modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Figure 2.1: Sorting a hand of cards using insertion sort.

Our pseudocode for insertion sort is presented as a procedure called INSERTION-SORT, which takes as a parameter an array $A[1 \square n]$ containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by $length[A]$.) The input numbers are ***sorted in place***: the numbers are rearranged within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when INSERTION-SORT is finished.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▸ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \square 5, 2, 4, 6, 1, 3 \square$. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the "outer" **for** loop, which is indexed by j, the subarray consisting of elements $A[1 \square j - 1]$ constitute the currently sorted hand, and elements $A[j + 1 \square n]$ correspond to the pile of cards still on the table. In fact, elements $A[1 \square j - 1]$ are the elements *originally* in positions 1 through j - 1, but now in sorted order. We state these properties of $A[1 \square j - 1]$ formally as a ***loop invariant***:

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the 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 checking just two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ 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. The idea is to put 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 nonsentinel 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  create arrays L[1 □ n1 + 1] and R[1 □ n2 + 1]
 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 n_1 of the subarray $A[p \,\square\, q]$, and line 2 computes the length n_2 of the subarray $A[q + 1 \,\square\, r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3. The **for** loop of lines 4-5 copies the subarray $A[p \,\square\, q]$ into $L[1 \,\square\, n_1]$, and the **for** loop of lines 6-7 copies the subarray $A[q + 1 \,\square\, r]$ into $R[1 \,\square\, n_2]$. 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 $A[p \,\square\, k - 1]$ contains the $k - p$ smallest elements of $L[1 \,\square\, n_1 + 1]$ and $R[1 \,\square\, n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

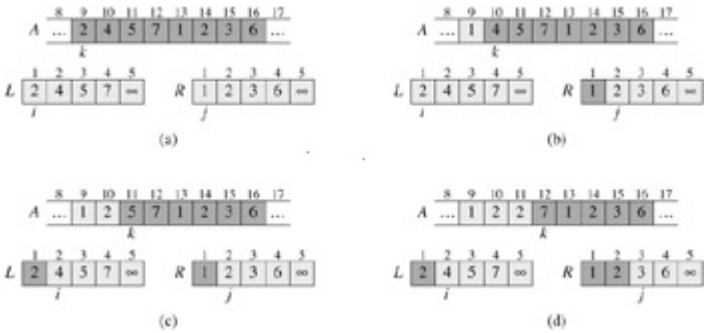
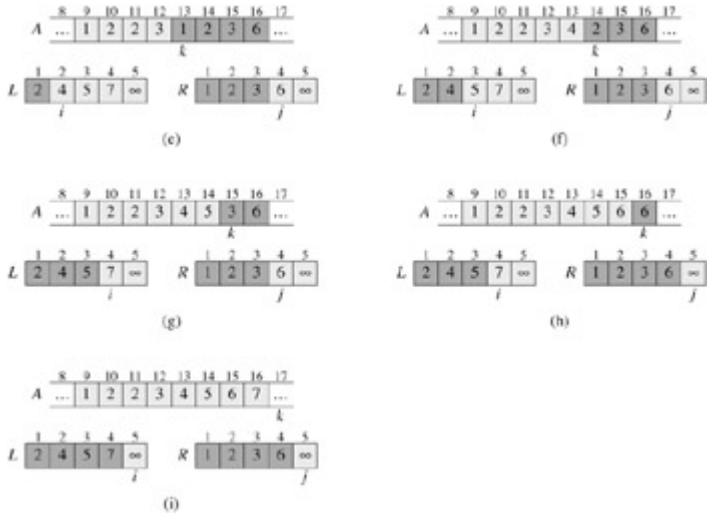


Figure 2.3: The operation of lines 10-17 in the call MERGE(*A*, 9, 12, 16), when the subarray *A*[9 □ 16] contains the sequence □2, 4, 5, 7, 1, 2, 3, 6□. After copying and inserting sentinels, the array *L* contains □2, 4, 5, 7, ∞□, and the array *R* contains □1, 2, 3, 6, ∞□. Lightly shaded positions 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 in *A*[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*.

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.

- **Initialization:** Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p \,\square\, k - 1]$ 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 $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into *A*. Because $A[p \,\square\, k - 1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \,\square\, k]$ 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