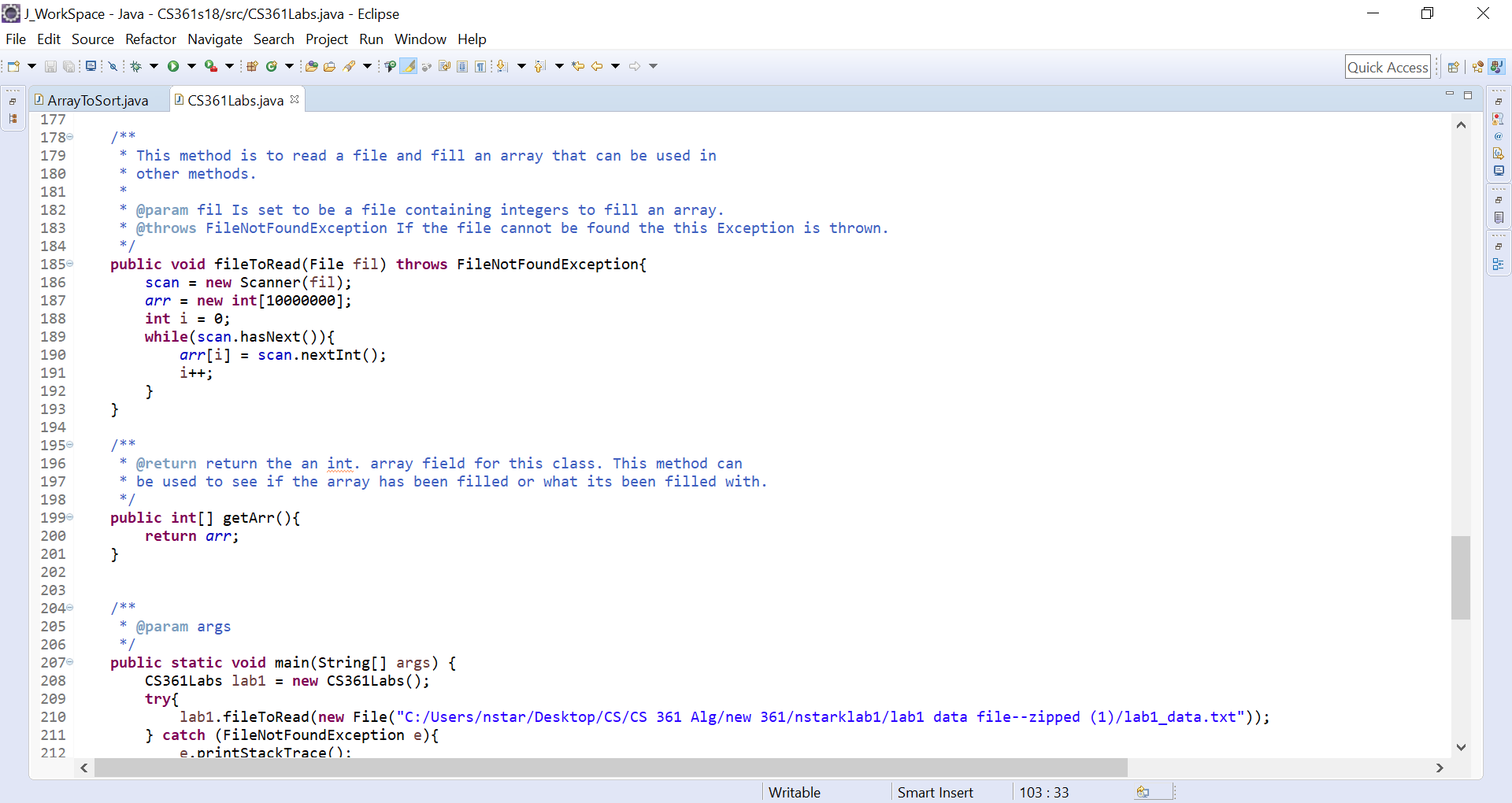
Nathan Stark

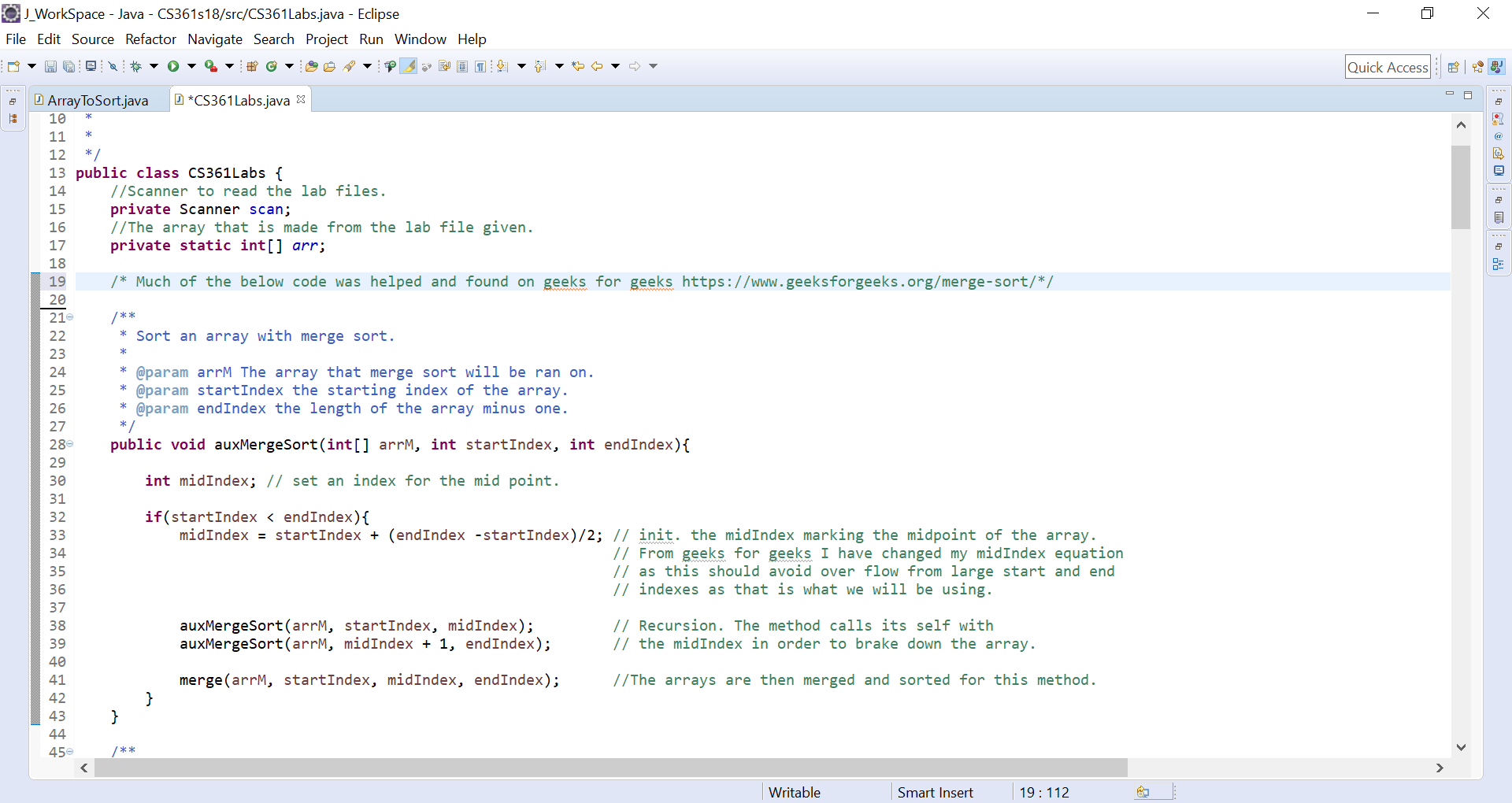
CS 361

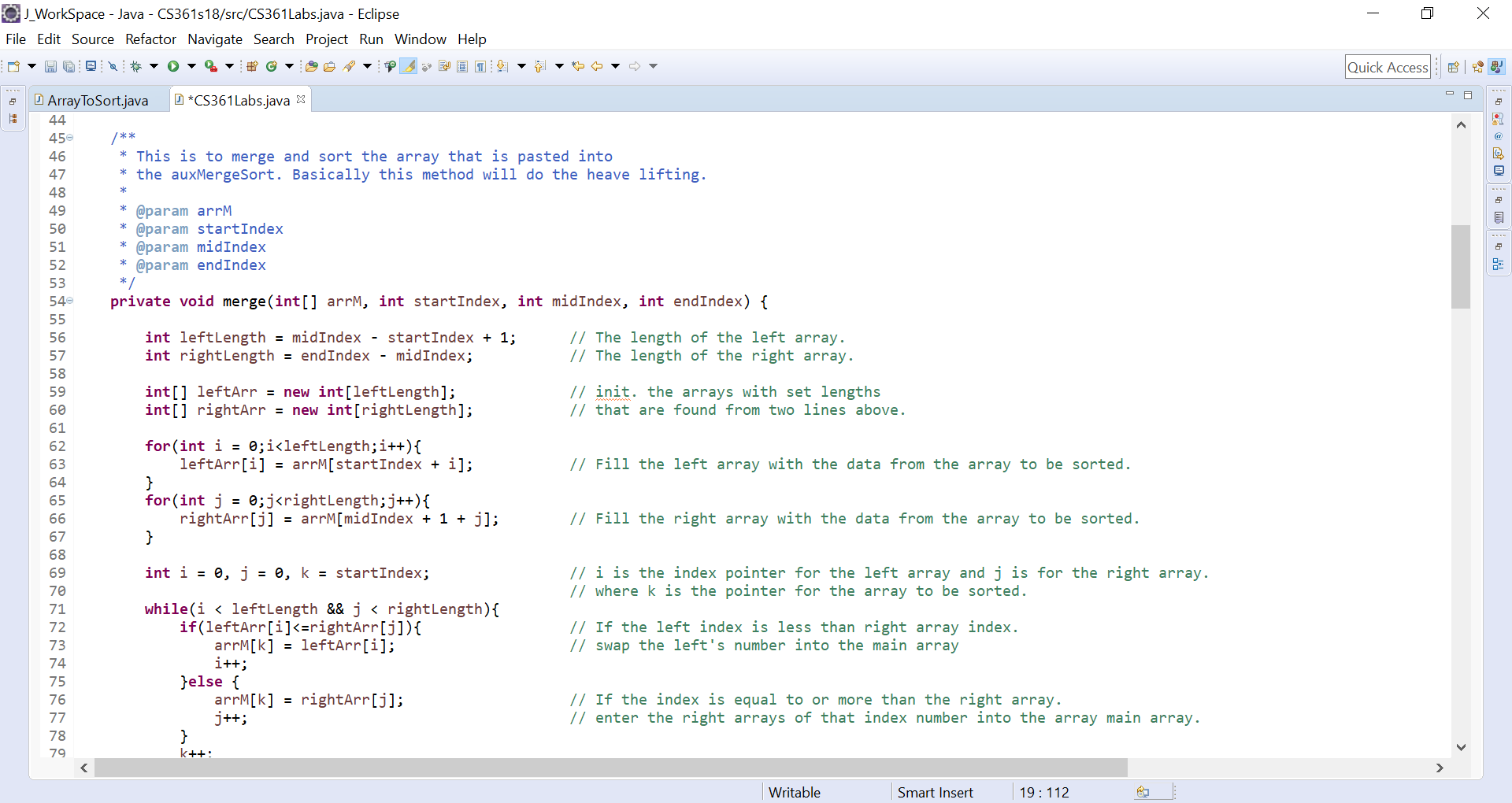
Lab 1

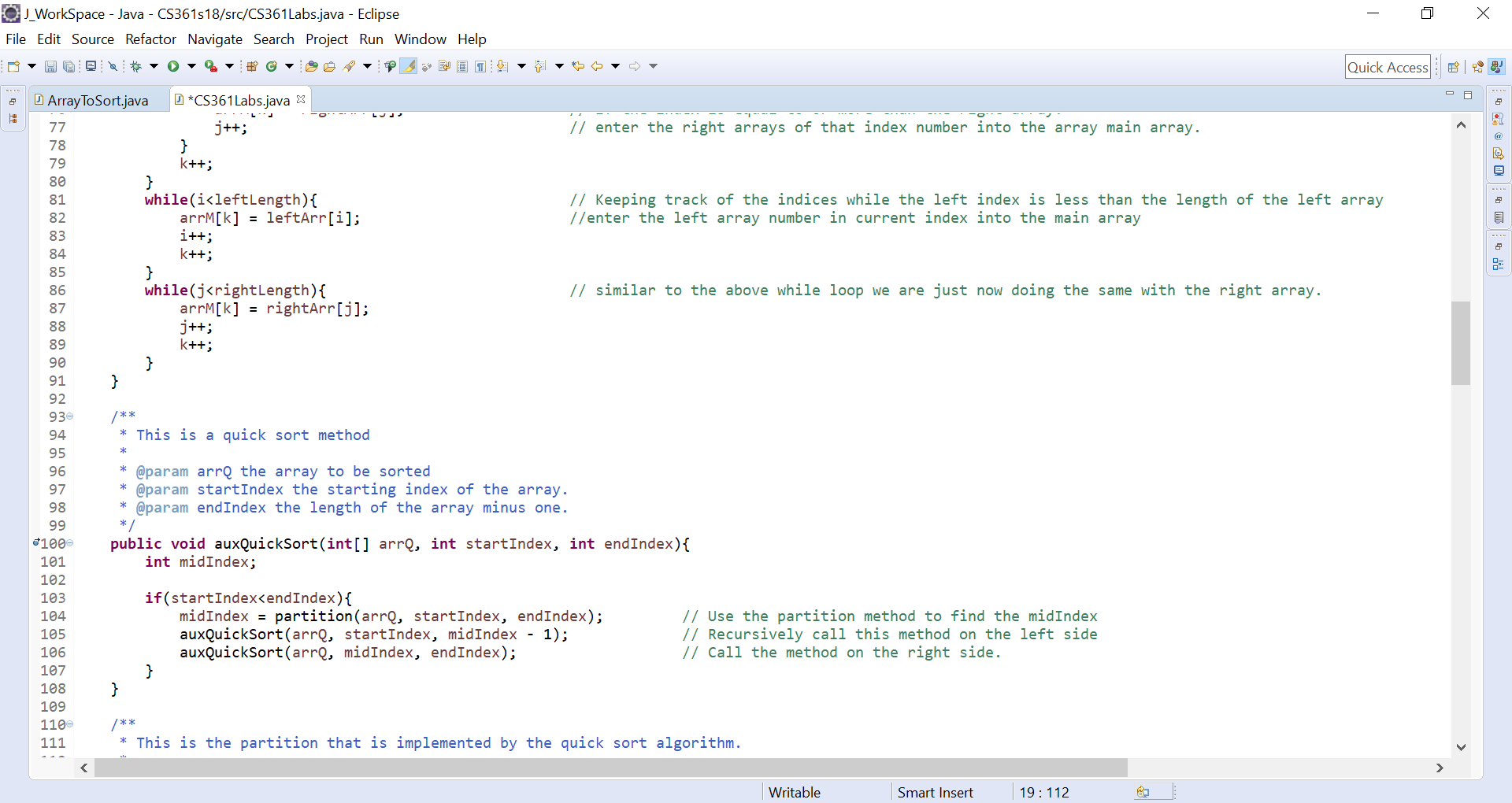
1. **Read the test data from the data file provided into an array of integers. The file