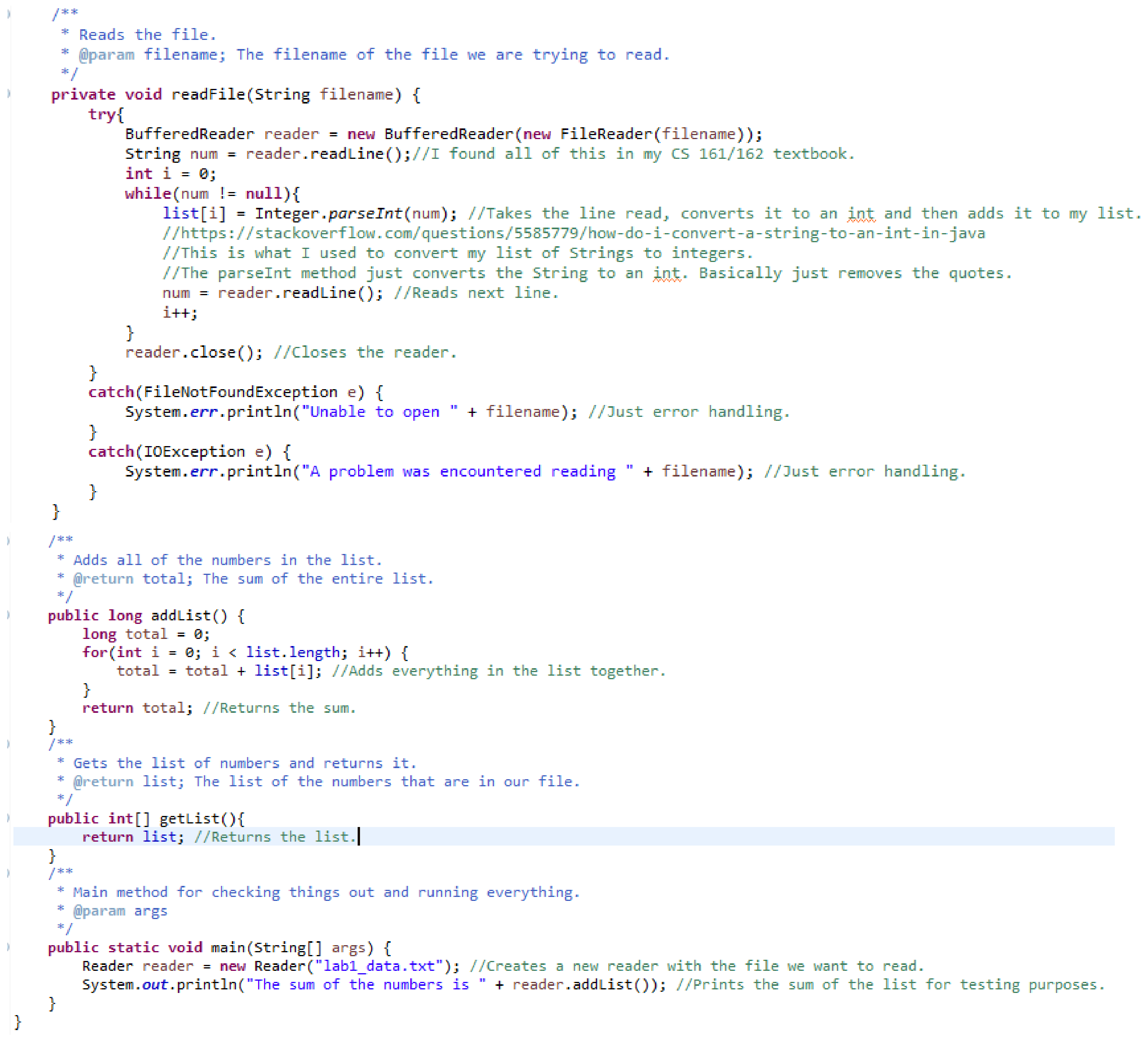
Tiffany Jansen
CS 361
Lab 1

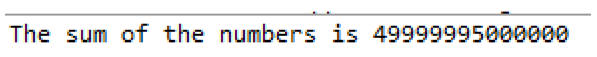
Lab Report

1. Code:



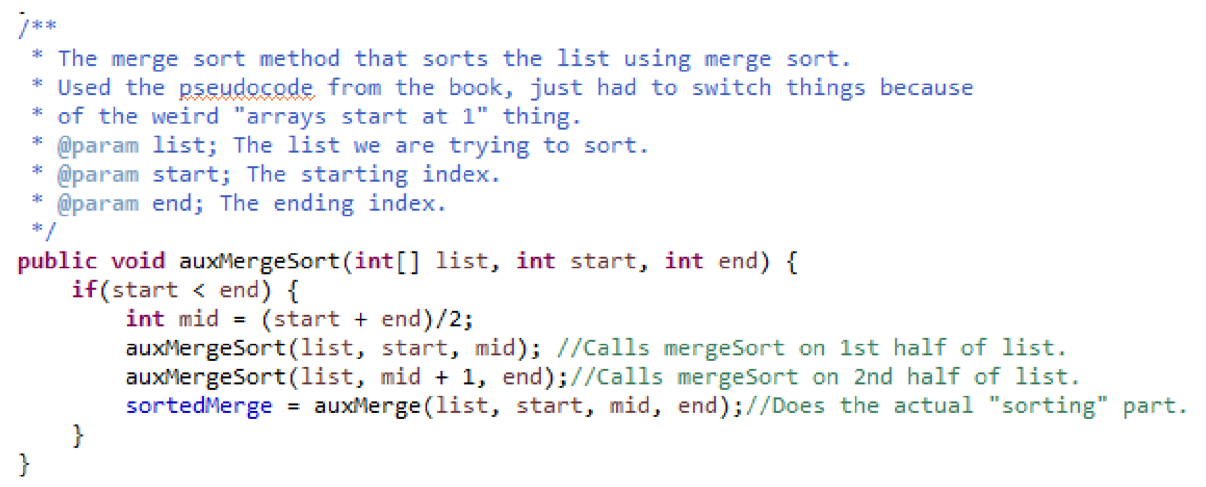
```java
/**
 * Reads the file.
 * @param filename; The filename of the file we are trying to read.
 */
private void readFile(String filename) {
    try{
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        String num = reader.readLine();//I found all of this in my CS 161/162 textbook.
        int i = 0;
        while(num != null){
            list[i] = Integer.parseInt(num); //Takes the line read, converts it to an int and then adds it to my list.
            //https://stackoverflow.com/questions/5585779/how-do-i-convert-a-string-to-an-int-in-java
            //This is what I used to convert my list of Strings to integers.
            //The parseInt method just converts the String to an int. Basically just removes the quotes.
            num = reader.readLine(); //Reads next line.
            i++;
        }
        reader.close(); //Closes the reader.
    }
    catch(FileNotFoundException e) {
        System.err.println("Unable to open " + filename); //Just error handling.
    }
    catch(IOException e) {
        System.err.println("A problem was encountered reading " + filename); //Just error handling.
    }
}

/**
 * Adds all of the numbers in the list.
 * @return total; The sum of the entire list.
 */
public long addList() {
    long total = 0;
    for(int i = 0; i < list.length; i++) {
        total = total + list[i]; //Adds everything in the list together.
    }
    return total; //Returns the sum.
}
/**
 * Gets the list of numbers and returns it.
 * @return list; The list of the numbers that are in our file.
 */
public int[] getList(){
    return list; //Returns the list.
}
/**
 * Main method for checking things out and running everything.
 * @param args
 */
public static void main(String[] args) {
    Reader reader = new Reader("lab1_data.txt"); //Creates a new reader with the file we want to read.
    System.out.println("The sum of the numbers is " + reader.addList()); //Prints the sum of the list for testing purposes.
}
}
```

