Quick Sort

In general, selection sort and insertion sort algorithms are not efficient. In this section, we describe the quick sort algorithm, which, in general, is much more efficient than selection and insertion sort. The quick sort algorithm was developed by C.A.R. Hoare in 1962.

The quick sort algorithm uses the divide-and-conquer technique to sort a list. The list is partitioned into two sublists, and the two sublists are then sorted and combined into one list in such a way so that the combined list is sorted. Thus, the general algorithm is

```
if (the list size is greater than 1)
{
   a. Partition the list into two sublists, say lowerSublist and
      upperSublist.
   b. Quick sort lowerSublist.
   c. Quick sort upperSublist.
   d. Combine the sorted lowerSublist and sorted upperSublist.
}
```

After partitioning the list into two sublists—`lowerSublist` and `upperSublist`, these two sublists are sorted using the quick sort algorithm. In other words, we use *recursion* to implement the quick sort algorithm.

In the quick sort algorithm, the list is partitioned in such way that combining the sorted `lowerSublist` and `upperSublist` is trivial. Therefore, in a quick sort, all the sorting work is done in partitioning the list. Because all the sorting work occurs during the partitioning of the list, we first describe the partition procedure in detail.

To partition the list into two sublists, first we choose an element of the list called `pivot`. The `pivot` is used to divide the list into two sublists: `lowerSublist` and `upperSublist`. The elements in `lowerSublist` are smaller than `pivot`, and the elements in `upperSublist` are greater than `pivot`. For example, consider the list in Figure Q-1.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
list	45	82	25	94	50	60	78	32	92

FIGURE Q-1 `list` before the partition

There are several ways to determine `pivot`. However, `pivot` is chosen so that, it is hoped, `lowerSublist` and `upperSublist` are of nearly equal size. (The choice of `pivot` affects the performance of the algorithm.) For illustration purposes, let us choose the middle element of the list as `pivot`. The partition procedure that we describe partitions this list using `pivot` as the middle element, in our case `50`, as shown in Figure Q-2.

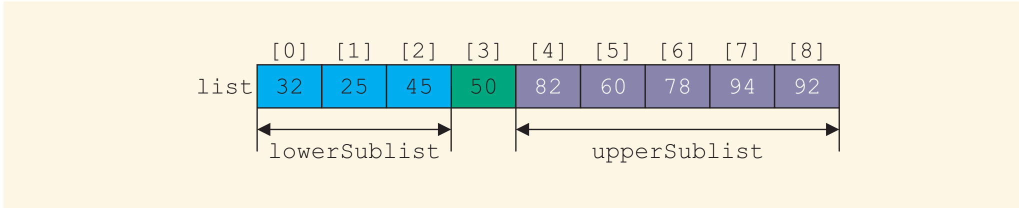

FIGURE Q-2 `list` after the partition

From Figure Q-2, it follows that after partitioning `list` into `lowerSublist` and `upperSublist`, pivot is in the right place. Thus, after sorting `lowerSublist` and `upperSublist`, combining the two sorted sublists is trivial.

The partition algorithm is as follows (we assume that `pivot` is chosen as the middle element of the list):

1. Determine `pivot`, and swap `pivot` with the first element of the list.

   Suppose that the index `smallIndex` points to the last element less than `pivot`. The index `smallIndex` is initialized to the first element of the list.
2. For the remaining elements in the list (starting at the second element):

   If the current element is less than pivot

   a. Increment `smallIndex`.

   b. Swap the current element with the array element pointed to by `smallIndex`.
3. Swap the first element, that is, `pivot`, with the array element pointed to by `smallIndex`.

Step 2 can be implemented using a **`for`** loop, with the loop starting at the second element of the list.

Step 1 determines the pivot and moves `pivot` to the first array position. During the execution of Step 2, the list elements get arranged as shown in Figure Q-3. (Suppose the name of the array containing the list elements is list.)