contains 10,000,000 integers, one line per integer. If you read an empty line, then skip it. The sum of these integers is 49,999,995,000,000. Make sure you are using this information to verify the accuracy of your input routine.**



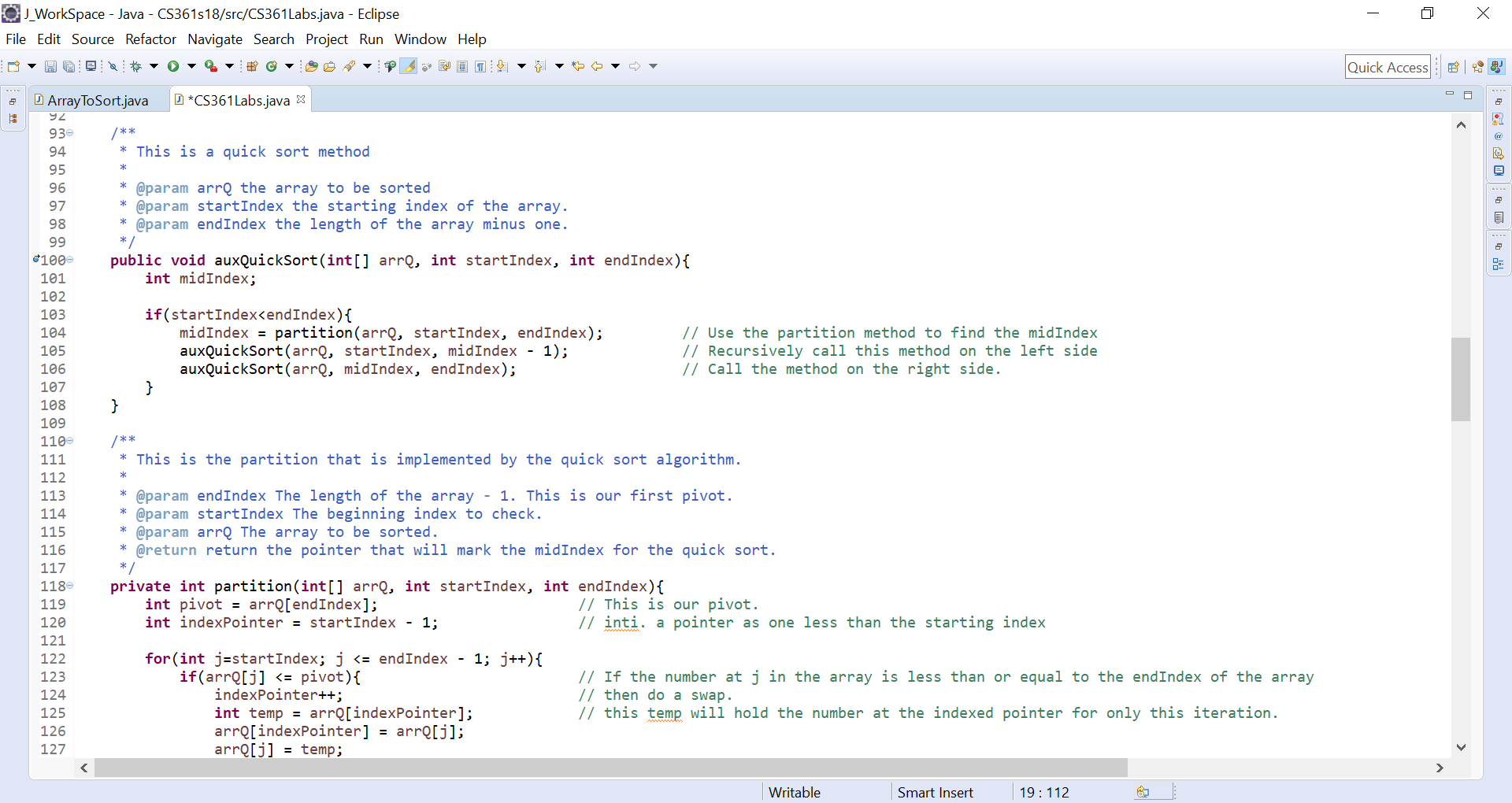
1. **Write a recursive method auxMergeSort that takes three parameters: the array, the startIndex, and the endIndex and sorts the elements between the startIndex and endIndex using Merge Sort. Consider coding the Merge part in a separate method that does not need to be recursive.**

