Recursion in Java – Quicksort

Quicksort, like all of our previous sorts, uses a number of passes to achieve its goal of ordering an entire list of values. Like insertion sort, quicksort inserts one or more items into the list on each pass. Unlike insertion sort, however, quicksort inserts items into their *final position* in the list. Once an item has been inserted by quicksort, it never has to be moved again.

The sort is based on a simple idea: Consider a list that is already sorted (in ascending order). We can pick any value in the list, K, and we know that all values less than K are to the left, and all values greater than K will be to the right (assuming, for now, that all values are unique).

values less than K	K	values greater than K

Quicksort uses this concept and applies it to the unsorted list. Each pass of quicksort chooses a value for K from the unsorted list to be inserted into its correct location. This element is called a *pivot*. The pass then rearranges all other elements so they are either to the left of (less than) or to the right of (greater than) the pivot. All of these other values will still be out of order, but the pivot value, K, will now be in its final position.

This rearrangement is known as a *partition* of the list. To achieve a partition, we will use two markers to locate values that are on the wrong side (left/right) of the list, and then send them over to the other side. The following example will show the partition process.

61	46	12	63	52	91	27	55	74	14	71	37	87

First, we need to select a pivot value. With an unsorted list, any element will do, so we choose the first value in the list (61). A copy is made of this value to allow for swapping within the list.

61	61	46	12	63	52	91	27	55	74	14	71	37	87
pivot													

We also set variables to track the left and right side of the partition. All elements between these values still need to be checked against the pivot value (which is initially the entire list).

61	61	46	12	63	52	91	27	55	74	14	71	37	87
pivot	left												right

The right marker starts moving toward the centre of the list, looking for values that should not be on the right. The first value is checks, 87, is in the correct location. The next, 37, is not.

61	61	46	12	63	52	91	27	55	74	14	71	37	87
pivot	left											right	

Since 37 does not belong on the right, it is moved to the left, denoted by the position of the left marker (which also shows why we make a copy of the pivot to start, since that element has just had a new value assigned to it).

61	37	46	12	63	52	91	27	55	74	14	71	37	87
pivot	left											right	

Now the left marker will move toward the centre. Both the values 46 and 12 are less than 61, and belong on the left. At 63, another change in the list is required.

61	37	46	12	63	52	91	27	55	74	14	71	37	87
pivot				left								right	

Our first exchange moved a value from the right marker to the left marker. Now we move 63 from the left marker to the right marker.

61	37	46	12	63	52	91	27	55	74	14	71	63	87
pivot				left								right	

The process resumes from the right marker. The active marker will always switch after a swap occurs in the list, as follows:

61	37	46	12	63	52	91	27	55	74	14	71	63	87
pivot				left						right			

61	37	46	12	14	52	91	27	55	74	14	71	63	87
pivot						left				right			

61	37	46	12	14	52	91	27	55	74	91	71	63	87
pivot						left		right					

61	37	46	12	14	52	55	27	55	74	91	71	63	87
pivot								left/right					

In the final movement, the left marker moves toward the centre and ends up at the same location as the right marker. No further swaps are required, and the common marker location is the correct position for the pivot, which is now copied to that location.

37	46	12	14	52	55	27	61	74	91	71	63	87
							left/right					

With the pivot now in the correct location, we need to address the two sublists, one on either side of the pivot. To accomplish this, we will apply our partitioning technique recursively to each sublist.

For each sublist, we will continue to use the leftmost item as the pivot. In our example, the pivots for the left and right sublist will be 37 and 74, respectively. If we apply the partitioning to each sublist, the result will be a list with four partitions, and the values 37, 74, and 61 will all be in their correct locations.

27	14	12	37	52	55	46	61	63	71	74	91	87

Recursion in Java – Quicksort

The methods that follow, both called quickSort, work together to implement quicksort to sort an array of integer values. They work regardless of whether or not there are repeated items in the array.

```
public static void quickSort (int[] list)
{
  quickSort(list, 0, list.length - 1);
}

public static void quickSort (int[] list, int low, int high)
{
  final int MOVING_LEFT = 0;
  final int MOVING_RIGHT = 1;

  if (low < high)
  {
    int left = low;
    int right = high;
    int currentDirection = MOVING_LEFT;
    int pivot = list[low];

    while (left < right)
    {
      if (currentDirection == MOVING_LEFT)
      {
        while ((list[right] >= pivot) && (left < right))
          right--;

        list[left] = list[right];
        currentDirection = MOVING_RIGHT;
      }
      if (currentDirection == MOVING_RIGHT)
      {
        while ((list[left] <= pivot) && (left < right))
          left++;

        list[right] = list[left];
        currentDirection = MOVING_LEFT;
      }
    }
    list[left] = pivot; // or list[right] = pivot, since left == right
    quickSort(list, low, left-1);
    quickSort(list, right+1, high);
  }
}
```

Does quicksort deserve the name? In general, it does. On average, it is faster than any of the other sorts we have examined so far. It does, however, have its negative characteristics. For short lists, it is more trouble than it is worth – the recursive calls require too much system overhead. In such cases it might be better to use shellsort, or for very short lists, even selection or insertion sort.

In addition, much of the efficiency and speed of this method is based upon the assumption that the pivot value will effectively split the list into two equal sublists, although we have no assurance that this will actually occur. As the number of bad partitions becomes large, the efficiency of quicksort decreases rapidly. There are techniques for avoiding bad partitions, but they are beyond the scope of this course.