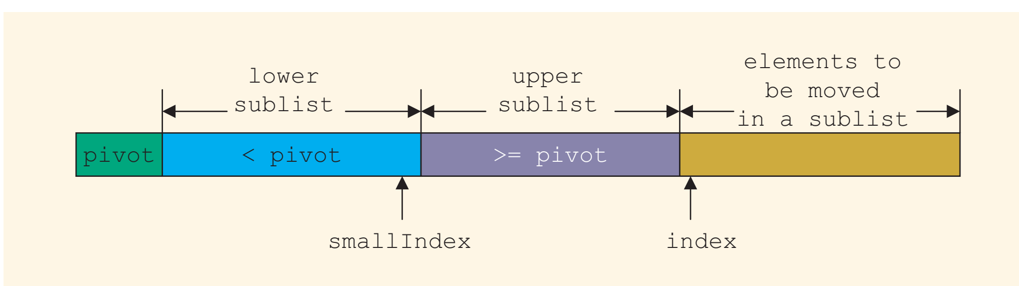

FIGURE Q-3 List during the execution of Step 2

As shown in Figure Q-3, `pivot` is in the first array position. Elements in the lower sublist are less than `pivot`; elements in the upper sublist are greater than or equal to `pivot`. The variable `smallIndex` contains the index of the last element of the lower sublist; the variable `index` contains the index of the next element that needs to be moved either in the lower sublist or in the upper sublist. As explained in Step 2, if the next element of the list (that is, `list[index]`) is less than `pivot`, we advance `smallIndex` to the next array position and swap `list[ index]` with `list[smallIndex]`). Next we illustrate Step 2.

Suppose that the list is as given in Figure Q-4.

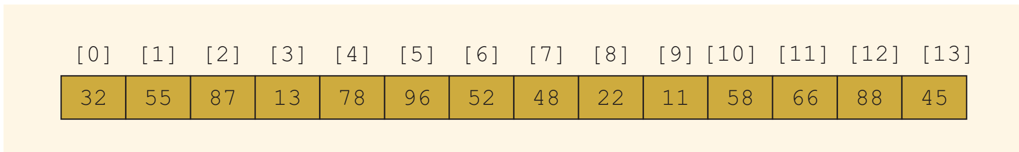

FIGURE Q-4 List before sorting

Step 1 requires us to determine the `pivot` and swap it with the first array element. For the list in Figure Q-4, the middle element is at the position `(0 + 13) / 2 = 6`. That is, `pivot` is at position `6`. Therefore, after swapping `pivot` with the first array element, the list is as shown in Figure Q-5. (Notice that in Figure Q-5, `52` is swapped with `32`.)

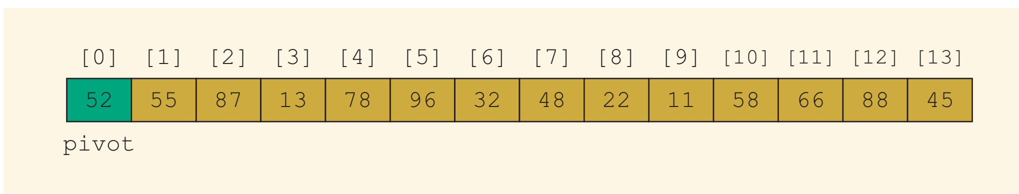

FIGURE Q-5 List after moving `pivot` to the first array position.

Suppose that after executing Step 2 a few times, the list is as shown in Figure Q-6.

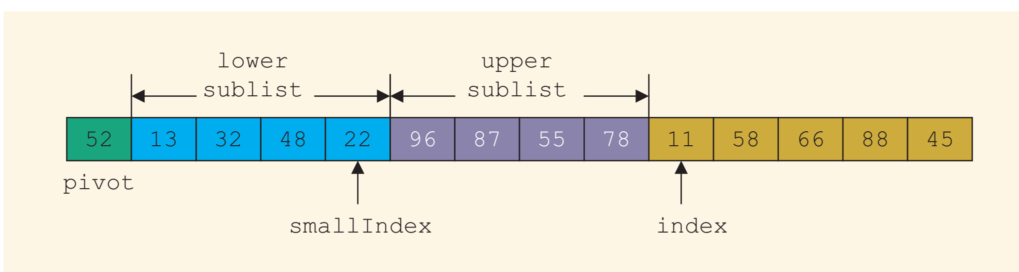

FIGURE Q-6 List after a few iterations of Step 2

As shown in Figure Q-6, the next element of the list that needs to be moved into a sublist is indicated by `index`. Because `list[index] < pivot`, we need to move the element `list[index]` into the lower sublist. To do so, we first advance `smallIndex` to the next array position and then swap `list[smallIndex]` with `list[index]`. The resulting list is as shown in Figure Q-7. (Notice that `11` is swapped with `96`.)