Output:

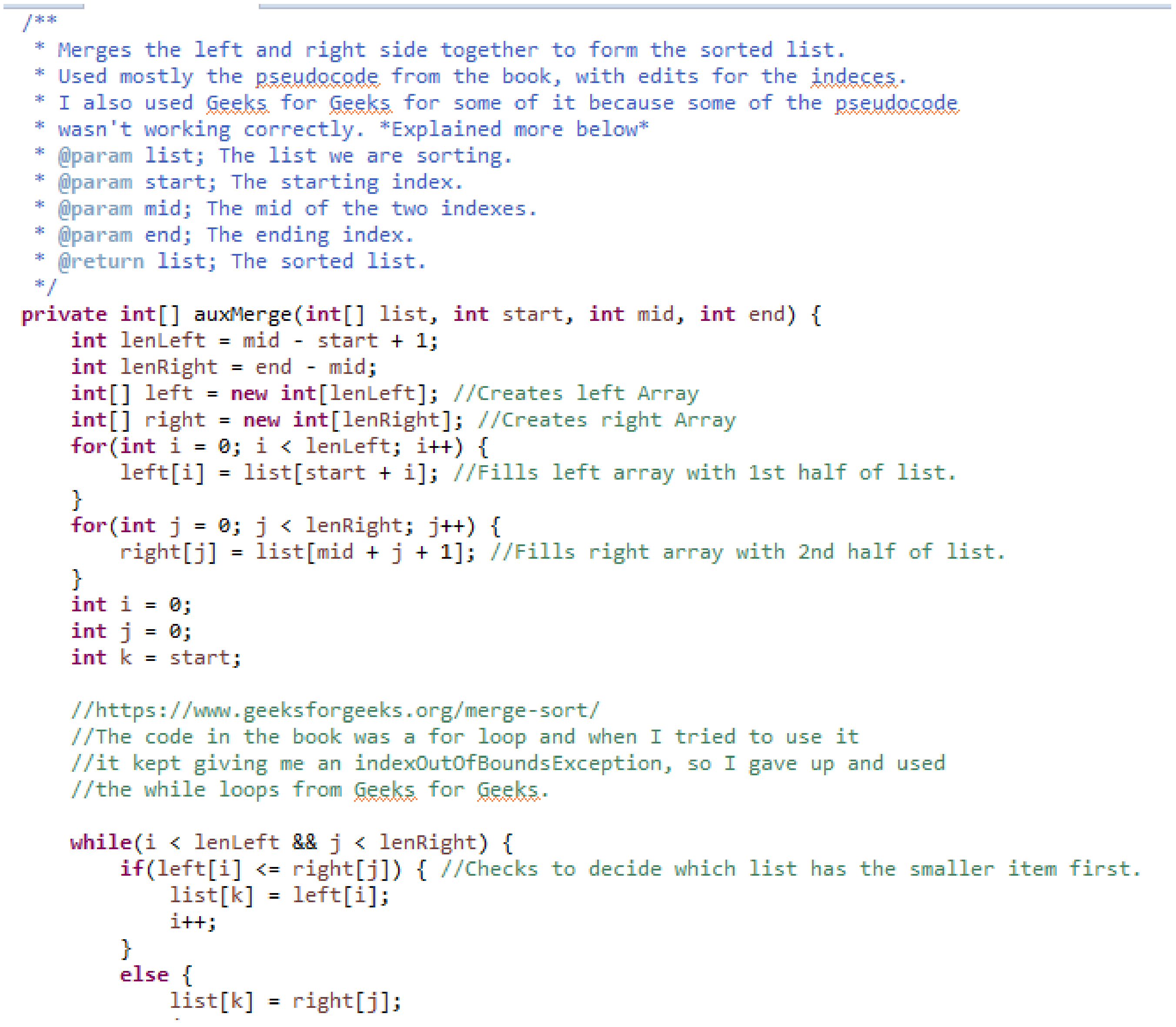


```
The sum of the numbers is 49999995000000
```

