The following procedure implements quicksort.

```
QUICKSORT(A, p, r)
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, \mathit{length}[A])$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Figure 7.1 shows the operation of PARTITION on an 8-element array. PARTITION always selects an element $x = A[r]$ as a ***pivot*** element around which to partition the subarray $A[p \,..\, r]$. As the procedure runs, the array is partitioned into four (possibly empty) regions. At the start of each iteration of the **for** loop in lines 3–6, each region satisfies certain properties, which we can state as a loop invariant:

> At the beginning of each iteration of the loop of lines 3–6, for any array index k,
>
> 1. If $p \le k \le i$, then $A[k] \le x$.
> 2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
> 3. If $k = r$, then $A[k] = x$.

Figure 7.2 summarizes this structure. The indices between j and $r - 1$ are not covered by any of the three cases, and the values in these entries have no particular relationship to the pivot x.

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

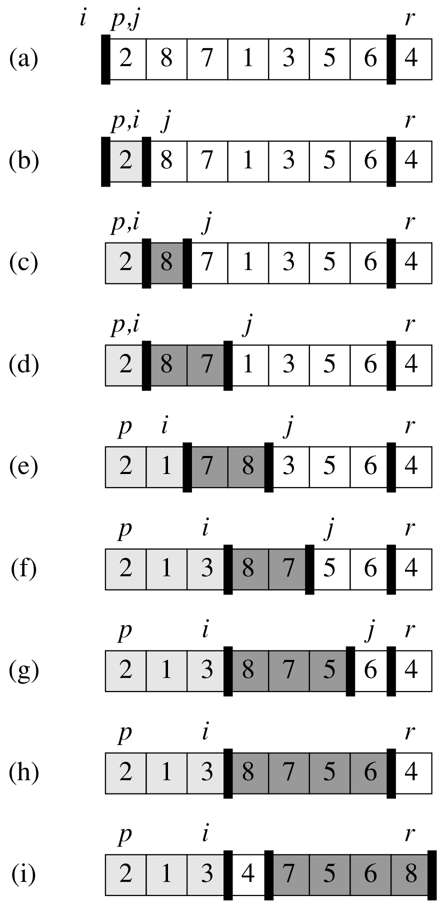

Figure 7.1 The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6 and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Initialization: Prior to the first iteration of the loop, $i = p - 1$, and $j = p$. There are no values between p and i, and no values between $i + 1$ and $j - 1$, so the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, there are two cases to consider, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when

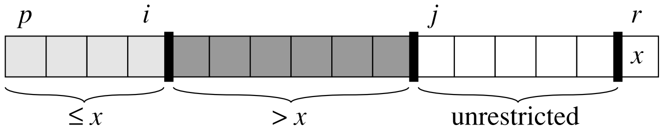

Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \mathinner{..} r]$. The values in $A[p \mathinner{..} i]$ are all less than or equal to x, the values in $A[i+1 \mathinner{..} j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j \mathinner{..} r-1]$ can take on any values.

$A[j] > x$; the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$; i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

The final two lines of PARTITION move the pivot element into its place in the middle of the array by swapping it with the leftmost element that is greater than x. The output of PARTITION now satisfies the specifications given for the divide step.

The running time of PARTITION on the subarray $A[p \mathinner{..} r]$ is $\Theta(n)$, where $n = r - p + 1$ (see Exercise 7.1-3).

Exercises

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

7.1-2

What value of q does PARTITION return when all elements in the array $A[p \mathinner{..} r]$ have the same value? Modify PARTITION so that $q = (p+r)/2$ when all elements in the array $A[p \mathinner{..} r]$ have the same value.

7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.