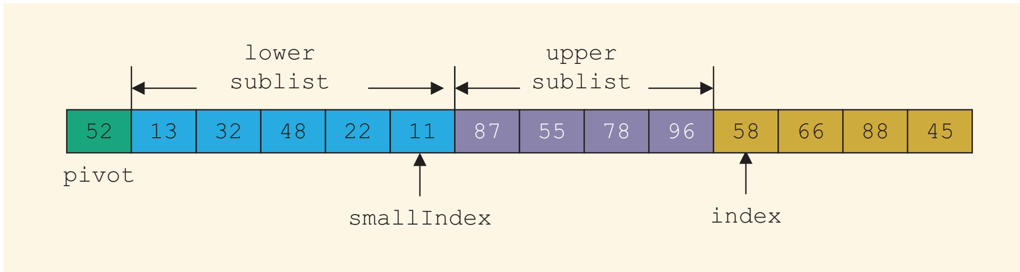

FIGURE Q-7 List after moving `11` into the lower sublist

Now consider the list in Figure Q-8.

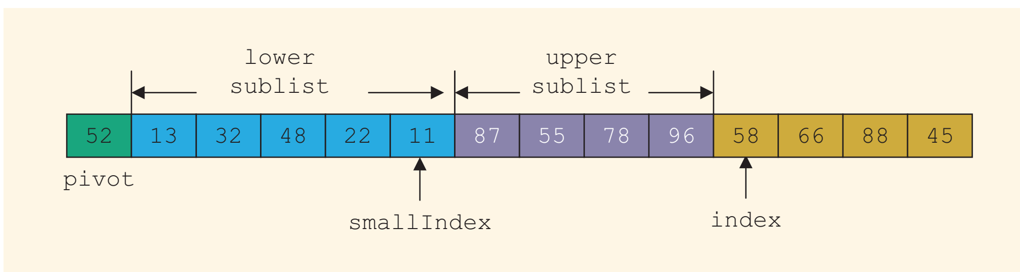

FIGURE Q-8 List before moving `58` into a sublist

For the list in Figure Q-8, `list[index]` is `58`, which is greater than `pivot`. Therefore, `list[index]` is to be moved in the upper sublist. This is accomplished by leaving `58` at its position and increasing the size of the upper sublist, by one, to the next array position. After moving `58` into the upper sublist, the list is shown in Figure Q-9.

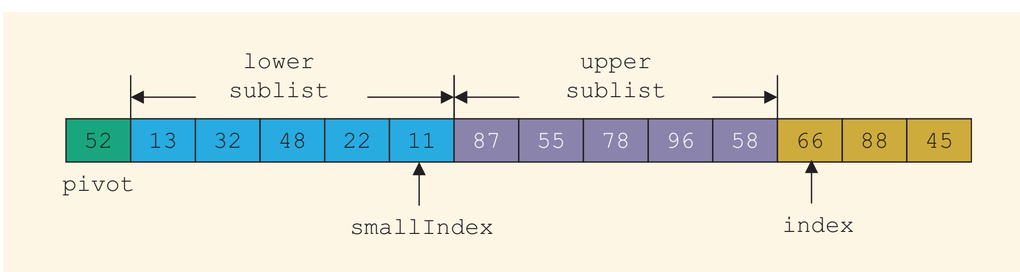

FIGURE Q-9 List after moving `58` into the upper sublist

After moving the elements that are less than pivot into the lower sublist and elements that are greater than pivot into the upper sublist (that is, after completely executing Step 2), Figure Q-10 shows the resulting list.

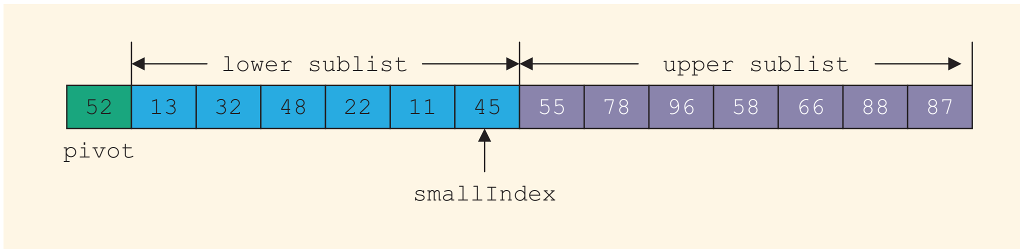

FIGURE Q-10 List elements after arranging into the lower sublist and upper sublist

Next we execute Step 3 and move `52`, `pivot`, to the proper position in the list. This is accomplished by swapping `52` with `45`. The resulting list is as shown in Figure Q-11.