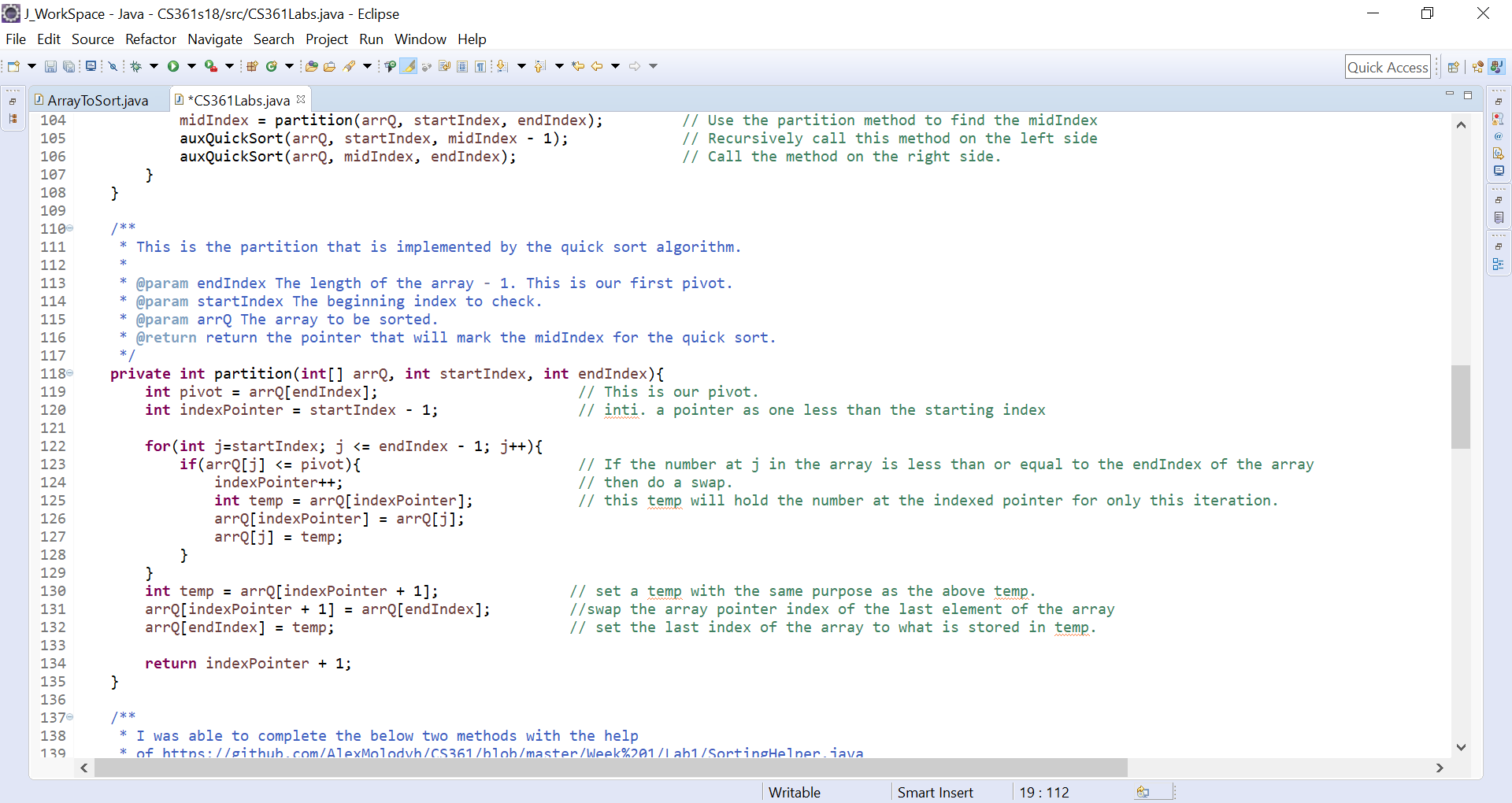




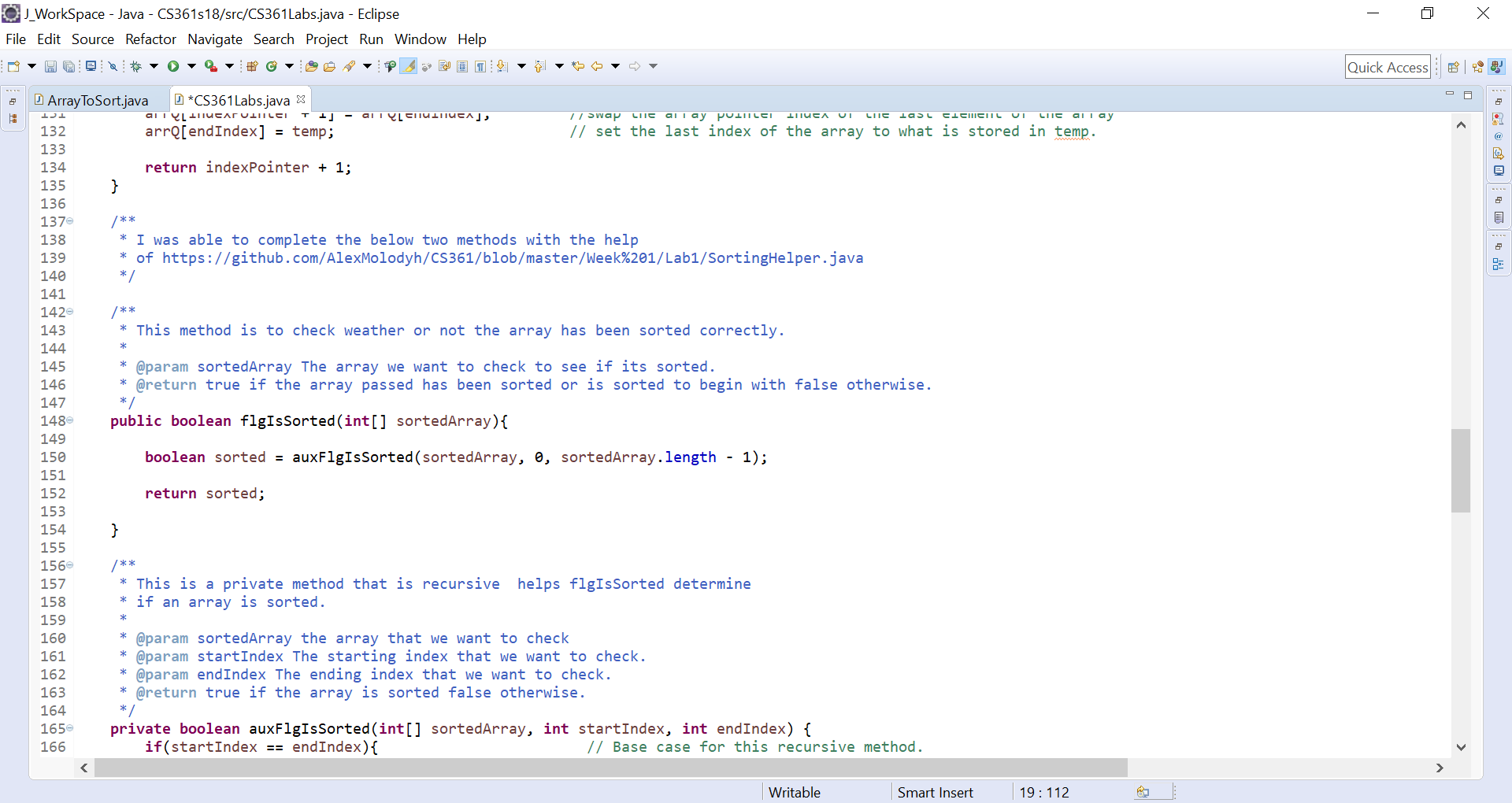


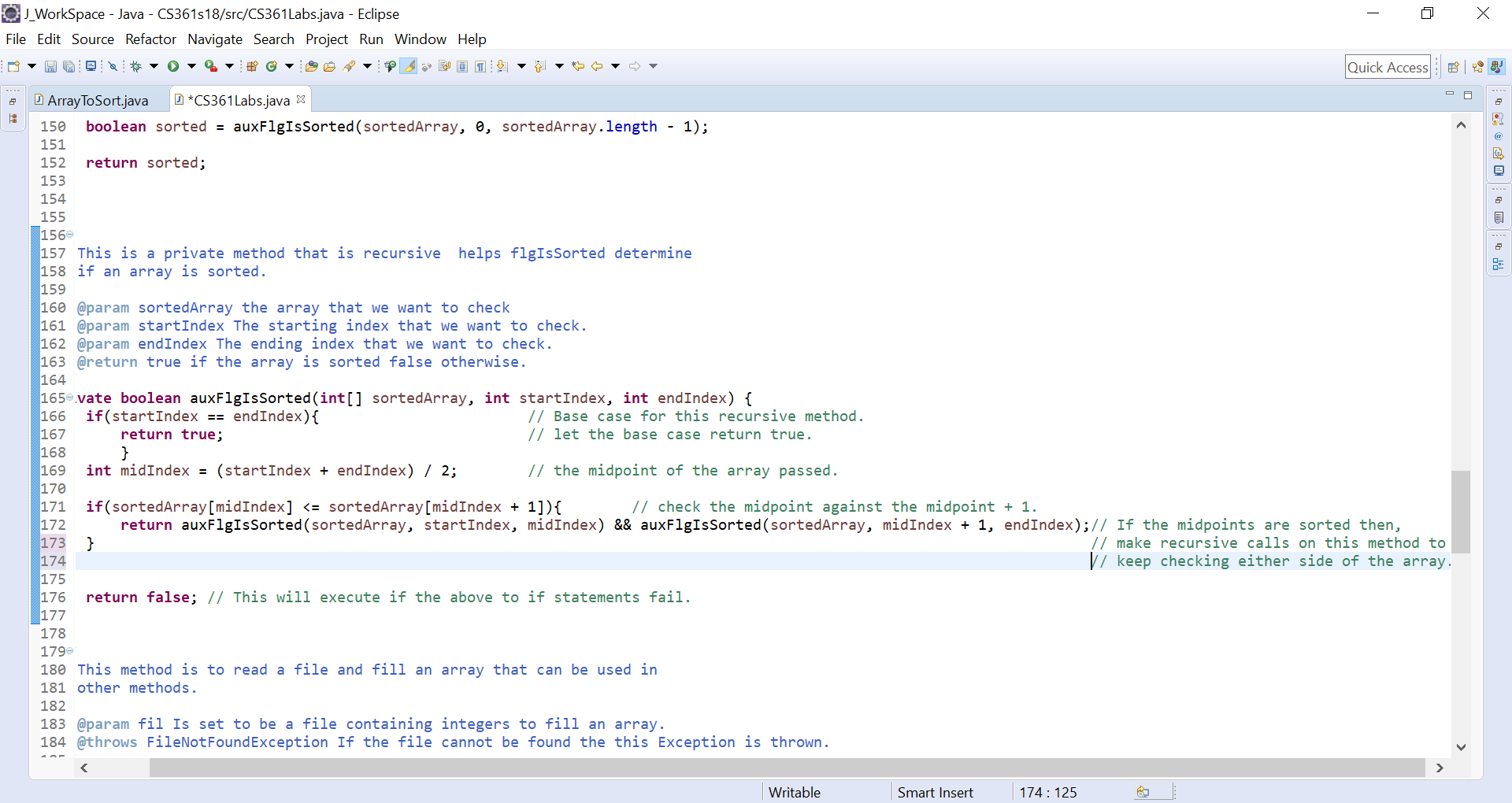
1. **Write a recursive method auxQuickSort that takes three parameters: the array, the startIndex, and the endIndex and sorts the elements between the startIndex and endIndex using Quick Sort with the pivot to be the average of the values at startIndex, endIndex, and the middle element between startIndex and endIndex using m = (startIndex + endIndex)/2. Consider coding the splitting part in a separate method that does not need to be recursive.**