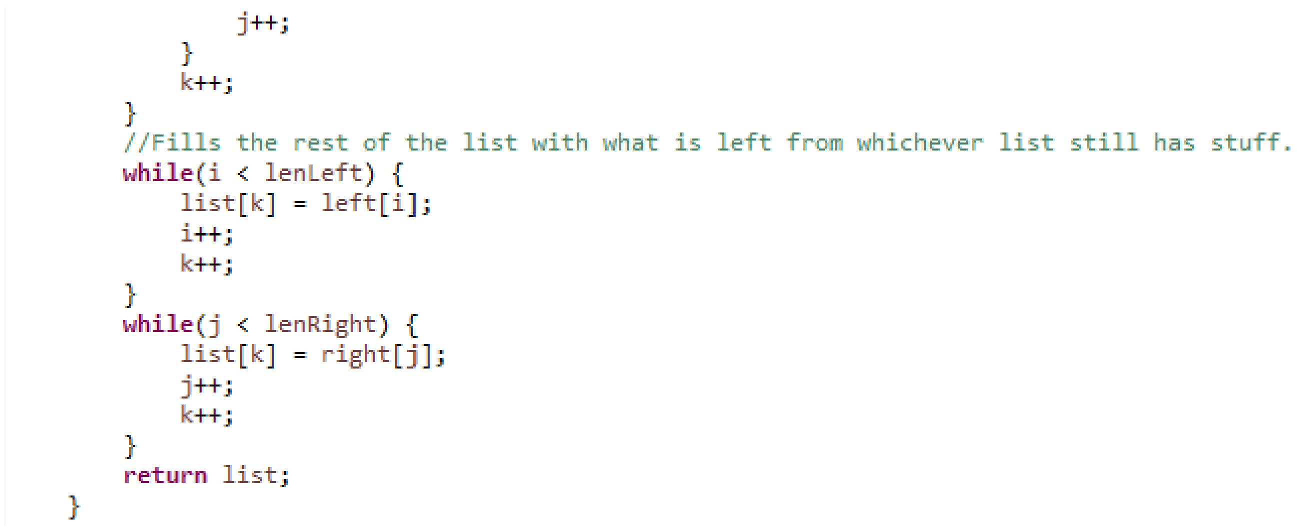
2. **Code:**



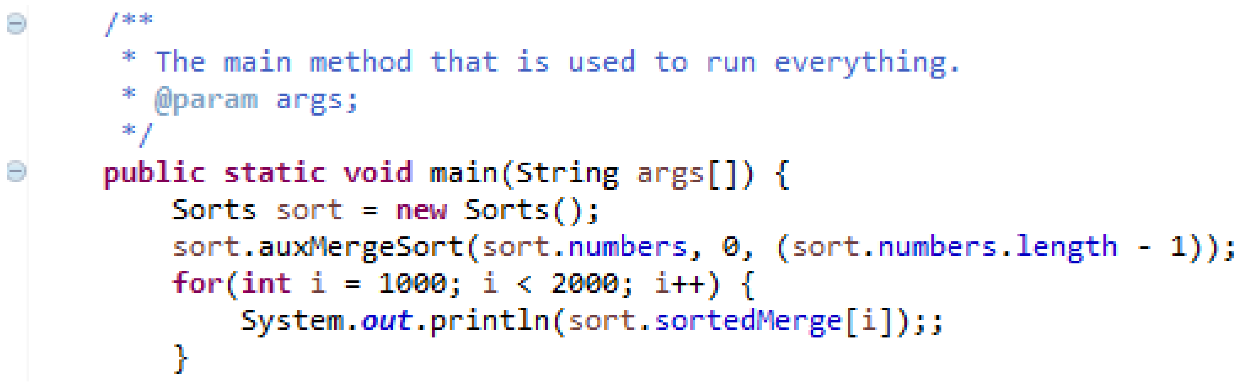
```java
/**
 * The merge sort method that sorts the list using merge sort.
 * Used the pseudocode from the book, just had to switch things because
 * of the weird "arrays start at 1" thing.
 * @param list; The list we are trying to sort.
 * @param start; The starting index.
 * @param end; The ending index.
 */
public void auxMergeSort(int[] list, int start, int end) {
    if(start < end) {
        int mid = (start + end)/2;
        auxMergeSort(list, start, mid); //Calls mergeSort on 1st half of list.
        auxMergeSort(list, mid + 1, end);//Calls mergeSort on 2nd half of list.
        sortedMerge = auxMerge(list, start, mid, end);//Does the actual "sorting" part.
    }
}
```



```java
/**
 * Merges the left and right side together to form the sorted list.
 * Used mostly the pseudocode from the book, with edits for the indeces.
 * I also used Geeks for Geeks for some of it because some of the pseudocode
 * wasn't working correctly. *Explained more below*
 * @param list; The list we are sorting.
 * @param start; The starting index.
 * @param mid; The mid of the two indexes.
 * @param end; The ending index.
 * @return list; The sorted list.
 */
private int[] auxMerge(int[] list, int start, int mid, int end) {
    int lenLeft = mid - start + 1;
    int lenRight = end - mid;
    int[] left = new int[lenLeft]; //Creates left Array
    int[] right = new int[lenRight]; //Creates right Array
    for(int i = 0; i < lenLeft; i++) {
        left[i] = list[start + i]; //Fills left array with 1st half of list.
    }
    for(int j = 0; j < lenRight; j++) {
        right[j] = list[mid + j + 1]; //Fills right array with 2nd half of list.
    }
    int i = 0;
    int j = 0;
    int k = start;

    //https://www.geeksforgeeks.org/merge-sort/
    //The code in the book was a for loop and when I tried to use it
    //it kept giving me an indexOutOfBoundsException, so I gave up and used
    //the while loops from Geeks for Geeks.

    while(i < lenLeft && j < lenRight) {
        if(left[i] <= right[j]) { //Checks to decide which list has the smaller item first.
            list[k] = left[i];
            i++;
        }
        else {
            list[k] = right[j];
```



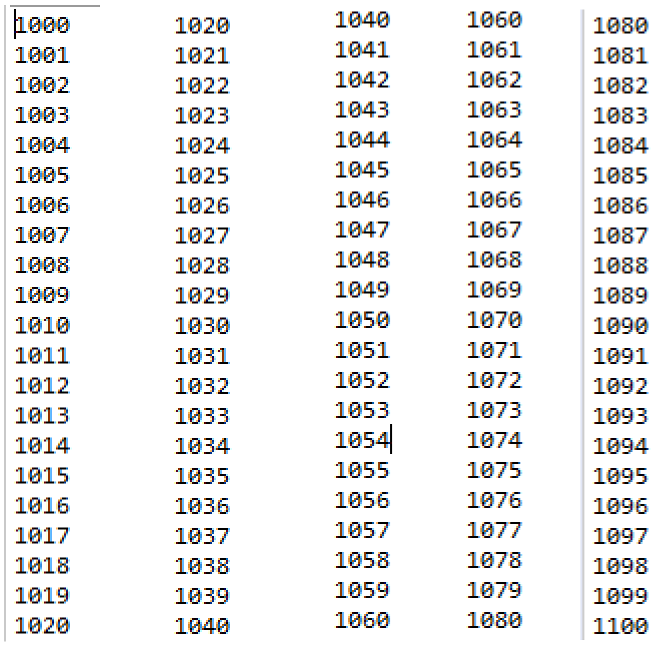
```java
                j++;
            }
            k++;
        }
        //Fills the rest of the list with what is left from whichever list still has stuff.
        while(i < lenLeft) {
            list[k] = left[i];
            i++;
            k++;
        }
        while(j < lenRight) {
            list[k] = right[j];
            j++;
            k++;
        }
        return list;
    }
```



```java
    /**
     * The main method that is used to run everything.
     * @param args;
     */
    public static void main(String args[]) {
        Sorts sort = new Sorts();
        sort.auxMergeSort(sort.numbers, 0, (sort.numbers.length - 1));
        for(int i = 1000; i < 2000; i++) {
            System.out.println(sort.sortedMerge[i]);;
        }
```

Output:

It did not actually repeat numbers, I just thought it would be easier to tell which order they go in.



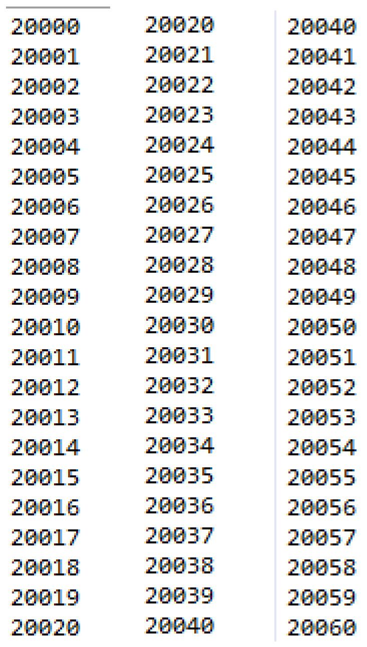
```
1000    1020    1040    1060    1080
1001    1021    1041    1061    1081
1002    1022    1042    1062    1082
1003    1023    1043    1063    1083
1004    1024    1044    1064    1084
1005    1025    1045    1065    1085
1006    1026    1046    1066    1086
1007    1027    1047    1067    1087
1008    1028    1048    1068    1088
1009    1029    1049    1069    1089
1010    1030    1050    1070    1090
1011    1031    1051    1071    1091
1012    1032    1052    1072    1092
1013    1033    1053    1073    1093
1014    1034    1054    1074    1094
1015    1035    1055    1075    1095
1016    1036    1056    1076    1096
1017    1037    1057    1077    1097
1018    1038    1058    1078    1098
1019    1039    1059    1079    1099
1020    1040    1060    1080    1100
```

This continued all the way to 1999, as to be expected.

