved on the run-time stack. Thus, as we go down the recursion chain sorting the `leftTables`, a sequence of `rightTables` of size $\frac{n}{2}, \frac{n}{4}, \cdots, \frac{n}{2^k}$ is allocated. Since $\frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1 = n - 1$, a total of n additional storage locations are required.

Code for Merge Sort

Listing 8.5 shows the `MergeSort` class.

LISTING 8.5
MergeSort.java

```
/** Implements the recursive merge sort algorithm. In this version, copies
    of the subtables are made, sorted, and then merged.
 */
public class MergeSort {
    /** Sort the array using the merge sort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        // A table with one element is sorted already.
        if (table.length > 1) {
            // Split table into halves.
            int halfSize = table.length / 2;
            T[] leftTable = (E[]) new Comparable[halfSize];
            T[] rightTable = (E[]) new Comparable[table.length - halfSize];
            System.arraycopy(table, 0, leftTable, 0, halfSize);
            System.arraycopy(table, halfSize, rightTable, 0,
                             table.length - halfSize);

            // Sort the halves.
            sort(leftTable);
            sort(rightTable);

            // Merge the halves.
            merge(table, leftTable, rightTable);
        }
    }
    // See Listing 8.4 for the merge method.
    . . .
}
```

EXERCISES FOR SECTION 8.6

SELF-CHECK

1. Trace the execution of the merge sort on the following array, providing a figure similar to Figure 8.5.

   55 50 10 40 80 90 60 100 70 80 20 50 22

2. For the array in Question 1 above, show the value of `halfSize` and arrays `leftTable` and `rightTable` for each recursive call to method `sort` in Listing 8.4 and show the array elements after returning from each call to `merge`. How many times is `sort` called, and how many times is `merge` called?

PROGRAMMING

1. Add statements that trace the progress of method `sort` by displaying the array `table` after each merge operation. Also display the arrays referenced by `leftTable` and `rightTable`.