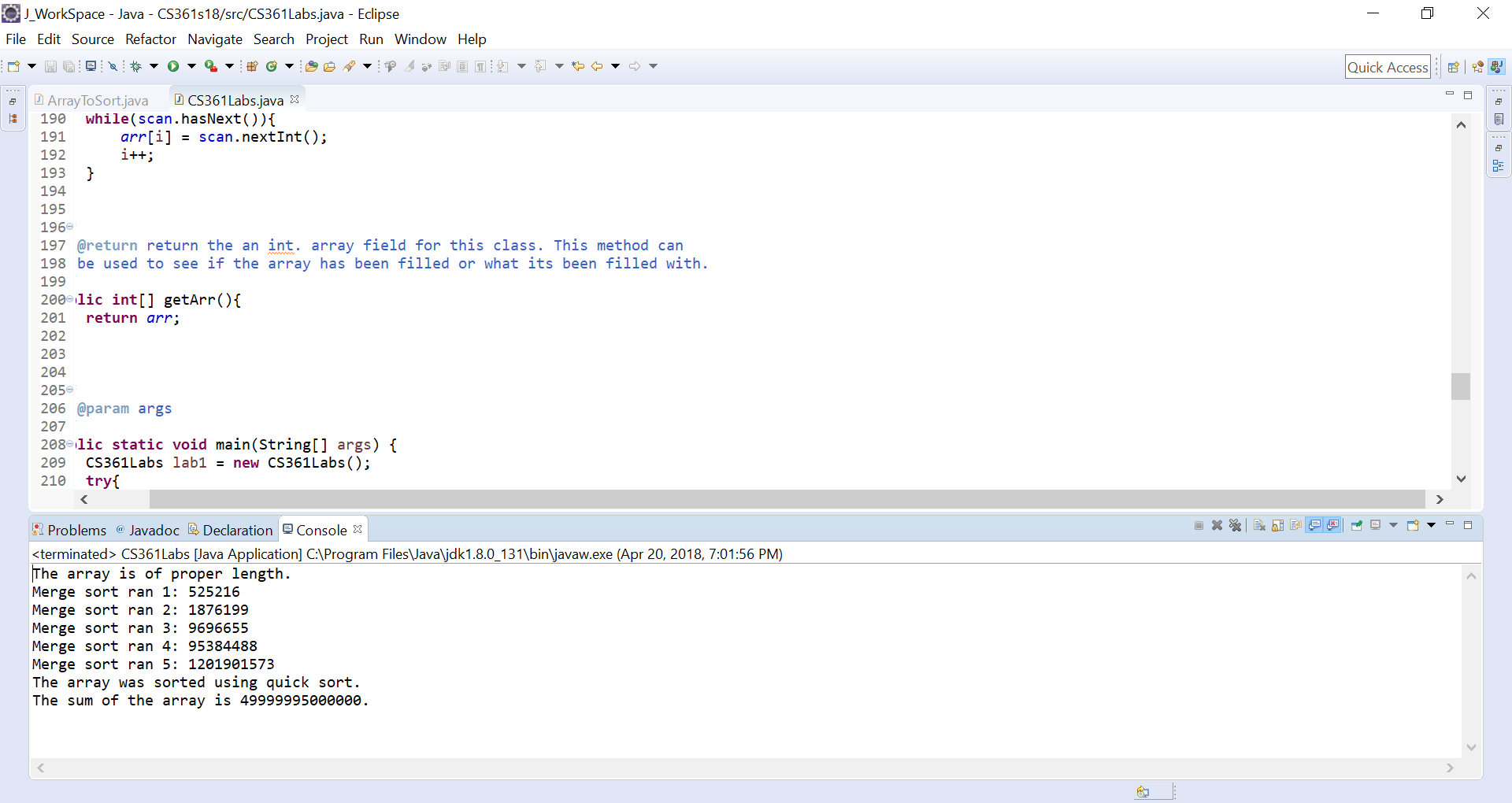


1. **Write a *recursive* method flgIsSorted to check if a given array (provided as a parameter) is sorted in increasing order. The method returns true if and only if the array is sorted in increasing order. Hint, when the array has only one element, it is sorted. If the first half is sorted, the second half is sorted, and the first element of the second half is larger than the last element in the first half, the array is sorted. Your initial method can only take one parameter – the array. That method can call another auxiliary method that takes other parameters.**





1. **Look into System.nanoTime() (or an equivalent); to time the number of nanoseconds needed to perform the above four methods. Now change your code so that, rather than perform all these steps on the 10 million integers, starts with 1,000 and increases at 10x until it reads more than 10 million numbers. Make sure to check whether the array is sorted using your flgIsSorted method. Run your code 3 times, record the execution time in milliseconds for each run on each size, enter the milliseconds reading into an Excel spreadsheet, calculate the average execution time in milliseconds for each run on each size and display your results in both a table and as a line chart. I am expecting to see a chart with two lines. Clearly indicate which line is which algorithm. Also, on your output show the result of using your flgIsSorted to check whether the array is sorted. Show the screen dump indicating your array is sorted and the time it takes for each run.**



Above is a screen dump quick sort running.

The first time that I ran for merge sort through the loop that started at 0 to 999 to 0 to 9999 and so on until 9999999.

The array is of proper length.

Merge sort ran 1: 859752

Merge sort ran 2: 1762170

Merge sort ran 3: 18047990

Merge sort ran 4: 162496852

Merge sort ran 5: 1822345918

The array was sorted using merge sort.

The sum of the array is 49999995000000.

The second time that I ran for merge sort through the loop.

The array is of proper length.

Merge sort ran 1: 838607

Merge sort ran 2: 1833533

Merge sort ran 3: 17933583

Merge sort ran 4: 160501715

Merge sort ran 5: 1906368241

The array was sorted using merge sort.

The sum of the array is 49999995000000.

The third time that I ran for merge sort through the loop.

The array is of proper length.

