Merge Sort O(n log(n)) for all cases

The Merge Sort uses the **divide-and-conquer** approach. It begins by placing **each** element into its own individual list. Then each pair of adjacent lists is combined into one **sorted** list. This continues until there is one big, final, sorted list. The process is illustrated below:

Put each element into its own list of one element.	72	83	40	90	51	30	18	75
Merge every two lists above into a single sorted list.	72,83		40,90		30,51		18,75	
Merge every two lists above into a single sorted list.	40,72,83,90				18,30,51,75			
Merge the two lists above into the final sorted list.	18,30,40,51,72,75,83,90							

The above, however, is a very simplistic approach. In reality, the merge sort is often implemented **recursively** as illustrated with the following code:

A Merge Sort method:

```
//Enter this method with left = the beginning index (initially 0) and right = the last index
//(initially a.length-1)
public static void sort (int a[ ], int left, int right)
{
        if (right = = left) return;
        int middle = (left + right) /2; //salient feature #1
        sort(a, left, middle); //salient feature #2 (recursion)
        sort(a, middle + 1, right); //salient feature #3
        merge(a, left, middle, right); //salient feature #4
}
```

//…see two pages forward for the *merge* method, an important component of this sorting technique.

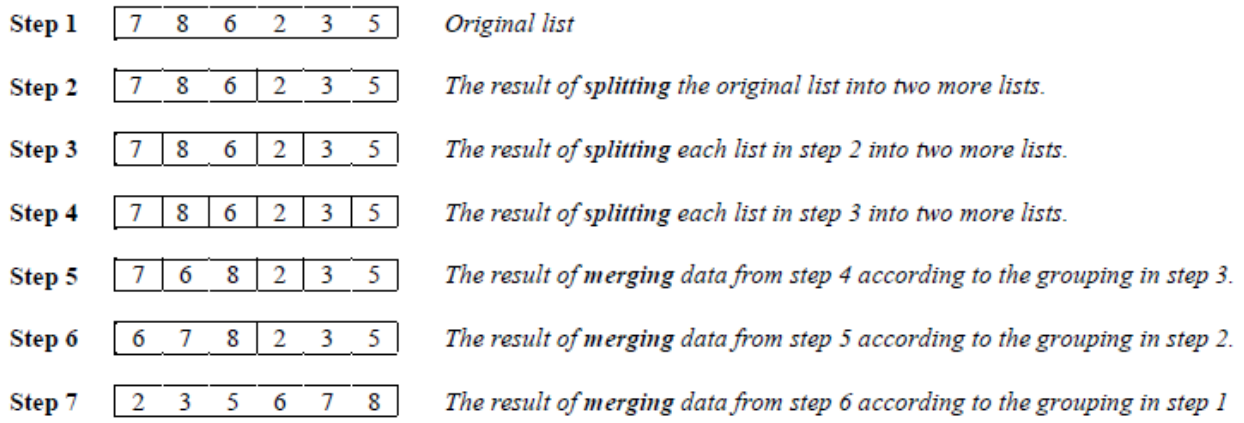
How it works:

The recursive calls to *sort* and the resulting recalculation of *middle = (left + right) / 2* continually subdivide the lists until we get individual "lists" of one element each. Due to the nature of recursion, that subdivision process continues until we reach the final lists of one element each **before** the *merge* method actually begins merging the lists together, two at a time.

Subdividing the list:

On the following page we see this process of recursively subdividing and subsequent

merging of the lists. We will consider the following original order of some numbers to be sorted:



Notice above that the **all splitting** processes **occur first** until after Step 4 at which time all "lists" consist of single elements. At that point successive **merging** of all the lists takes place according to how the split steps were originally done, but in reverse order.

Salient features #2 and #3 pointed out in the code on the previous page are the result of recursion. This corresponds to Steps 2-4 above in which a hierarchy of unordered lists are produced as a result of splitting previous lists. Steps 5-7 implement salient feature #4 which is the **reverse order merging** of those lists.

The *merge* method (used by Merge Sort):

```
private static void merge(int a[ ], int left, int middle, int right)
{
        //This temporary array will be used to build the merged list
        int tmpArray[ ] = new int[right – left +1];
        //This creation of a temporary array is a BIG feature of the merge sort.

        int index1 = left;
        int index2 = middle + 1;
        int indx = 0;

        //Loop until one of the sublists is finished, adding the smaller of the first
        //elements of each sublist to the merged list.
        while (index1 <= middle && index2 <= right)
        {
                if ( a[index1] < a[index2] )
                {
                        tmpArray[indx] = a[index1];
                        index1++;
                }
                else
```

```
                {
                        tmpArray[indx] = a[index2];
                        index2++;
                }
                indx++;
        }

        //Add to the merged list the remaining elements of whichever sublist is
        //not yet finished
        while(index1 <= middle)
        {
                tmpArray[indx] = a[index1];
                index1++;
                indx++;
        }

        while(index2 <= right)
        {
                tmpArray[indx] = a[index2];
                index2++;
                indx++;
        }

        //Copy the merged list from the tmpArray array into the a array
        for (indx = 0; indx < tmpArray.length; indx++)
        {
                a[left + indx] = tmpArray[indx];
        }
}
```

Used in *Array.sort( )*:

The Quick Sort is used on primitive arrays passed to *Arrays.sort( )*; however, the Merge Sort is used on object arrays.