# Merge Sort

Go to <http://math.hws.edu/eck/js/sorting/xSortLab.html> to watch the sorts in action.

The Merge sort is the first of our O(n log n) sorts. It creates smaller and smaller sublists by recurring on successive halves of the array, then zipper-merging each successive pair of sublists. The last zipper-merge produces the sorted list.

**private static void** mergeSortHelper(**double**[] array,

**double**[]copyBuffer, **int** low, **int** high)  
 {  
 **if** (low < high)  
 {  
 **int** middle = (low + high) / 2;  
 mergeSortHelper(array, copyBuffer, low, middle);  
 mergeSortHelper(array, copyBuffer, middle + 1, high);  
 merge(array, copyBuffer, low, middle, high);  
 }  
 }

Since low and high are moving towards each other, the sublists are successively cut in half, until we reach sublists of length one. By definition, sublists of length one are already sorted. As the recursion unwinds, the zipper-merge methods on each level merge and order the items of successively larger pairs of sublists. The zipper-merge method does the actual work of ordering the items in the array.

# Trace the Merge Sort

|  |  |
| --- | --- |
| the method calls | the data after the call |
| mergeSortHelper()mergeSortHelper()mergeSortHelper()mergeSortHelper()mergeSortHelper()mergeSortHelper()mergeSortHelper()mergeSortHelper()mergeSortHelper()etc. | $[3, 1, 4, 1, 5, 9, 2, 6]$ |
| merge [3, 1] | $[1, 3, 4, 1, 5, 9, 2, 6]$ |
| merge [4, 1] | $[1, 3, 1, 4, 5, 9, 2, 6]$ |
| merge [1, 3, 1, 4] | $[1, 1, 3, 4, 5, 9, 2, 6]$ |
| merge [5, 9] | $[1, 1, 3, 4, 5, 9, 2, 6]$ |
| merge [2, 6] | $[1, 1, 3, 4, 5, 9, 2, 6]$ |
| merge [5, 9, 2, 6] | $[1, 1, 3, 4, 2, 5, 6, 9]$ |
| merge [1, 1, 3, 4, 2, 5, 6, 9] | $[1, 1, 2, 3, 4, 5, 6, 9]$ |

Important! Merging two lists **must** be done using a single pass. That’s why it is called a “zipper merge.” Do **not** use nested loops! The merge method should use one loop, either for or while, running between low and high, which of course are different in each recursive level. Since you know the sub-lists are already sorted, you need **one loop** to compare values (index i1 and i2) at the front of each (low and middle) and copy the smaller value. The index of the smaller item is then moved to the next cell. Study this diagram of the merge method running on the last 8 items of an array with 16 items:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| array |  |  |  |  |  |  |  |  | 1 | 1 | 3 | 4 | 2 | 5 | 6 | 9 |
|  |  |  |  |  |  |  |  |  | low    i1 | | | | middle high    i2 | | | |
| copybuffer |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

The algorithm diagrammed above uses a temporary array copybuffer, which each time has to be copied (using a second, but not a nested loop) from low to high back to array. Some students prefer to write a zipper-merge method that works directly on the array.

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 | 8 | 2 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Exercises

# Suppose all the recursive calls of the Merge Sort have been completed on this array. None of the zipper-merge methods have run. What does the array look like after one, two, and three of the zipper-merge methods have run?

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 5 | 3 | 2 | 9 | 7 | 5 | 1 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

# Suppose all the recursive calls of the Merge Sort have been completed. None of the zipper-merge methods have run. What does the array look like after one, two, three, and four of the zipper-merge methods have run?

1. In the Merge Sort, 8 items of data takes 3 levels of recursion. For *n* items of data (in random order), how many levels of recursion do you need? \_\_\_\_\_\_\_\_\_ What is the Big-O of each zipper-merge? \_\_\_ Therefore, what is the Big-O of the Merge Sort (in the average case)? **\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Here is an array already in sorted order (ascending order):

# How many comparisons does the Merge Sort make to sort this array in ascending order? \_\_\_\_\_\_

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

# Here is an array in reverse order (descending order):

# How many comparisons does the Merge Sort make to sort this array in ascending order? \_\_\_\_\_\_

# Know thy Big-O: the Merge Sort is \_\_\_\_\_\_\_\_\_ in the best case, \_\_\_\_\_\_\_\_\_ in the average case, and \_\_\_\_\_\_\_\_\_ in the worst case.

# Lab Assignment

Parts of the MergeSort\_Driver are written for you. Parts of MergeSort are written for you. Implement the rest of the program to sort doubles, and again to sort Comparables.

**Extension**

As a last refinement, make your Merge Sort (on doubles only) use a Selection Sort (it’s faster on small lists) when the sub-lists get smaller than 10. If you do this, you will have to uncomment lines 54-56 as well as completing SelectionLowHigh to sort between the arguments low and high.

Extra Reading: Lambert & Osborne (aka Orange Book): Chap 12.5, pp 482 – 485