# CS201 - Assignment 4, Part 1
# Due: Friday, March 7, *at the beginning of class*

For part 1 of this assignment we'll be playing with some of the sorting algorithms we've discusses in class. In addition, you'll get some familiarity with the `merge` method of `MergeSort`, which you'll be implementing an on-disk version of for part 2 of this assignment.

## 1 Getting started

Create a new project in `Eclipse` called "Assign4.1" and then get all of the files from:

`http://www.cs.middlebury.edu/~dkauchak/classes/cs201/assignments/assign4/part1/`

into your project. You have two options of how to do this:

1. *(The slow way)* Create new classes for each of this in your project. Then copy and paste the code into the corresponding class.
2. *(The fast way)* Save each of the files into the "src" directory for your project. This will be under wherever you saved your workspaces at in a subdirectory called "Assign4.1". After you've saved all of the files in this directory, right-click on the project in Eclipse and select "Refresh". All of the classes should show up!

Spend 5 minutes looking at the different classes. In particular,

- Look at the `Sorter` interface
- Look at how the `Quicksort` and `MergeSort` classes implement the interface
- Look at how the `SortTimer` class is able to print out data for an arbitrary number of `Sorter` classes (this is the benefit of using an interface!)
- Notice that the `SortTimer` class does a check for correctness after sorting. If you make a mistake in implementing your `merge` method, you'll get an error here.

## 2 Finish `MergeSort`

I've given you all of the code for this part of the assignment except the `merge` method, which you should now implement. Give it a good effort, but if you get stuck, I've provided a solution below.

It will benefit you to figure it out in the for this part of the assignment, though, since you will be implementing something similar for the second part of the assignment.

Once this is done, you should be able to run the `SortTimer` class.

## 3 Play with the timing

Run the `SortTimer` class. The program runs Mergesort and Quicksort on increasing data sizes.

In a files called "assign4.1.txt' answer the following questions:

1. What are the running times of Quicksort and Mergesort? I know we haven't really talked about this yet, but look them up on the web. They should be something like "O(...)".
2. Which of these algorithms is faster?

As an additional experiment, change the `printTimes` method so that the algorithms try and sort data that's already sorted. (Change the call to `getRandom` to be `getSortedData` instead. You also need to change the `DATA_SIZE_START` constant at the top to something smaller, like 500. *Note:* even with making this smaller you will still get an exception eventually from the quicksort algorithm.

Add your answer to question below to your .txt file.

3. Which algorithm is faster now? Is that surprising? What does that mean?

## 4 What to submit

Submit your "Assign4.1.txt" with the answers to the three questions.

**An implementation of `merge`.** Here is one implementation of the `merge` algorithm. It uses an extra array to store the final sorted array and then copies the values back into the main array. Notice that this variant of the `merge` algorithm does **not** return a new array. The result result is stored back in the data array. This version is compatible with the 2nd version of the mergesort that we looked at in class.

```
public void merge(Comparable[] data, int low, int mid, int high){
    Comparable[] temp = new Comparable[high-low]; // temporary array

    int tempIndex = 0;
    int lowIndex = low;
    int midIndex = mid;

    while( lowIndex < mid && midIndex < high ){
        if( data[lowIndex].compareTo(data[midIndex]) < 1 ){
            temp[tempIndex] = data[lowIndex];
            lowIndex++;
        }else{
            temp[tempIndex] = data[midIndex];
            midIndex++;
        }

        tempIndex++;
    }

    // copy over the remaining data on the low to mid side if there
    // is some remaining.
    while( lowIndex < mid ){
        temp[tempIndex] = data[lowIndex];
        tempIndex++;
        lowIndex++;
    }

    // copy over the remaining data on the mid to high side if there
    // is some remaining.  Only one of these two while loops should
    // actually execute
    while( midIndex < high ){
        temp[tempIndex] = data[midIndex];
        tempIndex++;
        midIndex++;
    }

    // copy the data back from temp to list
    for( int i = 0; i < temp.length; i++ ){
        data[i+low] = temp[i];
    }
}
```