THE DIVIDE AND CONQUER APPROACH

Many useful algorithms are *recursive* in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems. These algorithms typically follow a **divide-and-conquer** approach: they break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion: **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.

**Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.

**Combine** the solutions to the sub-problems into the solution for the original problem.

The *merge sort* algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

**Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.

**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

The recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. We merge by calling an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that p ≤ q < r . The procedure assumes that the subarrays A(p . . q) and A(q+1. . r) are in sorted order. It *merges* them to form a single sorted subarray that replaces the current subarray A(p . . r).

Our MERGE procedure takes time *(n),* where n = r – p + 1 is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile.

Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most *n* basic steps, merging takes *(n)* time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step.

We place on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use ∞ as the sentinel value, so that whenever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the non-sentinel cards have already been placed onto the output pile. Since we know in advance that exactly r - p + 1 cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

MERGE(A, p, q, r)

1 n1 ← q - p + 1

2 n2 ← r - q

3 let L[1 . . n1+1] and R[1 . . n2 +1] be new arrays

4 for i ← 1 to n1

5 do L[i] ← A[p + i – 1]

6 for j ← 1 to n2

7 do R[j] ← A[q + j]

8 L[n1 + 1] ← ∞

9 R[n2 + 1] ← ∞

10 i ← 1

11 j ← 1

12 for k ← p to r

13 do if L[i ] ≤ R[j]

14 then A[k] ← L[i]

15 i ← i + 1

16 else A[k] ← R[j]

17 j ← j + 1

In detail, the MERGE procedure works as follows. Line 1 computes the length n1 of the subarray A[p . . q], and line 2 computes the length n2 of the subarray A[q + 1 . . r]. We create arrays L and R (“left” and “right”), of lengths n1 + 1 and n2 + 1, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray A[p . . q] into L[1 . . n1], and the **for** loop of lines 6–7 copies the subarray A[q + 1. . r] into R[1 . . n2]. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illustrated in Figure 2.3, perform the r - p + 1 basic steps by maintaining the following loop invariant:

At the start of each iteration of the for loop of lines 12–17, the subarray A[p . . k – 1] contains the k - p smallest elements of L[1 : : n1 + 1] and R[1 . . n2 + 1], in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.





**Figure 2.3** The operation of lines 10–17 in the call MERGE(A, 9, 12, 16), when the subarray A[9 . . 16] contains the sequence <2; 4; 5; 7; 1; 2; 3; 6>. After copying and inserting sentinels, the array L contains <2; 4; 5; 7; ∞>, and the array R contains <1; 2; 3; 6; ∞>. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in A[9 . . 16], along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. (a)–(h) The arrays A, L, and R, and their respective indices k, i , and j prior to each iteration of the loop of lines 12–17. (i) The arrays and indices at termination. At this point, the subarray inA[9 . . 16] is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

**Initialization:** Prior to the first iteration of the loop, we have k = p, so that the subarray A[p . . k – 1] is empty. This empty subarray contains the k - p = 0 smallest elements of L and R, and since i = j = 1, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

**Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that L[i] ≤ R[j]. Then L[i] is the smallest element not yet copied back into A. Because A[p . . k – 1] contains the k - p smallest elements, after line 14 copies L[i] into A[k], the subarray A[p . . k] will contain the k - p + 1 smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead L[i] > R[j], then lines 16–17 perform the appropriate action to maintain the loop invariant.

**Termination:** At termination, k = r + 1. By the loop invariant, the subarray A[p . . k – 1], which is A[p . . r], contains the k - p = r - p + 1 smallest elements of L[1 . . n1 + 1] and R[1 . . n2 + 1], in sorted order. The arrays L and R together contain n1 + n2 + 2 = r - p + 3 elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in *(n)* time, where n = r - p + 1, observe that each of lines 1–3 and 8–11 takes constant time, the **for** loops of lines 4–7 take *(*n1 + n2) = *(n)* time,7 and there are *n* iterations of the **for** loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) sorts the elements in the sub array A[p . . r]. If p ≥ r , the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions A[p . . r] into two subarrays: A[p . . q], containing ⎾n/2⏋ elements, and A[q + 1. . r], containing ⎿n/2⏌ elements.

MERGE-SORT(A, p, r)

1 if p < r

2 q ← ⎿p + r)/2⏌

3 MERGE-SORT(A, p, q)

4 MERGE-SORT(A, q + 1, r)

5 MERGE(A, p, q, r)

To sort the entire sequence A = 〈A[1], A[2, . . . , A[n]〉, we make the

initial call MERGE-SORT(A, 1, *length*[A]), where once again *length*[A]= n. Figure 2.4 illustrates the operation of the procedure bottom-up (left –to-right) when *n* is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length *n/2* are merged to form the final sorted sequence of length *n*.

The figure below (which is Figure 2.4) illustrates this merge process from left to right on array A = [5, 2, 4, 7, 1, 3, 2, 6].

5 2 2 1

2 merge 2 🡺 5 4 2

4 4 5 2

7 merge 2🡺 7 merge 4🡺 7 3

1 1 1 4

3 merge 2🡺 3 2 5

2 2 3 6

6 merge 2🡺 6 merge 4🡺 6 merge 8🡺 7