**Merge Sort**

Merge sort is too complicated for you to be able to look at the visual representation or the code and understand it. We are going to have to eat an elephant one bite at a time.  
  
**Part 1 - High Level Merge Method:** The first thing we want to grasp is the high level of the “merge” function in merge sort. Below is the method signature.

|  |
| --- |
| **public** **static** **void** merge(**int**[] array, **int**[] helper, **int** from, **int** mid, **int** to) {}  **public** **static** **void** testMerge() {  **int**[] array = **new** **int**[]{1,3,5,7,2,4,6,8}; // Array with two sorted sub arrays  **int** from = 0; // decide which indexes do you want to sort  **int** secondSortedSubArrayStart = 4; // index where second sorted subarray starts  **int** to = array.length - 1; // decide which indexes do you want to sort  *merge*(array, **new** **int**[array.length], from, secondSortedSubArrayStart, to);  System.*out*.println(Arrays.*toString*(array));//Prints sorted array  } |

This function takes an array that contains two sorted sub arrays, and sorts the array. For example, you look at the test for it above. The array I initialized contains two sorted sub arrays {1, 3, 5, and 7} and {2, 4, 6, 8}. What this tells us is that we don’t need to use our traditional sorting algorithms. We can use an efficient method to sort it that makes use of the fact that we know that the left and right side is sorted, but the entire array is not sorted. We need to specify at which point does the second sorted sub array starts (which is four in this case). Finally we throw in a helper array which is empty that is used for temporary storage. The reason we throw it in the parameter and not initialize it in the method is because this method will be called a lot using recursion. We want the same storage to be passed around and not have a bunch of storage memory be initialized.

|  |
| --- |
| **Modifying To and From In Merge:** If we change ‘from’ to 2 and ‘to’ to 6, this will sort indexes 2-6 and leave indexes 0,1 and 8 untouched.  **public** **static** **void** main(String[] args) {  **int**[] array = **new** **int**[]{1,3,5,7,2,4,6,8};  **int** secondSortedSubArrayStart = 4;  *merge*(array, **new** **int**[array.length], 2, secondSortedSubArrayStart, 6);  System.*out*.println(Arrays.*toString*(array));//Prints sorted array  }  Yields: [1, 3, 2, 4, 5, 6, 7, 8]//The 1,3 & 8 remain while the rest is sorted |

**Part 2 – Implementation of Merge Method**: There are four parts to the implementation of this merge method. All these four parts are fairly simple.

1. Copy the contents in array into the empty helper array.
2. Initialize three pointers. One iterates over the first/left sorted sub array in helper. The other iterates over the second/right sorted sub array in helper. The last iterates over the entire array in the original array.
3. While you haven’t iterated through the entire array, compare the item in the left sub array in helper and the item in the right sub array in helper. Copy whichever is smaller into the array. Increment the pointers.

|  |
| --- |
| **public** **static** **void** merge(**int**[] array, **int**[] helper, **int** from, **int** secondSubArrayStart, **int** to){  *arrayCopy*(helper, array, from, to);  **int** helperIteratorL = from;  **int** helperIteratorR = secondSubArrayStart;  **int** arrayIterator = from; //Remember its from, to! From comes first!  **while** (!*mergeComplete*(arrayIterator, to)) {  **if** (*leftItemIsSmaller*(helperIteratorL, helperIteratorR, helper, secondSubArrayStart, to)){  array[arrayIterator] = helper[helperIteratorL];  helperIteratorL++;  } **else** {  array[arrayIterator] = helper[helperIteratorR];  helperIteratorR++;  }  arrayIterator++;  }  }  } |

|  |
| --- |
| **public** **static** **void** arrayCopy(**int**[] dest, **int**[] src, **int** fromIndex, **int** toIndex){  **for** (**int** i = fromIndex; i <= toIndex; i++) {  dest[i] = src[i];  }  }  **public** **static** **boolean** mergeComplete(**int** arrayIterator, **int** to){  **return** arrayIterator > to; //we sorted the part we want to sort  }  **public** **static** **boolean** leftItemIsSmaller(**int** helperIteratorL, **int** helperIteratorR,  **int**[] helper, **int** secondSubArrayStart, **int** to){  **if**(helperIteratorL >= secondSubArrayStart){  **return** **false**; //already iterated through 1st subarray  }**else** **if**(helperIteratorR > to){  **return** **true**;//already iterated through 2nd subarray  }**else**{  **return** (helper[helperIteratorL] <= helper[helperIteratorR]);  }  } |

