19.8 Merge Sort

Merge sort is an *efficient* sorting algorithm but is conceptually *more complex* than selection sort and insertion sort. The merge sort algorithm sorts an array by *splitting* it into two equal-sized subarrays, *sorting* each subarray, then *merging* them into one larger array. With an odd number of elements, the algorithm creates the two subarrays such that one has one more element than the other.

The implementation of merge sort in this example is recursive. The base case is an array with one element, which is, of course, sorted, so the merge sort immediately returns in this case. The recursion step splits the array into two approximately equal pieces, recursively sorts them, then merges the two sorted arrays into one larger, sorted array.

Suppose the algorithm has already merged smaller arrays to create sorted arrays A:

```
4  10  34  56  77
```

and B:

```
5  30  51  52  93
```

Merge sort combines these two arrays into one larger, sorted array. The smallest element in A is 4 (located in the zeroth index of A). The smallest element in B is 5 (located in the zeroth index of B). In order to determine the smallest element in the larger array, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the first element in the merged array. The algorithm continues by comparing 10 (the second element in A) to 5 (the first element in B). The value from B is smaller, so 5 becomes the second element in the larger array. The algorithm continues by comparing 10 to 30, with 10 becoming the third element in the array, and so on.

19.8.1 Merge Sort Implementation

Figure 19.6 declares the `MergeSortTest` class, which contains:

- `static` method `mergeSort` to initiate the sorting of an `int` array using the merge sort algorithm,
- `static` method `sortArray` to perform the recursive merge sort algorithm—this is called by method `mergeSort`,
- `static` method `merge` to merge two sorted subarrays into a single sorted subarray,
- `static` method `subarrayString` to get a subarray’s `String` representation for output purposes, and
- `main` to test method `mergeSort`.

Method `main` (lines 104–113) is identical to `main` in Figs.

19.4–19.5 except that line 111 calls method `mergeSort` to sort the array elements into ascending order. The output from this program displays the splits and merges performed by merge sort, showing the progress of the sort at each step of the algorithm. It’s well worth your time to step through these outputs to fully understand this elegant sorting algorithm.

