# Merge Sort

**In both of Selection Sort and Insertion Sort, we end up making a significant number of possible comparisons and swaps between elements. Perhaps it's possible to use information from certain comparisons to reduce the number of future comparisons.**

**So, 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.**

**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.**

# 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**

$$T(n) = 2^kT(n/2^k) + kn$$

**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 $k = \log_2 n$. To see why this works, note that we know what T(1) is. Also, we have $T(n/2^k)$ in our formula. So it would be nice if $n = 2^k$. But this occurs when $k = \log_2 n$. Plugging in the value for k we find:**

$$T(n) = nT(1) + n\log_2 n$$
$$= O(n\log_2 n)$$

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

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

$$T(n) = \begin{cases} O(n^{(\log_B A)}), \text{ if } A > B^k \\ O(n^k(\log n)), \text{ if } A = B^k \\ O(n^k), \text{ if } A < B^k. \end{cases}$$

**Here are some examples worked out:**

| Recurrence Rel. | Case | Answer |
|---|---|---|
| $T(n) = 3T(n/2) + O(n^2)$ | 3 | $O(n^2)$ |
| $T(n) = 4T(n/2) + O(n^2)$ | 2 | $O(n^2 \log n)$ |
| $T(n) = 9T(n/2) + O(n^3)$ | 1 | $O(n^{(\log_2 9)})$ |
| $T(n) = 6T(n/3) + O(n^2)$ | 3 | $O(n^2)$ |
| $T(n) = 5T(n/5) + O(n)$ | 2 | $O(n \log n)$ |