3. Code:

```
 /**
  * The quick sort method that sorts the list using quick sort.
  * Used pseudocode from the book, again just editing the index problem.
  * @param list; The list we are trying to sort.
  * @param start; The starting index.
  * @param end; The ending index.
  */
 public void auxQuickSort(int[] list, int start, int end) {
     if(start < end) {
         int done = auxPartition(list, start, end);
         auxQuickSort(list, start, done - 1);
         auxQuickSort(list, done + 1, end);
     }
 }

/**
 * Partitions the list into "less than" and "greater than" sections.
 * Used as a helper method for quick sort.
 * Used the pseudocode from the book with some minor adjustments.
 * @param list; The list we are partitioning.
 * @param start; The starting index.
 * @param end; The ending index.
 * @return pivot; The number that the list was pivoted around.
 */
private int auxPartition(int[] list, int start, int end) {
    int mid =(start + end)/2;
    list = auxExchange(list, mid, end);
    int pivot = start;
    for(int j = start; j <= end - 1; j++) {
        if(list[j] <= list[end]) {
            list = auxExchange(list, pivot, j);
            pivot++;
        }
    }
    list = auxExchange(list, pivot, end);
    return pivot;
}   /**
     * This method was just used to exchange places in the list so I didn't
     * have to write it in the partition code 3 times.
     * @param list; The list we are sorting.
     * @param first; The index of the first number to switch.
     * @param second; The index of the second number to switch.
     * @return list; The list with the two numbers switched.
     */
    private int[] auxExchange(int[] list, int first, int second) {
        int newFirst = list[second];
        list[second] = list[first];
        list[first] = newFirst;
        return list;
    }
```

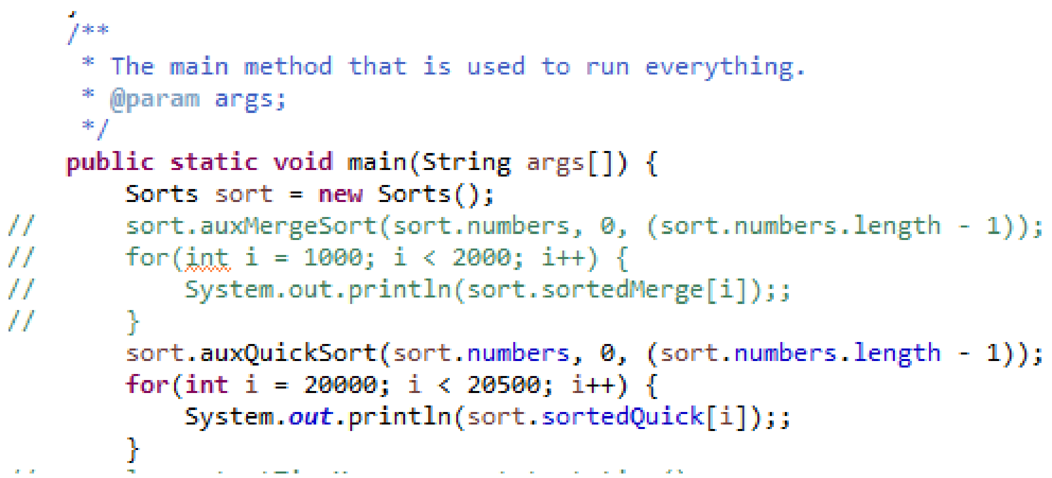
Output:



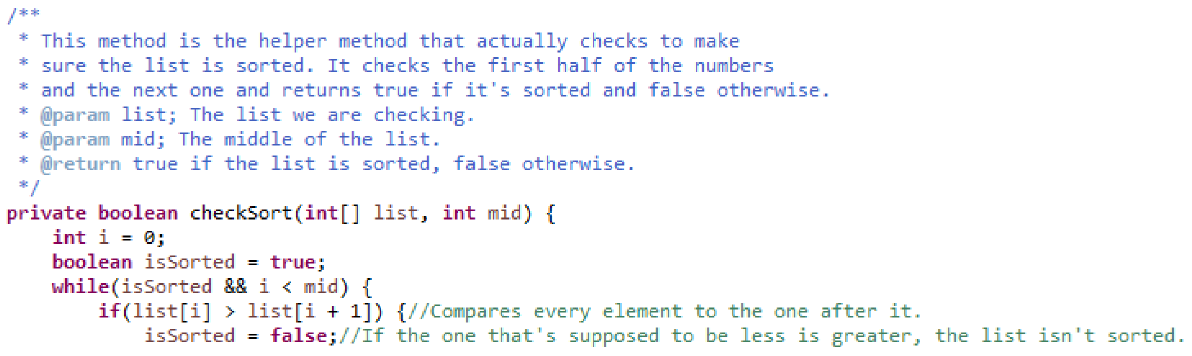
```
20000    20020    20040
20001    20021    20041
20002    20022    20042
20003    20023    20043
20004    20024    20044
20005    20025    20045
20006    20026    20046
20007    20027    20047
20008    20028    20048
20009    20029    20049
20010    20030    20050
20011    20031    20051
20012    20032    20052
20013    20033    20053
20014    20034    20054
20015    20035    20055
20016    20036    20056
20017    20037    20057
20018    20038    20058
20019    20039    20059
20020    20040    20060
```

**Same sort of thing with the repeating numbers. I’m only going to put 60 numbers here, but it continues all the way until I told it to stop.

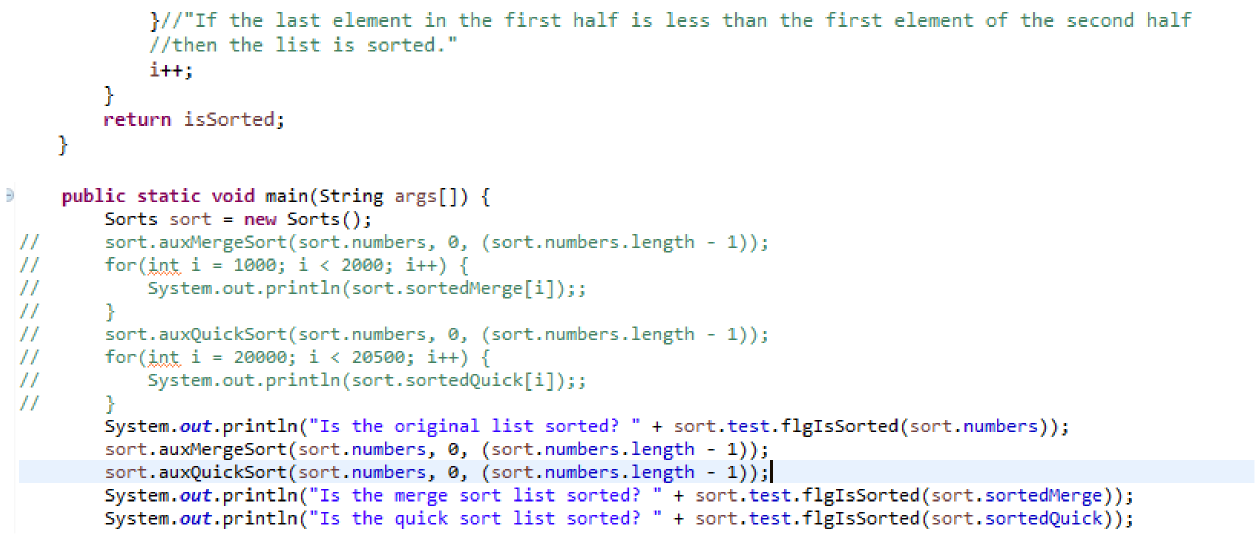
4. **Code:**



```
    /**
     * The main method that is used to run everything.
     * @param args;
     */
    public static void main(String args[]) {
        Sorts sort = new Sorts();
//      sort.auxMergeSort(sort.numbers, 0, (sort.numbers.length - 1));
//      for(int i = 1000; i < 2000; i++) {
//          System.out.println(sort.sortedMerge[i]);;
//      }
        sort.auxQuickSort(sort.numbers, 0, (sort.numbers.length - 1));
        for(int i = 20000; i < 20500; i++) {
            System.out.println(sort.sortedQuick[i]);;
        }
```

