For the second problem, we are asked to make an in place merge sort. The in place sorting is much more complex than the sorting from Problem 1. For an in place merge sort, the function will take the **inputted array, the first element, and the last element** as the positions it will compare too. The function mergeSort will test if the last and first (if there are any elements in the array to sort).

In this merge sort, we will need to use the swap function several times if the compared values are not in order.

**public** **static** **void** swap(**int**[] arr, **int** index1, **int** index2) {

// swap 2 number with XOR bit Operator

arr[index1] = arr[index1] ^ arr[index2];

arr[index2] = arr[index1] ^ arr[index2];

arr[index1] = arr[index1] ^ arr[index2];

}

mergeSort uses an if-else statement to check what kind of values are in the array. It will first check if there are any values in the first and last nodes of the array. If there are only 2 values in the array, it will compare if one is greater than or less than, and swap accordingly. Code below:

**if** (0 == last - first) {

// one element in an array - do nothing

**return**;

}

**else** **if** (1 == last - first) {

// two elements in an array - swap if needed

**if** (arr[first] > arr[last])

swap(arr, first, last);

}

Otherwise, the if-else statement will end up with:

**else** {

// Array size >= 3

**int** mid = (first + last) / 2;

mergeSort(arr, first, mid);

mergeSort(arr, mid+1, last);

merge(arr, first, last, mid);

}

In this case, we will redefine the mid of the array, so we can sort the first half, then the second half of the array. The loop will continue to split the array from the middle over and over again. By doing this, we can do the merge sort by an array of 2n where n will increase as they are merged together after the sort.

Inside the merge function, we use a for loop to test if the next element in an array is greater than the element in the array. If it is, we implement the swap function. And then test the next 2 elements in the array with mergeRight.

**public** **static** **void** merge(**int**[] arr, **int** start, **int** last, **int** mid) {

**for** (**int** idx = start; idx <= mid; idx++) {

**if** (arr[idx] > arr[mid+1]) {

swap(arr, idx, mid+1);

mergeRight(arr, mid+1, last);

}

}

}

mergeRight will take the next inputs of the array and compare them. If the values need to be swapped, they are using the swap function.

**static** **void** mergeRight(**int**[] arr, **int** start, **int** last) {

**for** (**int** idx = start; idx < last; idx++) {

**if** (arr[idx] > arr[idx+1])

swap(arr, idx, idx+1);

}

}

The acutal mergeSort function that will be called in the main file is:

**public** **static** **void** mergeSort(**int**[] arr, **int** arraySize) {

arraySize = arr.length;

*mergeSort*(arr, 0, arraySize-1);

}

A merge sort is always stable: it will compare 2 elements at a time, but when putting the values back into the array, it will do so in order, instead of moving values in the array over and over again in a loop.