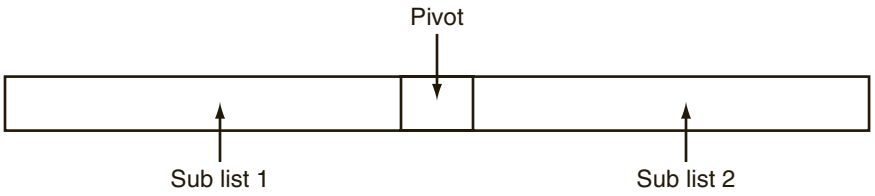
APPENDIX K

The QuickSort Algorithm

The QuickSort algorithm is a popular general-purpose sorting routine developed in 1960 by C. A. R. Hoare. It sorts an array by dividing it into two sub lists. Between the sub lists is a selected value known as the pivot. This is illustrated in Figure K-1.

Figure K-1 Sub lists and pivot



Notice in the figure that sub list 1 is positioned to the left of (before) the pivot, and sub list 2 is positioned to the right of (after) the pivot. Once a pivot value has been selected, the algorithm swaps the other values in the array until all the elements in sub list 1 are less than the pivot, and all the elements in sub list 2 are greater than the pivot.

Once this is done, the algorithm repeats the procedure on sub list 1, and then on sub list 2. The recursion stops when there is only one element in a sub list. At that point the array is completely sorted.

The algorithm is coded primarily in two methods: `quickSort` and `partition`. The `quickSort` method is recursive. Its psuedocode is shown here:

```
quickSort:
        If Starting Index < Ending Index
                Partition the List around a Pivot.
                quickSort Sub list 1.
                quickSort Sub list 2.
        End If.
```

Here is the Java code for the `quickSort` method:

```
public static void quickSort(int array[], int start, int end)
{
   int pivotPoint;

   if (start < end)
   {
      // Get the pivot point.
      pivotPoint = partition(array, start, end);

      // Sort the first sub list.
      quickSort(array, start, pivotPoint - 1);

      // Sort the second sub list.
      quickSort(array, pivotPoint + 1, end);
   }
}
```

This version of `quickSort` works with an array of integers. Its first argument is the array holding the list that is to be sorted. The second and third arguments are the starting and ending subscripts of the list.

The subscript of the pivot element is returned by the `partition` method. The `partition` method not only determines which element will be the pivot, but also controls the rearranging of the other values in the list. The method selects the element in the middle of the array as the pivot, and then scans the remainder of the array searching for values less than the pivot.

The code for the `partition` method is shown here:

```
private static int partition(int array[], int start, int end)
{
   int pivotValue, pivotIndex, mid;

   mid = (start + end) / 2;

   swap(array, start, mid);
   pivotIndex = start;
   pivotValue = array[start];
   for (int scan = start + 1; scan <= end; scan++)
   {
     if (array[scan] < pivotValue)
     {
        pivotIndex++;
        swap(array, pivotIndex, scan);
     }
   }
   swap(array, start, pivotIndex);

   return pivotIndex;

}
```

The `partition` method does not initially sort the values into their final order.

Its job is only to move the values that are less than the pivot to the pivot's left, and move the values that are greater than the pivot to the pivot's right. As long as that condition is met, they may appear in any order. The ultimate sorting order of the entire array is achieved cumulatively, though the recursive calls to the `quickSort` method.

There are many different ways of partitioning the array. As previously stated, the technique shown in this `partition` method selects the middle value as the pivot. That value is then moved to the beginning of the array (by swapping it with the value stored there). This simplifies the next step, which is to scan the array.

A `for` loop scans the remainder of the array, and when an element is found whose value is less than the pivot, that value is moved to a location left of the pivot point.

A third method, `swap`, is used to swap the values found in any two elements of the array. The method is shown here:

```
private static void swap(int[] array, int a, int b)
{
   int temp;

   temp = array[a];
   array[a] = array[b];
   array[b] = temp;
}
```

The program in Code Listing K-1 demonstrates these methods.

Code Listing K-1 **(QuickSortDemo.java)**

```
1  /**
2   *  This program demonstrates the QuickSort algorithm.
3   */
4
5  public class QuickSortDemo
6  {
7     /**
8      *  main method
9      */
10
11    public static void main(String[] args)
12    {
13       int[] array = { 7, 3, 9, 2, 0, 1, 8, 4, 6, 5 };
14
15       // Display the array as it is now.
16       System.out.println("Before the sort:");
17       for (int i = 0; i < 10; i++)
18          System.out.print(array[i] + " ");
19       System.out.println();
20
```

```
21        // Sort the array.
22        quickSort(array, 0, 9);
23
24        // Display the array again.
25        System.out.println("After the sort:");
26        for (int i = 0; i < 10; i++)
27           System.out.print(array[i] + " ");
28        System.out.println();
29     }
30
31     /**
32      *  The quickSort method uses the QuickSort algorithm to
33      *  sort array, from array[start] through array[end].
34      */
35
36     public static void quickSort(int array[], int start, int end)
37     {
38        int pivotPoint;
39
40        if (start < end)
41        {
42           // Get the pivot point.
43           pivotPoint = partition(array, start, end);
44
45           // Sort the first sub list.
46           quickSort(array, start, pivotPoint - 1);
47
48           // Sort the second sub list.
49           quickSort(array, pivotPoint + 1, end);
50        }
51     }
52
53     /**
54      *  The partition method selects the value in the middle of
55      *  the array as the pivot. The list is rearranged so all
56      *  the values less than the pivot are on its left and all
57      *  the values greater than pivot are on its right.
58      */
59
60     private static int partition(int array[], int start, int end)
61     {
62        int pivotValue, pivotIndex, mid;
63
64        mid = (start + end) / 2;
65
66        swap(array, start, mid);
67        pivotIndex = start;
68        pivotValue = array[start];
```

```
69          for (int scan = start + 1; scan <= end; scan++)
70          {
71             if (array[scan] < pivotValue)
72             {
73                pivotIndex++;
74                swap(array, pivotIndex, scan);
75             }
76          }
77          swap(array, start, pivotIndex);
78
79          return pivotIndex;
80       }
81
82       /**
83        *  The swap method swaps the element at array[a] with the
84        *  element at array[b].
85        */
86
87       private static void swap(int[] array, int a, int b)
88       {
89          int temp;
90
91          temp = array[a];
92          array[a] = array[b];
93          array[b] = temp;
94       }
95    }
```

Program Output

```
Before the sort:
7 3 9 2 0 1 8 4 6 5
After the sort:
0 1 2 3 4 5 6 7 8 9
```