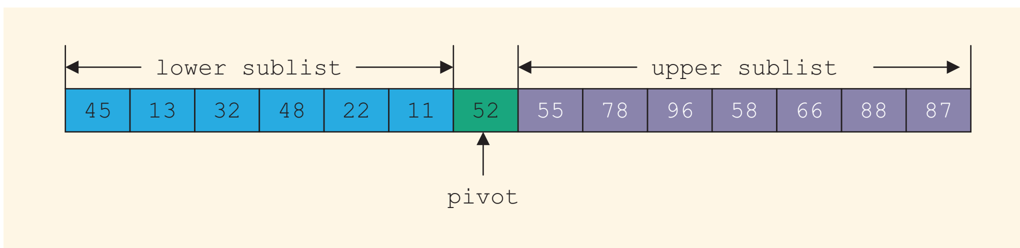

FIGURE Q-11 List after swapping `52` with `45`

As shown in Figure Q-11, Steps 1, 2, and 3 in the preceding algorithm partition the list into two sublists. The elements less than pivot are in the lower sublist; the elements greater than or equal to `pivot` are in the upper sublist.

To partition the list into the lower and upper sublists, we need to keep track of only the last element of the lower sublist and the next element of the list that needs to be moved into either the lower sublist or the upper sublist. In fact, the upper sublist is between the two indices `smallIndex` and `index`.

We now write the method, `partition`, to implement the preceding partition algorithm. After rearranging the elements of the list, the method `partition` returns the location of `pivot` so that we can determine the starting and ending locations of the sublists. The definition of the method `partition` is:

```
private int partition(int[] list, int first, int last)
{
    int pivot;

    int smallIndex;

    swap(list, first, (first + last) / 2);

    pivot = list[first];
    smallIndex = first;

    for (int index = first + 1; index <= last; index++)
    {
        if (list[index] < list[pivot])
        {
            smallIndex++;
            swap(list, smallIndex, index);
        }
    }

    swap(list, first, smallIndex);

    return smallIndex;
}//end partition
```

Note that the formal parameters `first` and `last` specify the starting and ending indices, respectively, of the sublist of the `list` to be partitioned. If `first = 0` and `last = list.length - 1`, the entire list is partitioned.

As you can see from the definition of the method `partition`, certain elements of the list need to be swapped. The following method, `swap`, accomplishes this task. (Notice that this `swap` method is the same as the one given earlier in this chapter for the selection sort algorithm.)

```
private void swap(int [] list, int first, int second)
{
    int temp;

    temp = list[first];
    list[first] = list[second];
    list[second] = temp;
}//end swap
```

Once the list is partitioned into `lowerSublist` and `upperSublist`, we again apply the quick sort method to sort the two sublists. Because both sublists are sorted using the same quick sort algorithm, the easiest way to implement this algorithm is to use recursion. Therefore, this section gives the recursive version of the quick sort algorithm. As explained previously, after rearranging the elements of the list, the method `partition` returns the index of `pivot` so that the starting and ending indices of the sublists can be determined.

Given the starting and ending indices of a list, the following method, `recQuickSort`, implements the recursive version of the quick sort algorithm:

```
private void recQuickSort(int [] list, int first, int last)
{
    if (first < last)
    {
        int pivotLocation = partition(list, first, last);
        recQuickSort(list, first, pivotLocation - 1);
        recQuickSort(list, pivotLocation + 1, last);
    }
}//end recQuickSort
```

Finally, we write the quick sort method, `quickSort`, that calls the method `recQuickSort` on the original list.

```
public void quickSort(int[] list, int length)
{
    recQuickSort(list, 0, length - 1);
}//end quickSort
```

We leave it as an exercise for you to write a program to test the quick sort algorithm.