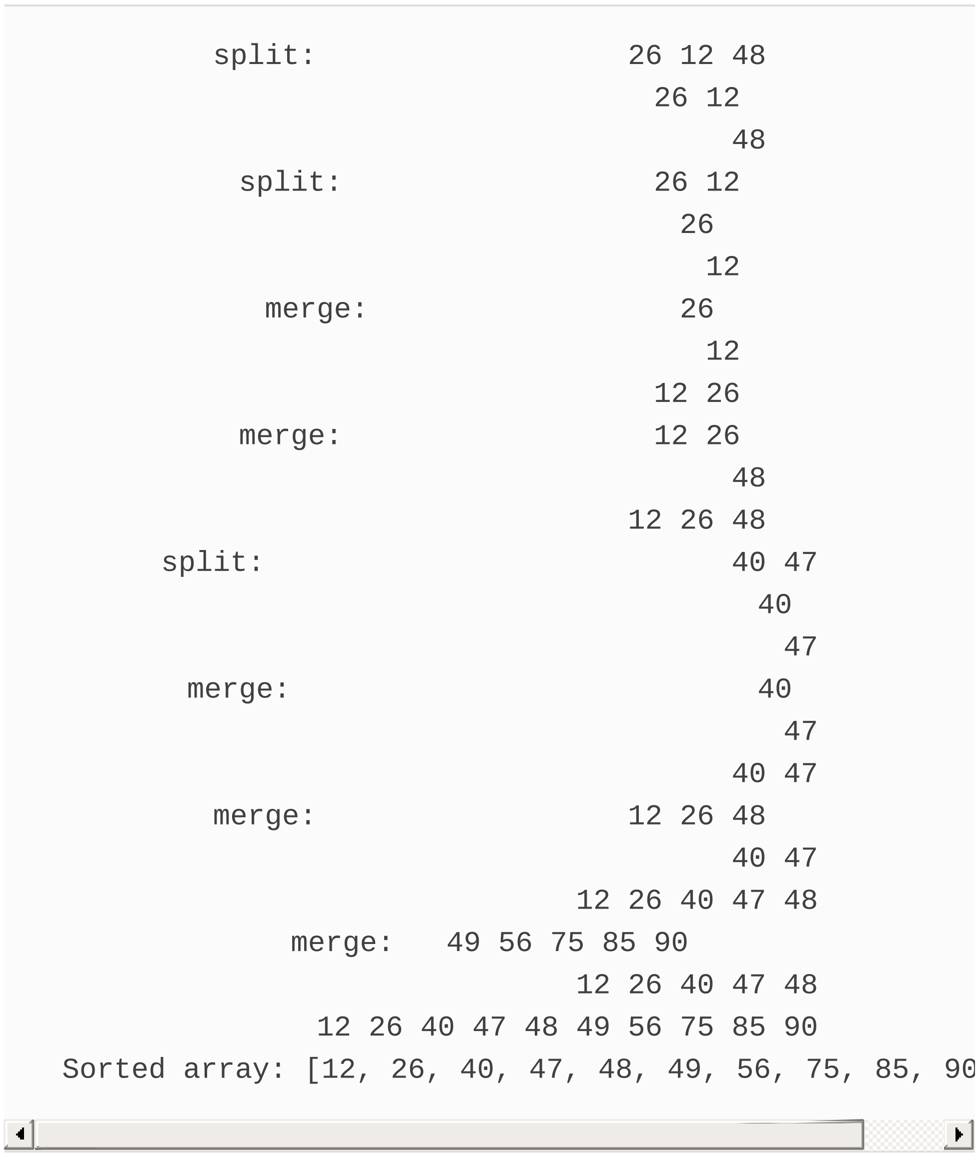
```
1   // Fig. 19.6: MergeSortTest.java
2   // Sorting an array with merge sort.
3   import java.security.SecureRandom;
4   import java.util.Arrays;
5
6   public class MergeSortTest {
7      // calls recursive sortArray method to begin
8      public static void mergeSort(int[] data) {
9         sortArray(data, 0, data.length - 1); // so
10     }
11
12     // splits array, sorts subarrays and merges s
13     private static void sortArray(int[] data, int
14        // test base case; size of array equals 1
15        if ((high - low) >= 1) { // if not base ca
16           int middle1 = (low + high) / 2; // calc
17           int middle2 = middle1 + 1; // calculate
18
19           // output split step
20           System.out.printf("split:  %s%n",
21              subarrayString(data, low, high));
22           System.out.printf("  %s%n",
23              subarrayString(data, low, middle1));
24           System.out.printf("  %s%n%n",
25              subarrayString(data, middle2, high))
26
27           // split array in half; sort each half
28           sortArray(data, low, middle1); // first
29           sortArray(data, middle2, high); // seco
30
```

```java
            // merge two sorted arrays after split
            merge (data, low, middle1, middle2, hig
        }
    }

    // merge two sorted subarrays into one sorted
    private static void merge(int[] data, int lef
            int middle2, int right) {

        int leftIndex = left; // index into left s
        int rightIndex = middle2; // index into ri
        int combinedIndex = left; // index into te
        int[] combined = new int[data.length]; //

        // output two subarrays before merging
        System.out.printf("merge:  %s%n",
            subarrayString(data, left, middle1));
        System.out.printf("  %s%n",
            subarrayString(data, middle2, right));

        // merge arrays until reaching end of eit
        while (leftIndex <= middle1 && rightIndex
            // place smaller of two current elemen
            // and move to next space in arrays
            if (data[leftIndex] <= data[rightIndex
                combined[combinedIndex++] = data[le
            }
            else {
                combined[combinedIndex++] = data[ri
            }
        }

        // if left array is empty
        if (leftIndex == middle2) {
            // copy in rest of right array
            while (rightIndex <= right) {
                combined[combinedIndex++] = data[ri
            }
        }
        else { // right array is empty
```

```
         // copy in rest of left array
         while (leftIndex <= middle1) {
            combined[combinedIndex++] = data[le
         }
      }

      // copy values back into original array
      for (int i = left; i <= right; i++) {
         data[i] = combined[i];
      }

      // output merged array
      System.out.printf("  %s%n%n",
         subarrayString(data, left, right));
   }

   // method to output certain values in array
   private static String subarrayString(int[] d
      StringBuilder temporary = new StringBuild

      // output spaces for alignment
      for (int i = 0; i < low; i++) {
         temporary.append("  ");
      }

      // output elements left in array
      for (int i = low; i <= high; i++) {
         temporary.append(" " + data[i]);
      }

      return temporary.toString();
   }

   public static void main(String[] args) {
      SecureRandom generator = new SecureRandom

      // create unordered array of 10 random in
      int[] data = generator.ints(10, 10, 91).t

      System.out.printf("Unsorted array: %s%n%n
```

```java
      mergeSort(data); // sort array
      System.out.printf("Sorted array: %s%n", A
   }
}
```

```
            Unsorted array:
[75, 56, 85, 90, 49, 26, 12, 48, 40, 47]
 split:   75 56 85 90 49 26 12 48 40 47
                 75 56 85 90 49
                         26 12 48 40 47
        split:   75 56 85 90 49
                    75 56 85
                          90 49
           split:   75 56 85
                      75 56
                          85
             split:   75 56
                       75
                         56
              merge:   75
                         56
                      56 75
             merge:   56 75
                          85
                    56 75 85
        split:            90 49
                            90
                             49
          merge:            90
                             49
                          49 90
           merge:   56 75 85
                          49 90
                 49 56 75 85 90
 split:                  26 12 48 40 47
                            26 12 48
                                  40 47
```



```
         split:                  26 12 48
                                   26 12
                                       48
           split:                  26 12
                                    26
                                      12
            merge:                  26
                                      12
                                   12 26
           merge:                  12 26
                                       48
                                 12 26 48
      split:                           40 47
                                         40
                                          47
        merge:                           40
                                          47
                                       40 47
         merge:                  12 26 48
                                       40 47
                              12 26 40 47 48
              merge:   49 56 75 85 90
                              12 26 40 47 48
               12 26 40 47 48 49 56 75 85 90
Sorted array: [12, 26, 40, 47, 48, 49, 56, 75, 85, 90
```