**Part 3 – Time and Space Complexity of the Merge Method:**

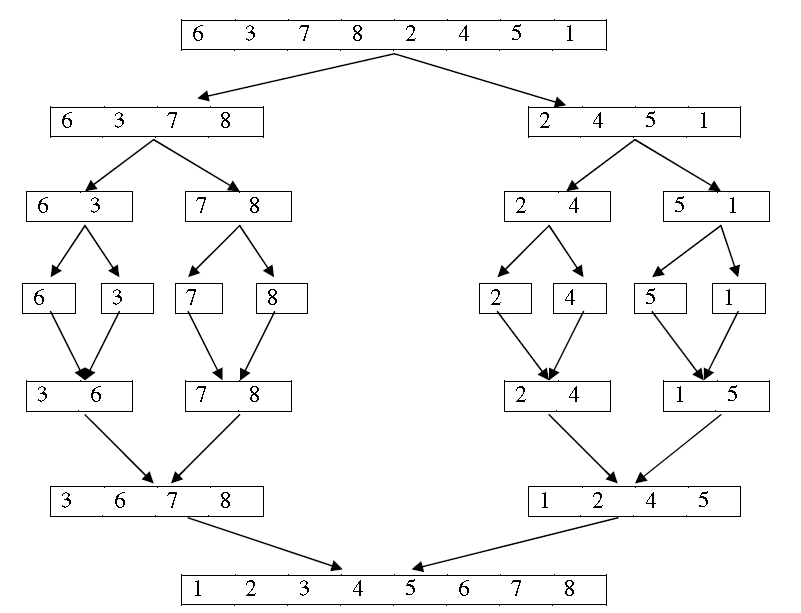
So we have built this merge method that given two sorted sub arrays, we can sort it in O(n) time (we pass through the array once and the helper array once, which is still linear time) and O(n) space (since we need a helper array that is the same size of the original array to use this method).

**Part 4 – Merge Sort using the merge method:**

|  |
| --- |
| **public** **static** **void** mergesort(**int**[] array) {  **int**[] helper = **new** **int**[array.length];  *mergesort*(array, helper, 0, array.length - 1);  }  **public** **static** **void** mergesort(**int**[] array, **int**[] helper, **int** low, **int** high) {  **if** (low < high) {  **int** mid = (low + high) / 2;  *mergesort*(array, helper, low, mid); // Sort left half  *mergesort*(array, helper, mid+1, high); // Sort right half  *merge*(array, helper, low, mid + 1, high); // Merge them  }  } |

This is very similar to the binary search method. Suppose you have the array {8,7,6,5,4,3,2,1}. Merge sort would call merge sort on the first half {8,7,6,5} which would sort it to {5,6,7,8} and then it would call merge sort on the second half {4,3,2,1} and then it would sort it to {1,2,3,4} and then the merge method would merge {5,6,7,8,1,2,3,4} because it is an array of two sorted sub arrays. Note that you pass (mid + 1) into merge because it is looking for the second sorted sub array start which you can see the second sub array is from (mid + 1, high). Practice this part because it is easy to confuse.

