# TEACHER’s guide

# Merge and MergeSort

**OBJECTIVES:** The student will write a function to merge two sorted lists into one sorted list.

The student will write a recursive mergeSort routine to sort a list in non-decreasing order.

**ACTIVITIES/TIME:** Four Days:

Days one-two: Presentation - 50 minutes each day

Days three-four: Lab Assignment – 50 minutes each day

**MATERIALS:** Student Lesson A18: *Merge and MergeSort*

Lab Assignment A18.1, *Merge*

Lab Lab Assignment A18.1, Starter Code, *MergeTemplate.java*

Lab Assignment A18.2, *Recursive MergeSort*

Transparency A18.1*, MergeSort Example*

Transparency A18.2, *Merging Two Lists*

Worksheet A18.1, *Non-recursive Merge and MergeSort*

Teacher’s Guide A18: *Merge and MergeSort*

Lab Assignment A18.1 – Answers, *Merge.java*

Lab Assignment A18.2 – Answers, *SortsCounting.java, Sorts.java, SortsTemplate.java*

Worksheet A18.1*, Answer Sheet*

**REFERENCES:** **Interactive Data Structure Visualizations**  
<http://www.student.seas.gwu.edu/~idsv/idsv.html>

Interactive Data Structure Visualizations on Sorting and Merging.

**INSTRUCTOR**

**NOTES:** The curriculum's presentation of merging two sorted arrays should be fairly straightforward. To save time, a four-case analysis of the merge algorithm is provided in the lesson. Students will usually have trouble implementing the merge algorithm. Allow for extra programming time and monitor their progress carefully.

A non-recursive mergeSort algorithm is also presented in the curriculum. Students will be asked to produce a recursive mergeSort. Developing a difficult recursive algorithm may pose a challenge for some students. Try the following strategy. Give only a general description of the answer, and then allow students to work on it for at least one day.

For those who cannot get started on the algorithm, develop the following pseudocode version of the answer at a later time. It might be helpful for these students to be in a small group setting.

**void** *mergeSort* (*ArrayList <Integer> list*, **int** *first*, **int** *last*) {

// Recursively divides a list in half, over and over. When the

// sublist has one or two values, stop subdividing.

**if** (sublist has only one value)

do nothing

**else** **if**

sublist has two values, sort it if necessary

**else** // recursion, divide list into two halves

Find midpoint of current sublist

Call mergeSort and process left sublist

Call mergeSort and process right sublist

merge left and right sublists

}

Rather than using only one base case, which is a list of one, establish two base cases. This strategy dramatically improves the performance of the algorithm. If a list of two numbers is identified and dealt with as a base case, the algorithm avoids two more recursive calls of mergeSort.

Role-playing these algorithms is also helpful as it was with the quadratic algorithms in Lesson 17, *Quadratic Sorting Algorithms*. The students usually want to throw everything into the ArrayList and then call a sort. This works but it is not a merge. The merge is the cornerstone of the recursive mergeSort, which is an improvement over any quadratic sort. A good discussion of the comparisons in steps is natural at this point. A level students need to understand the concept of Big O.

On the AP exam, students must be cautioned that they will not receive credit for their code if they do not answer a merge question using merge code. For instance, if they are asked to write a merge and they write code that does not specifically use a merge – but solves the problem - their answer won't be counted.

**WORKSHEET**

**NOTES:** Worksheet A18.1, *Non-recursive Merge and MergeSort* gives the students practice in tracking the movement of data in an ArrayList using the non-recursive mergeSort method. Note that this worksheet sorts two parts of *one* array as discussed in section B of the student lesson - using Selection Sort - as opposed to viewing these parts in two separate arrays (e.g. List A and List B as discussed in section A of the student lesson).