Merge sort ran 1: 898642

Merge sort ran 2: 2468624

Merge sort ran 3: 20850773

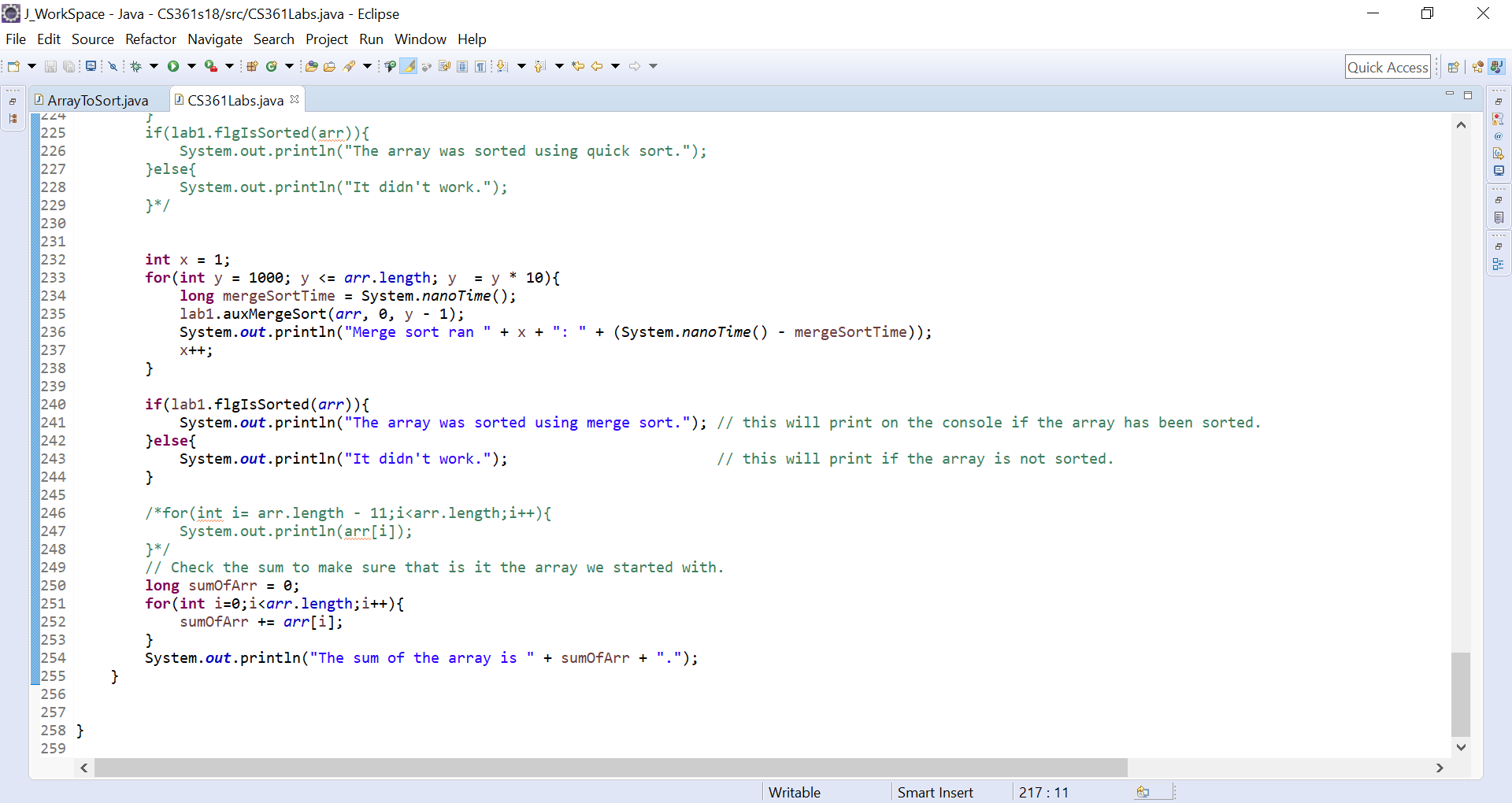
Merge sort ran 4: 163005454

Merge sort ran 5: 1907656170

The array was sorted using merge sort.

The sum of the array is 49999995000000.

Below is a screen shot of the code that I use to check if the array is sorted and to run the merge and quick sort. We can see that if flgIssorted returns true the the proper message will be printed.



The first time that I ran for quick sort through the loop that started at 0 to 999 to 0 to 9999 and so on until 9999999.

The array is of proper length.

Merge sort ran 1: 506713

Merge sort ran 2: 1701379

Merge sort ran 3: 12238530

Merge sort ran 4: 96868004

Merge sort ran 5: 1107885815

The array was sorted using quick sort.

The sum of the array is 49999995000000.

The second time that I ran for quick sort through the loop.

The array is of proper length.

Merge sort ran 1: 471599

Merge sort ran 2: 1817297

Merge sort ran 3: 9829564

Merge sort ran 4: 96188737

Merge sort ran 5: 1118819048

The array was sorted using quick sort.

The sum of the array is 49999995000000.

The third time that I ran for quick sort through the loop.

The array is of proper length.