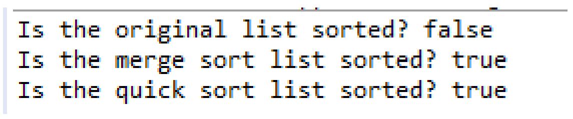


```
/**
 * This method is the helper method that actually checks to make
 * sure the list is sorted. It checks the first half of the numbers
 * and the next one and returns true if it's sorted and false otherwise.
 * @param list; The list we are checking.
 * @param mid; The middle of the list.
 * @return true if the list is sorted, false otherwise.
 */
private boolean checkSort(int[] list, int mid) {
    int i = 0;
    boolean isSorted = true;
    while(isSorted && i < mid) {
        if(list[i] > list[i + 1]) {//Compares every element to the one after it.
            isSorted = false;//If the one that's supposed to be less is greater, the list isn't sorted.
```



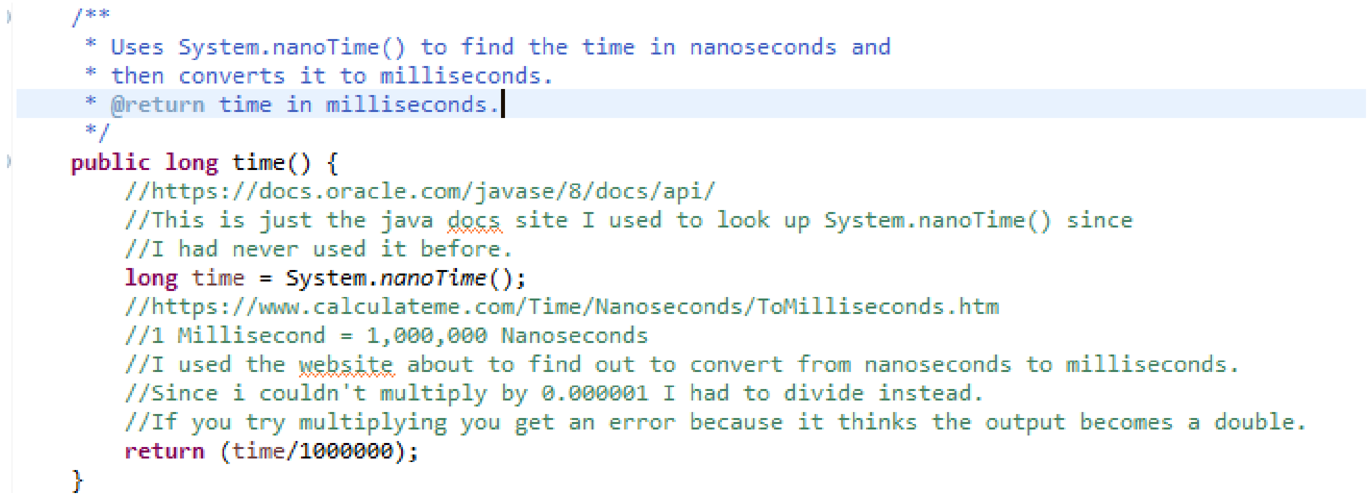
```
            }//"If the last element in the first half is less than the first element of the second half
            //then the list is sorted."
            i++;
        }
        return isSorted;
    }

    public static void main(String args[]) {
        Sorts sort = new Sorts();
//      sort.auxMergeSort(sort.numbers, 0, (sort.numbers.length - 1));
//      for(int i = 1000; i < 2000; i++) {
//          System.out.println(sort.sortedMerge[i]);;
//      }
//      sort.auxQuickSort(sort.numbers, 0, (sort.numbers.length - 1));
//      for(int i = 20000; i < 20500; i++) {
//          System.out.println(sort.sortedQuick[i]);;
//      }
        System.out.println("Is the original list sorted? " + sort.test.flgIsSorted(sort.numbers));
        sort.auxMergeSort(sort.numbers, 0, (sort.numbers.length - 1));
        sort.auxQuickSort(sort.numbers, 0, (sort.numbers.length - 1));
        System.out.println("Is the merge sort list sorted? " + sort.test.flgIsSorted(sort.sortedMerge));
        System.out.println("Is the quick sort list sorted? " + sort.test.flgIsSorted(sort.sortedQuick));
```

Output:

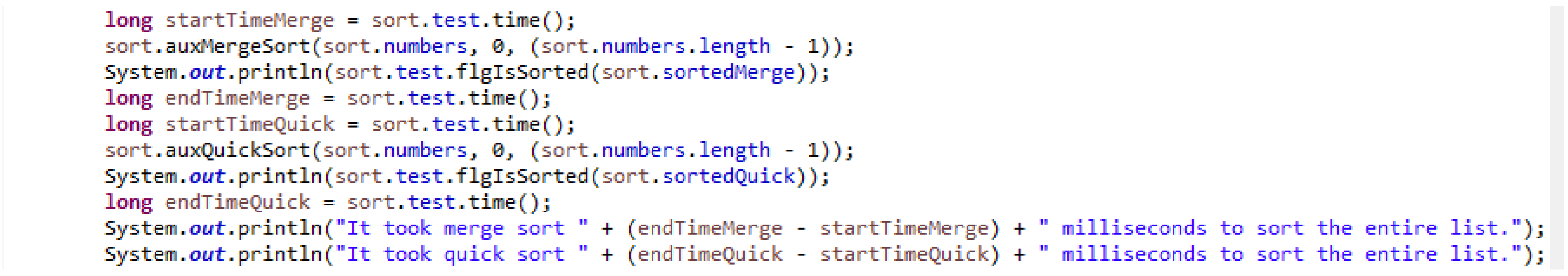


```
Is the original list sorted? false
Is the merge sort list sorted? true
Is the quick sort list sorted? true
```

5. Code:

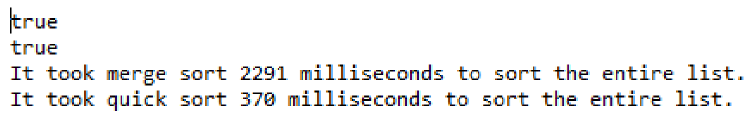


```
    /**
     * Uses System.nanoTime() to find the time in nanoseconds and
     * then converts it to milliseconds.
     * @return time in milliseconds.
     */
    public long time() {
        //https://docs.oracle.com/javase/8/docs/api/
        //This is just the java docs site I used to look up System.nanoTime() since
        //I had never used it before.
        long time = System.nanoTime();
        //https://www.calculateme.com/Time/Nanoseconds/ToMilliseconds.htm
        //1 Millisecond = 1,000,000 Nanoseconds
        //I used the website about to find out to convert from nanoseconds to milliseconds.
        //Since i couldn't multiply by 0.000001 I had to divide instead.
        //If you try multiplying you get an error because it thinks the output becomes a double.
        return (time/1000000);
    }
```



