Merge Sort

After looking at straightforward sorting algorithms such as Insertion Sort and Selection sort, one might ask if there is a more clever, quicker way to sort numbers that does not require looking at most possible pair of numbers. (Perhaps we can gather extra information from a comparison that renders other comparison's that could have been done useless.) In this class we will utilize the concept of recursion to come up with a couple more efficient algorithms.

One of the more clever sorting algorithms is merge sort. Merge sort utilizes recursion and a clever idea in sorting two separately sorted arrays.

The Merge

The merging problem is one that is more simple than sorting an unsorted array, and one that will be a tool we can use in Merge Sort.

The problem is that you are given two arrays, each of which is already sorted. Now, your job is to efficiently combine the two arrays into one larger one which contains all of the values of the two smaller arrays in sorted order.

The essential idea is this:

1) **Keep track of the smallest value in each array that hasn't been placed in order in the larger array yet.**
2) **Compare these two smallest values from each array. One of these must be the smallest of all the values in both arrays that are left. Placed the smallest of the two values in the next location in the larger array.**
3) **Adjust the smallest value for the appropriate array.**
4) **Continue this process until all values have been placed in the large array.**

This should look amazingly similar to the Sorted List Matching Algorithm we looked at last time. The same principle is in use here: because we are dealing with two sorted lists, we can streamline our job. This saves us comparisons.

Illustration of Merge Algorithm

Here is an illustration of an algorithm to do a merge. (It's easier to understand with pictures instead of pseudocode.)

2	**7**	**16**	**44**	**55**	**89**

1	**6**	**9**	**13**	**15**	**49**

Here is what happens after the first step:

1											

2	**7**	**16**	**44**	**55**	**89**

min A

	6	**9**	**13**	**15**	**49**

min B

Here is what happens after the second step:

1	**2**										

	7	**16**	**44**	**55**	**89**

min A

	6	**9**	**13**	**15**	**49**

min B

Here is what happens after the third step:

1	**2**	**6**									

	7	**16**	**44**	**55**	**89**

min A

		9	**13**	**15**	**49**

min B

As you can see, when we are done, our large array will be in sorted order, like so:

1	**2**	**6**	**7**	**9**	**13**	**15**	**16**	**44**	**49**	**55**	**89**

Now, the big question is how can we use this to sort an entire array, since this would only sort a specific type of array, where the first half and second half of the array were already in sorted order.

Here is the main idea for merge sort:

1) **Sort the first half of the array, using merge sort.**
2) **Sort the second half of the array, using merge sort.**
3) **Now, we do have a situation to use the Merge algorithm! Simply merge the first half of the array with the second half.**

So, this points to a recursive solution.

You might ask, "But how do we know that Merge Sort is going to work on both halves of the array?" The answer is that in each call to merge sort, you must run the Merge method on some two parts of the array. All of the actual sorting gets done in the Merge method.

Let's demonstrate how this algorithm is going to work before looking at the code to implement it.

// This is a method used to perform a merge sort on the instance variable array.
// array[start1..start2-1] and array[start2..end2] must both be sorted sub-arrays.
// When completed array[start1…end2] will be a sorted sub-array.

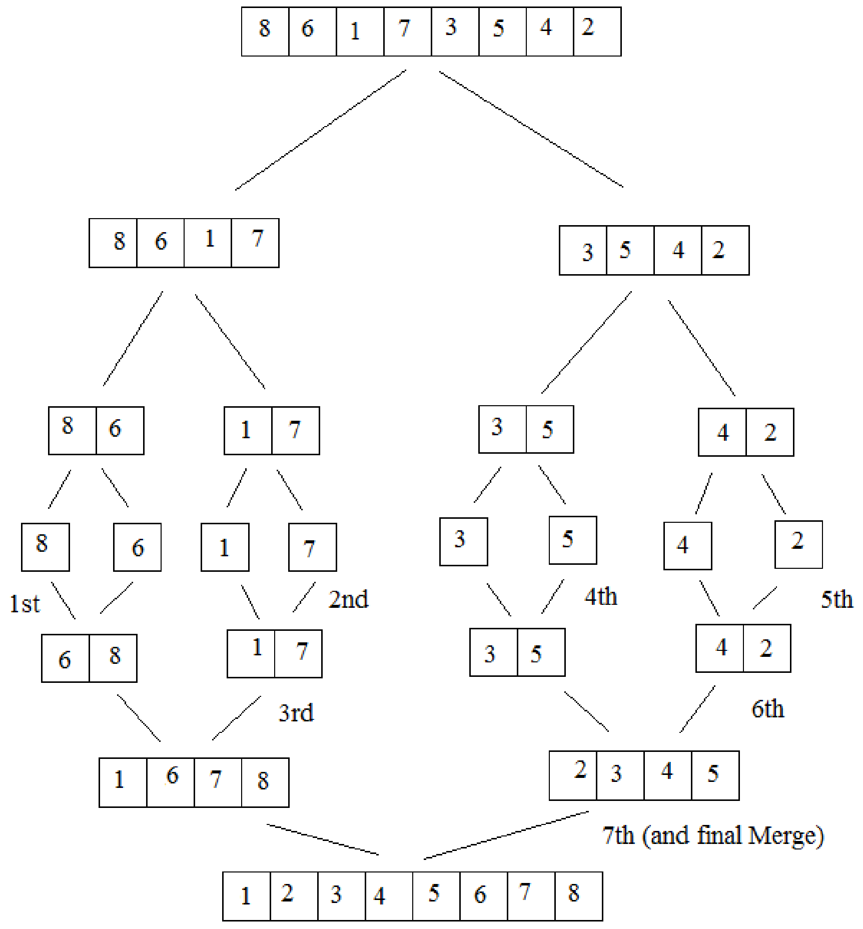
```
private void merge(int start1, int start2, int end2) {

   int end1 = start2-1;
   int[] tmp = new int[end2-start1+1];

   int i = start1, j = start2;

   int curI = 0;
   while (i < start2 || j <= end2) {

       if (i == start2) {
           tmp[curI] = array[j];
           j++;
       }
       else if (j == end2+1) {
           tmp[curI] = array[i];
           i++;
       }
       else if (array[i] < array[j]) {
           tmp[curI] = array[i];
           i++;
       }
       else {
           tmp[curI] = array[j];
           j++;
       }

       curI++;
   }

   for (i=0; i<tmp.length; i++)
       array[i+start1] = tmp[i];

}

public void mergeSort() {
   mergeSortRec(0, array.length-1);
}

private void mergeSortRec(int low, int high) {
   if (low < high) {
       int mid = (low+high)/2;
       mergeSortRec(low, mid);
       mergeSortRec(mid+1, high);
       merge(low, mid+1, high);
   }
}
```

Merge Sort Analysis - Summary

If we look at the structure of the recursion, separate from the order in which merges actually occur, we get a tree like structure. Consider running a merge sort on the following 8 elements:

8	**6**	**1**	**7**	**3**	**5**	**4**	**2**

Our tree structure looks like this:



Notice that at each "level" of the tree we do roughly n steps in several Merges, where n is the original array length. Thus, to calculate a total run time, we just need to know how many levels

there are in this computation tree. Since we divide by 2 at each level, let k equal the number of times we divide until we get to a row of all individual cells. We get the following equation:

$$\frac{n}{2^k} = 1$$
$$n = 2^k$$
$$k = log_2 n$$

It follows that the total number of steps in Merge Sort is roughly n x log$_2$n (times a constant). Thus, Merge Sort runs in O(nlgn) time for an array of size n.