Merge sort ran 1: 714005

Merge sort ran 2: 1275846

Merge sort ran 3: 9368538

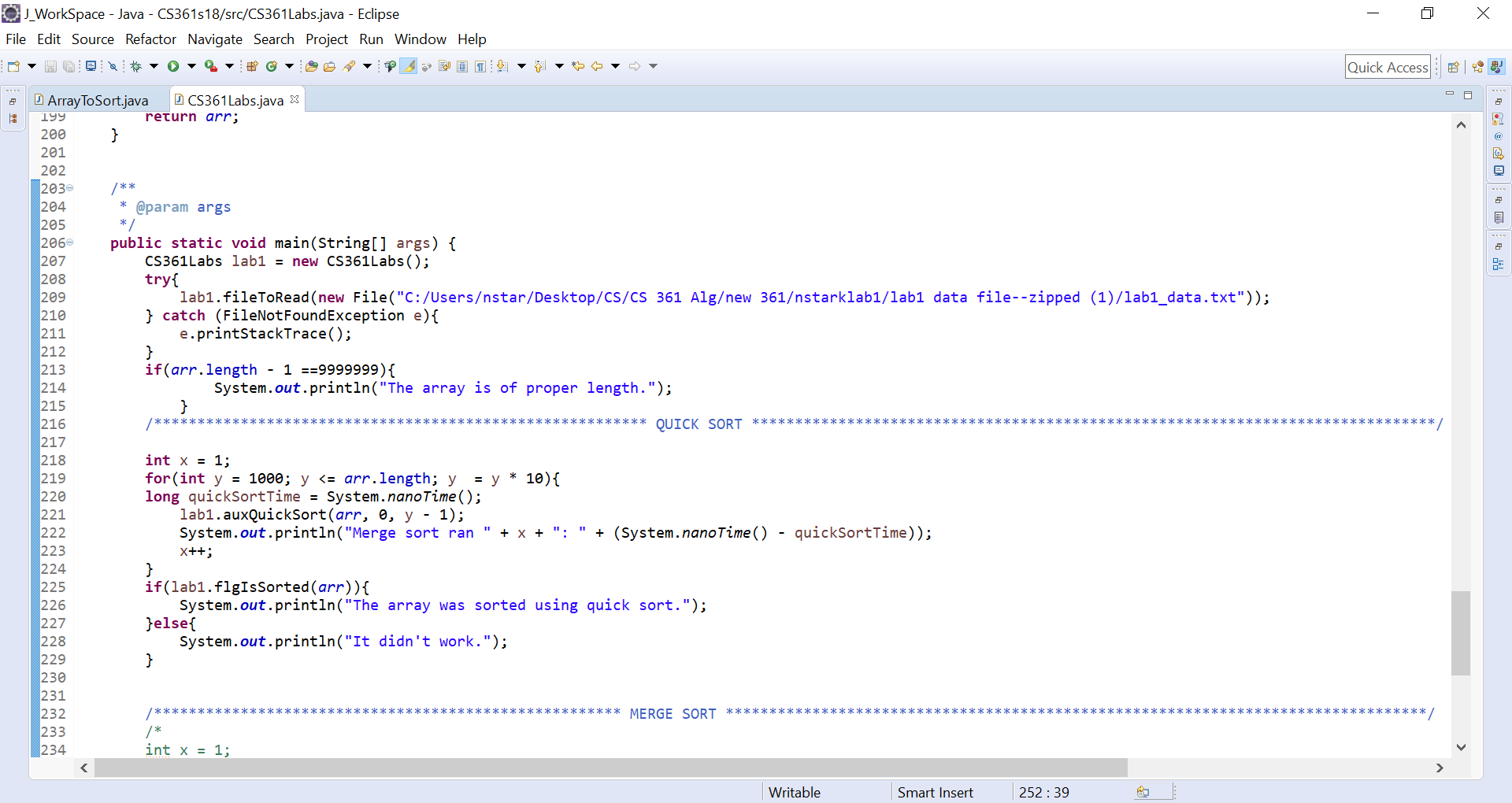
Merge sort ran 4: 97028099

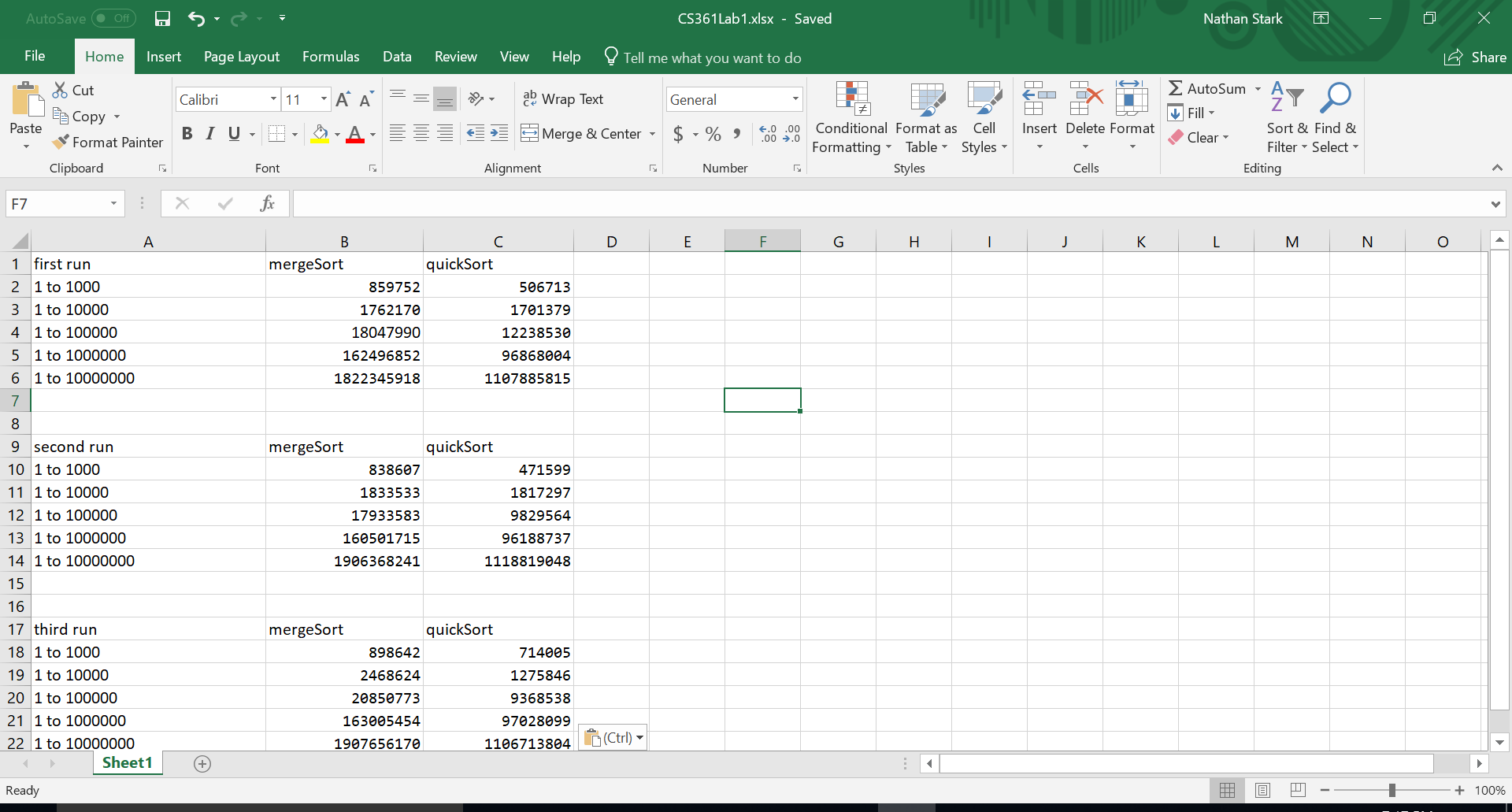
Merge sort ran 5: 1106713804

The array was sorted using quick sort.

The sum of the array is 49999995000000.

Below is a screen shot of the code that I use to check if the array is sorted and to run the quick sort.





Because it’s hard to read the first three results from merge and quick sort being ran against all five I added a second chart so that we can more clearly see what’s happening.

Form this first run we can clearly see that quicksort is faster than merge sort for all iterations.

As we can see from the charts above quick sort is more efficient than merge sort in these instances.