**Merge Sort**

**Like [QuickSort](http://quiz.geeksforgeeks.org/quick-sort/), Merge Sort is a**[**Divide and Conquer**](http://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/)**algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one**.

**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.**

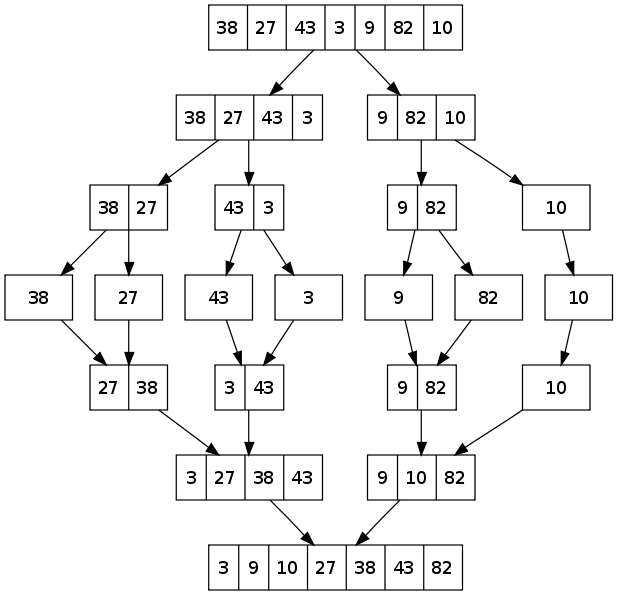
**The problem is that you are given two arrays, each of which is already sorted. Now, your job is to efficiently combine the two arrays into one larger one which contains all of the values of the two smaller arrays in sorted order.**

**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**

****

**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.**

1. **Sort the second half of the array, using merge sort.**
2. **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.**

**Merge Sort Analysis**

**Here are the steps of Merge Sort:**

**1) Merge Sort the first half of the list**

**2) Merge Sort the second half of the list**

**3) Merge both halves together.**

**Let T(n) be the running time of Merge Sort on input of size n. Then we have**

**T(n) = (Time in step 1) + (Time in step 2) + (Time in step 3)**

**Noticing that step 1 and step 2 are sorting problems also, but of size n/2, and that the last step runs in O(n) time, we get the following equation for T(n):**

**T(n) = T(n/2) + T(n/2) + O(n)**

**= 2T(n/2) + O(n)**

**This is known as a recurrence relation since the function T(n) is defined in terms of another value of the function T. Now, let's see if we can try to figure out what T(n) is, just in terms of n, (for the time being, let's simplify O(n) to n):**

**T(n) = 2T(n/2) + n**

**T(n) = 2[2T(n/4)+n/2] + n**

**= 4T(n/4) + 2n**

**= 4[2T(n/8)+n/4] + 2n**

**= 8T(n/8) + 3n**

**Hopefully, by this point you can see a pattern and realize that after the kth application of the formula you will find that**

**T(n) = 2kT(n/2k) + kn**

**Eventually, when applying this recurrence, we should stop. In particular, we can assume that T(1) = 1. Then, we can solve for T(n) directly by plugging in k = log2n. To see why this works, note that we know what T(1) is. Also, we have T(n/2k) in our formula. So it would be nice if n = 2k. But this occurs when k = log2n. Plugging in the value for k we find:**

**T(n) = nT(1) + nlog2n**

**= O(nlog2n)**

**Another way to derive this result is the following:**

**Take our original equation,**

**T(n) = 2T(n/2) + n and divide through by n:**

**T(n)/n = 2T(n/2)/n + n/n, so**

**T(n)/n = T(n/2)/(n/2) + 1**

**Now, we can use the equation directly above and plug in n, n/2, n/4, etc. and see what we get:**

**T(n)/n = T(n/2)/(n/2) + 1**

**T(n/2)/(n/2) = T(n/4)/(n/4) + 1**

**T(n/4)/(n/4) = T(n/8)/(n/8) + 1**

**...**

**T(2)/2 = T(1)/1 + 1**

**Now, add all of these equations together, to create a new large equation. Notice how most of the right hand side and left hand sides are nearly identical. Virtually everything crosses out. When we simplify that equation, we get**

**T(n)/n = T(1)/1 + 1 + 1 ... + 1**

**T(n)/n = 1 + log2n**

**T(n) = n(1 + log2n)**

**Thus, T(n) = O(nlog2n).**

**For recurrences like the one above, there is a general plug-n-chug formula. It is as follows. For the recurrence relation**

**T(n) = AT(n/B) + O(nk), where A, B and k are constants, we have**

**O(n^(logBA)), if A > Bk**

**T(n) = O(nk(log n)), if A = Bk**

**O(nk), if A < Bk.**

**Here are some examples worked out:**

**Recurrence Rel. Case Answer**

**T(n) = 3T(n/2) + O(n2) 3 O(n2)**

**T(n) = 4T(n/2) + O(n2) 2 O(n2log n)**

**T(n) = 9T(n/2) + O(n3) 1 O(n^(log29))**

**T(n) = 6T(n/3) + O(n2) 3 O(n2)**

**T(n) = 5T(n/5) + O(n) 2 O(nlog n)**

**All the recursive code covered will be included under code samples.**