**Part 5 – Time Complexity of Merge Sort:**



1. You start of repeatedly dividing the data sets in two, until you have a bunch one element subarrays at row 4. This happens when you get the midpoint (constant time) and then you call merge sort on the left and right parts of it. The ‘splitting’ of an array to two sub arrays takes constant time because all you have to do is compute the midpoints at each row. If you have a data set of size 8, you have to divide (or calculate midpoints at C1 constant time) a total of 7 times. If your original data set was of size 4, you would divide a total of 3 times. So overall, the repeated divides (first, second, third and third rows) take a total of

O(dividing part) = ( n – 1 divides) \* (C1 = constant time/steps to get midpoint to divide) = C1 (n -1).

1. At each merging row, the complexity is O(n). So this problem has 3 merging rows, so its complexity is O(3n). Below is the proof of this.

* At row 4, there are 8 sub arrays, each of size 1 that get merged into 4 sub arrays each of size two. There are 4 merge calls (each one merges two single elements into a sub array with two elements). Although there are are 4 merges, each merge contains a quarter of the original array. So it is 4 × (n/4) = n.
* At the fifth row, there are 4 groups of sub arrays each of size two that get merged into two groups of sub arrays each of size four. Although there are only 2 merges, each merge contains a half of the original array. So it is 2 × (n/2) = n.
* At the sixth row, there are 2 groups of sub arrays each of size four that get merged into one sub array of size 8. Although there is only 1 merge, that merge contains the entire original array so it is O(n).

1. Every time you double your data set, you add an extra divide row, and an extra merge row. So the O(merging part) = ( n) \* ( # of merge rows) = (n) \* ( log n).

Big Oh of merge sort = (Big Oh of dividing part) + (big Of of Merging Part)

= C1 (n -1) + C \* (n) \* ( log n) = C1(n – 1) + (n log n) = n log n.

**Part 6 – Space Complexity of Merge Sort:** The entire time, through all those recursive calls, merge calls, you only use the same helper array which is of size n. So the only additional space you need is O(n).

**Complete Code:**

|  |
| --- |
| **public** **class** MergeSort {  **public** **static** **void** mergesort(**int**[] array) {  **int**[] helper = **new** **int**[array.length];  *mergesort*(array, helper, 0, array.length - 1);  }  **public** **static** **void** mergesort(**int**[] array, **int**[] helper, **int** low, **int** high) {  **if** (low < high) {  **int** mid = (low + high) / 2;  *mergesort*(array, helper, low, mid); // Sort left half  *mergesort*(array, helper, mid+1, high); // Sort right half  *merge*(array, helper, low, mid+1, high); // Merge them  }  }  **public** **static** **void** merge(**int**[] array, **int**[] helper, **int** from, **int** secondSubArrayStart, **int** to) {  *arrayCopy*(helper, array, from, to);    **int** helperIteratorR = secondSubArrayStart;  **int** helperIteratorL = from;  **int** arrayIterator = from;    **while**(!*mergeComplete*(arrayIterator, to)){  **if**(*leftItemIsSmaller*(helperIteratorL, helperIteratorR,helper, secondSubArrayStart,to)){  array[arrayIterator] = helper[helperIteratorL];  helperIteratorL++;  }**else**{  array[arrayIterator] = helper[helperIteratorR];  helperIteratorR++;  }  arrayIterator++;  }  }  **public** **static** **void** arrayCopy(**int**[] dest, **int**[] src, **int** fromIndex, **int** toIndex){  **for**(**int** i = fromIndex; i <= toIndex; i++){  dest[i] = src[i];  }  }  **public** **static** **boolean** mergeComplete(**int** arrayIterator, **int** to){  **return** arrayIterator > to;  }    **public** **static** **boolean** leftItemIsSmaller(**int** helperIteratorL, **int** helperIteratorR,  **int**[] helper, **int** secondSubArrayStart, **int** to){  **if**(helperIteratorL >= secondSubArrayStart){  **return** **false**; //already iterated through 1st subarray  }**else** **if**(helperIteratorR > to){  **return** **true**;//already iterated through 2nd subarray  }**else**{  **return** (helper[helperIteratorL] <= helper[helperIteratorR]);  }  }  } |