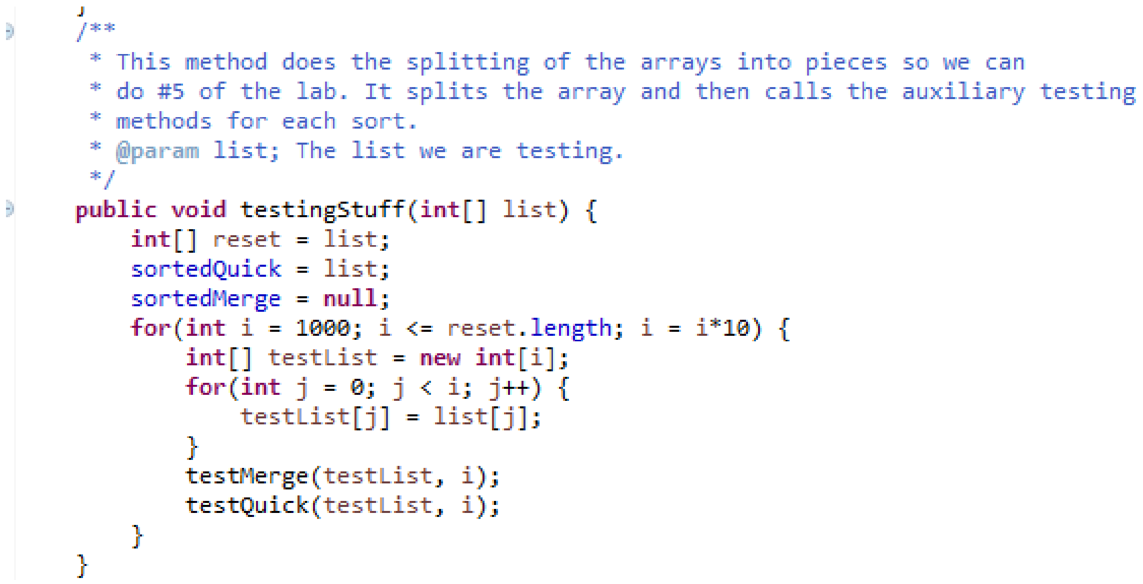
```
long startTimeMerge = sort.test.time();
sort.auxMergeSort(sort.numbers, 0, (sort.numbers.length - 1));
System.out.println(sort.test.flgIsSorted(sort.sortedMerge));
long endTimeMerge = sort.test.time();
long startTimeQuick = sort.test.time();
sort.auxQuickSort(sort.numbers, 0, (sort.numbers.length - 1));
System.out.println(sort.test.flgIsSorted(sort.sortedQuick));
long endTimeQuick = sort.test.time();
System.out.println("It took merge sort " + (endTimeMerge - startTimeMerge) + " milliseconds to sort the entire list.");
System.out.println("It took quick sort " + (endTimeQuick - startTimeQuick) + " milliseconds to sort the entire list.");
```

Output:

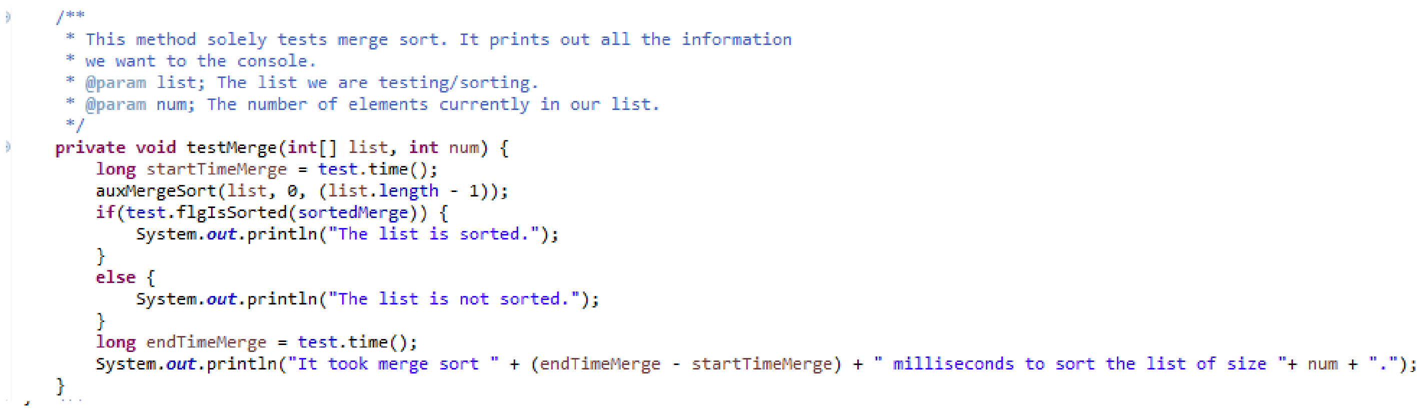


```
true
true
It took merge sort 2291 milliseconds to sort the entire list.
It took quick sort 370 milliseconds to sort the entire list.
```

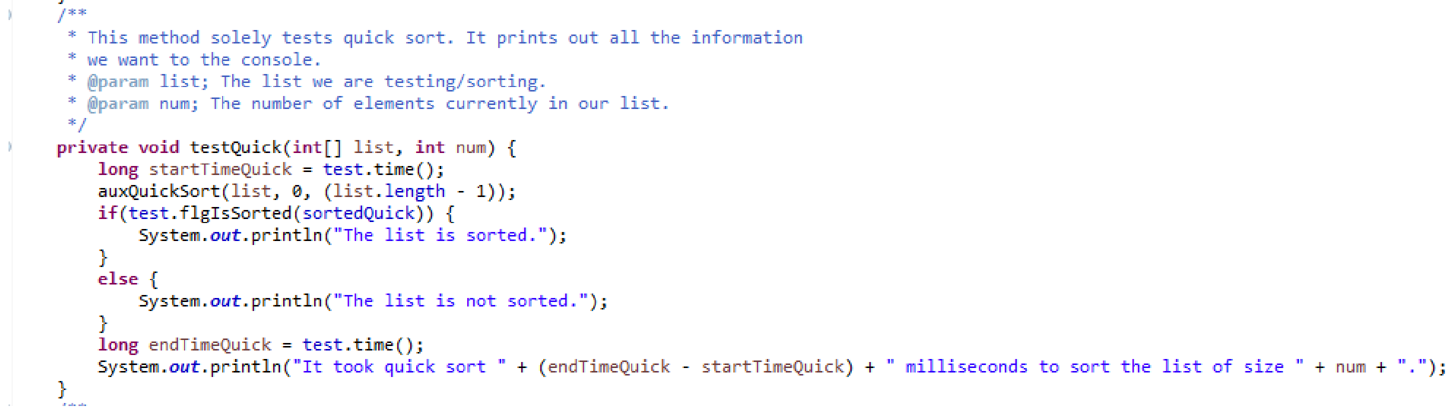
More code:



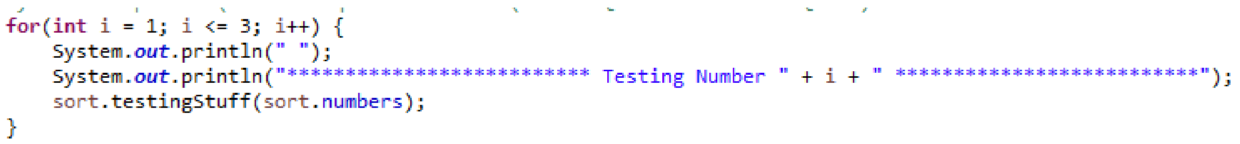
```java
    /**
     * This method does the splitting of the arrays into pieces so we can
     * do #5 of the lab. It splits the array and then calls the auxiliary testing
     * methods for each sort.
     * @param list; The list we are testing.
     */
    public void testingStuff(int[] list) {
        int[] reset = list;
        sortedQuick = list;
        sortedMerge = null;
        for(int i = 1000; i <= reset.length; i = i*10) {
            int[] testList = new int[i];
            for(int j = 0; j < i; j++) {
                testList[j] = list[j];
            }
            testMerge(testList, i);
            testQuick(testList, i);
        }
    }
```



```java
    /**
     * This method solely tests merge sort. It prints out all the information
     * we want to the console.
     * @param list; The list we are testing/sorting.
     * @param num; The number of elements currently in our list.
     */
    private void testMerge(int[] list, int num) {
        long startTimeMerge = test.time();
        auxMergeSort(list, 0, (list.length - 1));
        if(test.flgIsSorted(sortedMerge)) {
            System.out.println("The list is sorted.");
        }
        else {
            System.out.println("The list is not sorted.");
        }
        long endTimeMerge = test.time();
        System.out.println("It took merge sort " + (endTimeMerge - startTimeMerge) + " milliseconds to sort the list of size "+ num + ".");
    }
```