Fig. 19.6

Sorting an array with merge sort.

Method `mergeSort`

Lines 8–10 of Fig. 19.6 declare the `mergeSort` method. Line 9 calls method `sortArray` with `0` and `data.length - 1` as the arguments—corresponding to the beginning and ending indices, respectively, of the array to be sorted. These values tell method `sortArray` to operate on the entire array.

Recursive Method `sortArray`

Method `sortArray` (lines 13–34) performs the recursive merge sort algorithm. Line 15 tests the base case. If the size of the array is `1`, the array is already sorted, so the method returns immediately. If the size of the array is greater than `1`, the method splits the array in two, recursively calls method `sortArray` to sort the two subarrays, then merges them. Line 28 recursively calls method `sortArray` on the first half of the array, and line 29 recursively calls method `sortArray` on the second half. When these two method calls return, each half of the array has been sorted. Line 32 calls method `merge` (lines 37–85) on the two halves of the array to combine the two sorted arrays into one larger sorted array.

Method `merge`

Lines 37–85 declare method `merge`. Lines 52–61 in `merge` loop until the end of either subarray is reached. Line 55 tests which element at the beginning of the arrays is smaller. If the element in the left array is smaller, line 56 places it in position in the combined array. If the element in the right array is smaller, line 59 places it in position in the combined array. When the `while` loop completes, one entire subarray has been placed in the combined array, but the other subarray still contains data. Line 64 tests whether the left array has reached the end. If so, lines 66–68 fill the combined array with the elements of the right array. If the left array has not reached the end, then the right array must have reached the end, and lines 72–74 fill the combined array with the elements of the left array. Finally, lines 78–80 copy the combined array into the original array.

19.8.2 Efficiency of the Merge Sort

Merge sort is *far more efficient* than insertion or selection sort. Consider the first (non-recursive) call to `sortArray`. This results in two recursive calls to `sortArray` with subarrays each approximately half the original array's size, and a single call to `merge`, which requires, at worst, $n-1$ comparisons to fill the original array, which is $O(n)$. (Recall that each array element can be chosen by comparing one element from each subarray.) The two calls to `sortArray` result in four more recursive `sortArray` calls, each with a subarray approximately a quarter of the original array's size, along with two calls to `merge` that each require, at worst, $n/2\ -1$ comparisons, for a total number of comparisons of $O(n)$. This process continues, each `sortArray` call generating two additional `sortArray` calls and a `merge` call, until the algorithm has *split* the array into one-element subarrays. At each level, $O(n)$ comparisons are required to *merge* the subarrays. Each level splits the arrays in half, so doubling the array size requires one more level. Quadrupling the array size requires two more levels. This pattern is logarithmic and results in $\log_2 n$ levels. This results in a total efficiency of **O(*n* log *n*)**.