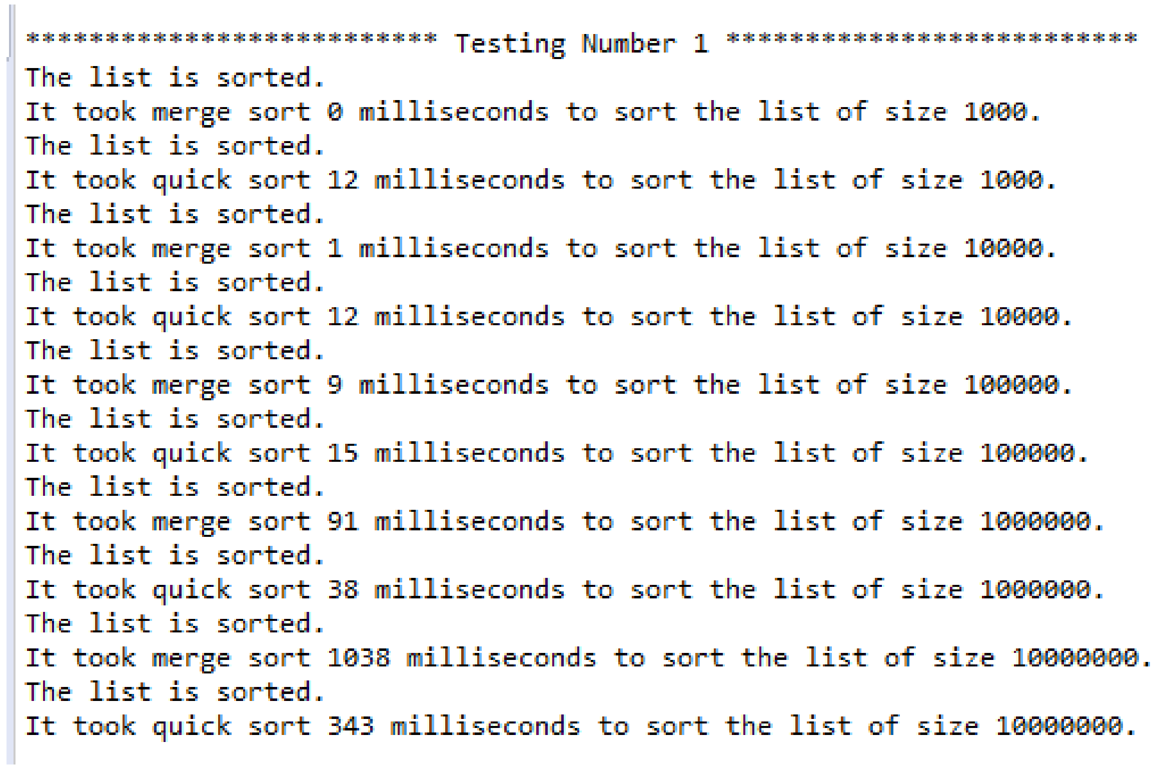


```java
    /**
     * This method solely tests quick sort. It prints out all the information
     * we want to the console.
     * @param list; The list we are testing/sorting.
     * @param num; The number of elements currently in our list.
     */
    private void testQuick(int[] list, int num) {
        long startTimeQuick = test.time();
        auxQuickSort(list, 0, (list.length - 1));
        if(test.flgIsSorted(sortedQuick)) {
            System.out.println("The list is sorted.");
        }
        else {
            System.out.println("The list is not sorted.");
        }
        long endTimeQuick = test.time();
        System.out.println("It took quick sort " + (endTimeQuick - startTimeQuick) + " milliseconds to sort the list of size " + num + ".");
    }
```

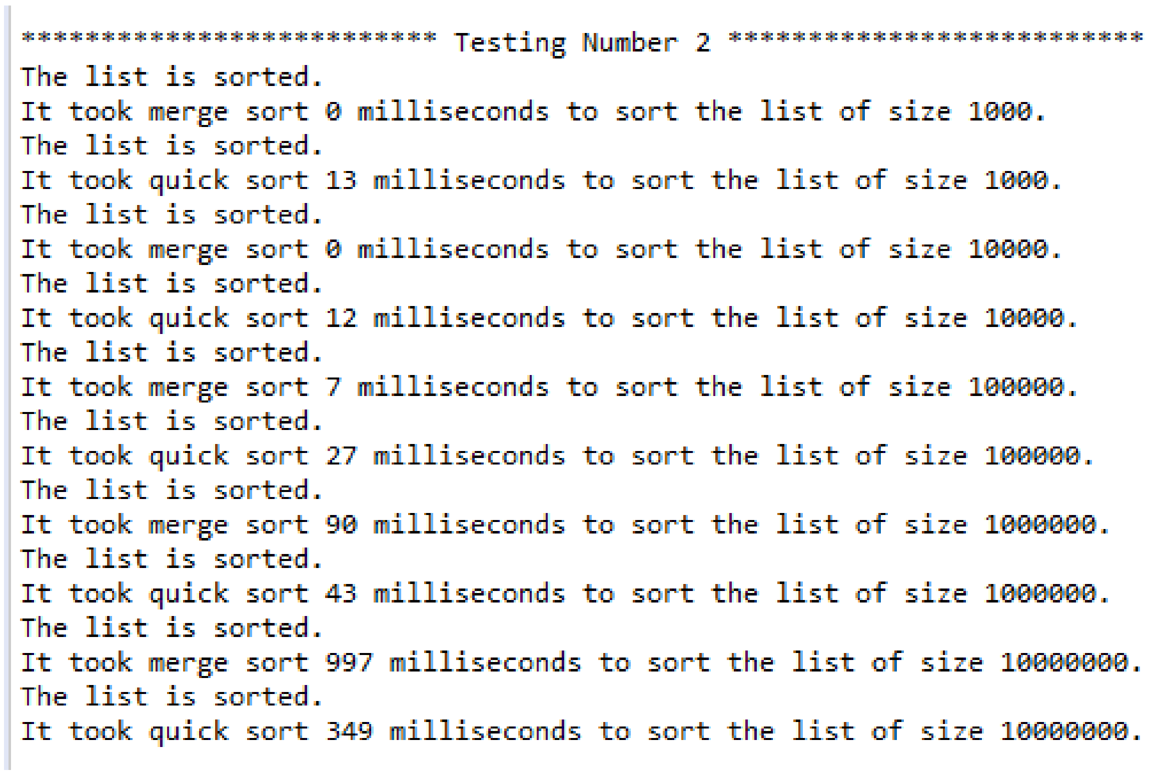


```
for(int i = 1; i <= 3; i++) {
    System.out.println(" ");
    System.out.println("************************* Testing Number " + i + " *************************");
    sort.testingStuff(sort.numbers);
}
```

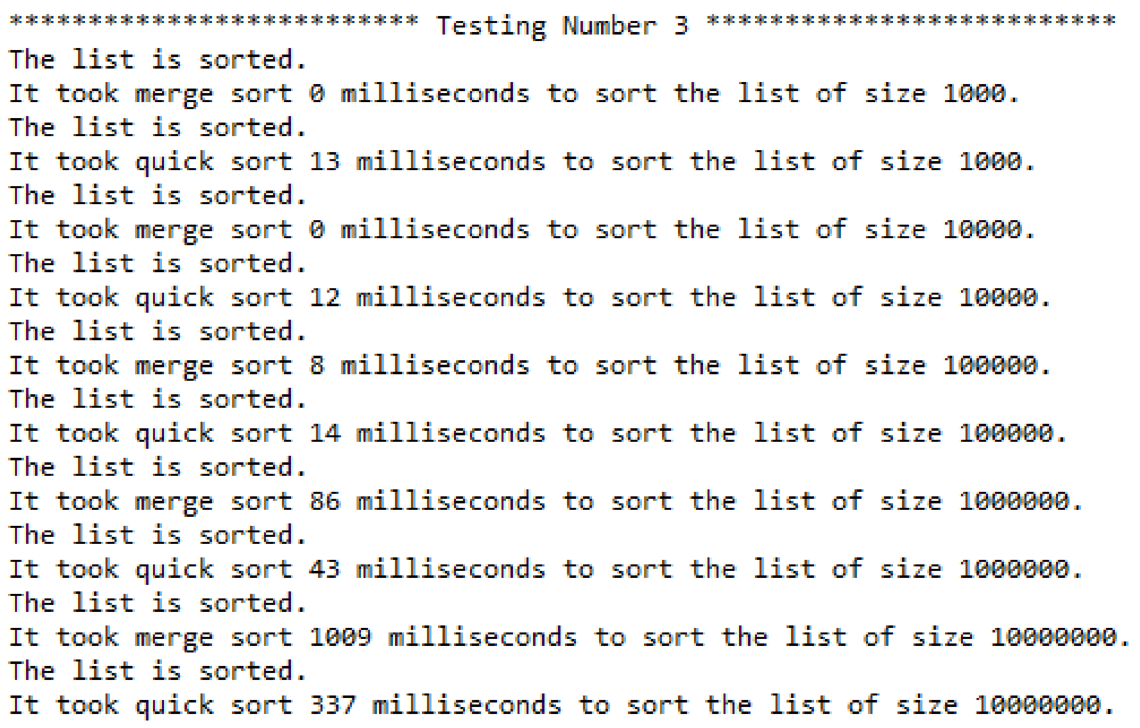
Output:



```
************************** Testing Number 1 **************************
The list is sorted.
It took merge sort 0 milliseconds to sort the list of size 1000.
The list is sorted.
It took quick sort 12 milliseconds to sort the list of size 1000.
The list is sorted.
It took merge sort 1 milliseconds to sort the list of size 10000.
The list is sorted.
It took quick sort 12 milliseconds to sort the list of size 10000.
The list is sorted.
It took merge sort 9 milliseconds to sort the list of size 100000.
The list is sorted.
It took quick sort 15 milliseconds to sort the list of size 100000.
The list is sorted.
It took merge sort 91 milliseconds to sort the list of size 1000000.
The list is sorted.
It took quick sort 38 milliseconds to sort the list of size 1000000.
The list is sorted.
It took merge sort 1038 milliseconds to sort the list of size 10000000.
The list is sorted.
It took quick sort 343 milliseconds to sort the list of size 10000000.
```

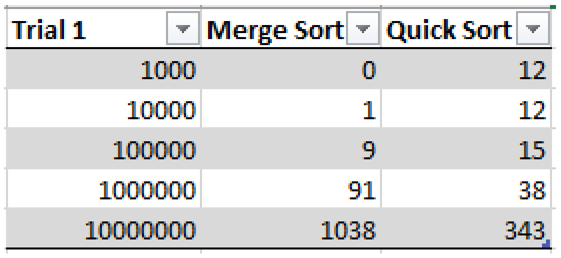


```
************************** Testing Number 2 **************************
The list is sorted.
It took merge sort 0 milliseconds to sort the list of size 1000.
The list is sorted.
It took quick sort 13 milliseconds to sort the list of size 1000.
The list is sorted.
It took merge sort 0 milliseconds to sort the list of size 10000.
The list is sorted.
It took quick sort 12 milliseconds to sort the list of size 10000.
The list is sorted.
It took merge sort 7 milliseconds to sort the list of size 100000.
The list is sorted.
It took quick sort 27 milliseconds to sort the list of size 100000.
The list is sorted.
It took merge sort 90 milliseconds to sort the list of size 1000000.
The list is sorted.
It took quick sort 43 milliseconds to sort the list of size 1000000.
The list is sorted.
It took merge sort 997 milliseconds to sort the list of size 10000000.
The list is sorted.
It took quick sort 349 milliseconds to sort the list of size 10000000.
```

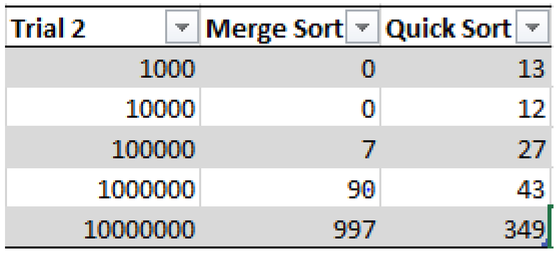


```
************************** Testing Number 3 **************************
The list is sorted.
It took merge sort 0 milliseconds to sort the list of size 1000.
The list is sorted.
It took quick sort 13 milliseconds to sort the list of size 1000.
The list is sorted.
It took merge sort 0 milliseconds to sort the list of size 10000.
The list is sorted.
It took quick sort 12 milliseconds to sort the list of size 10000.
The list is sorted.
It took merge sort 8 milliseconds to sort the list of size 100000.
The list is sorted.
It took quick sort 14 milliseconds to sort the list of size 100000.
The list is sorted.
It took merge sort 86 milliseconds to sort the list of size 1000000.
The list is sorted.
It took quick sort 43 milliseconds to sort the list of size 1000000.
The list is sorted.
It took merge sort 1009 milliseconds to sort the list of size 10000000.
The list is sorted.
It took quick sort 337 milliseconds to sort the list of size 10000000.
```

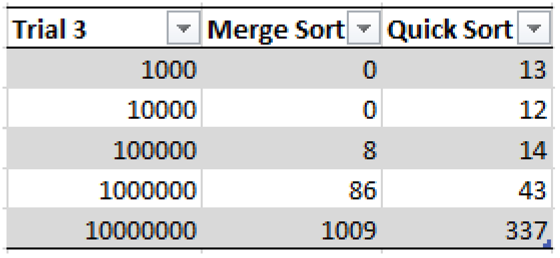
Graph and Tables:



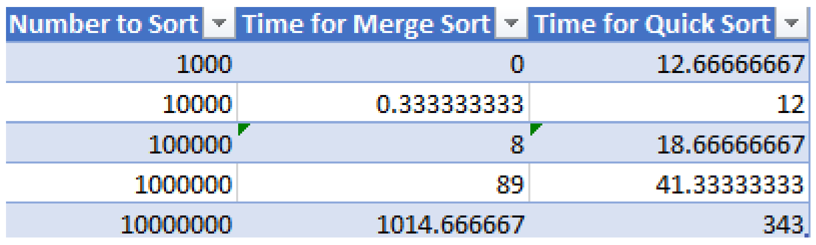
Trial 1	Merge Sort	Quick Sort
1000	0	12
10000	1	12
100000	9	15
1000000	91	38
10000000	1038	343



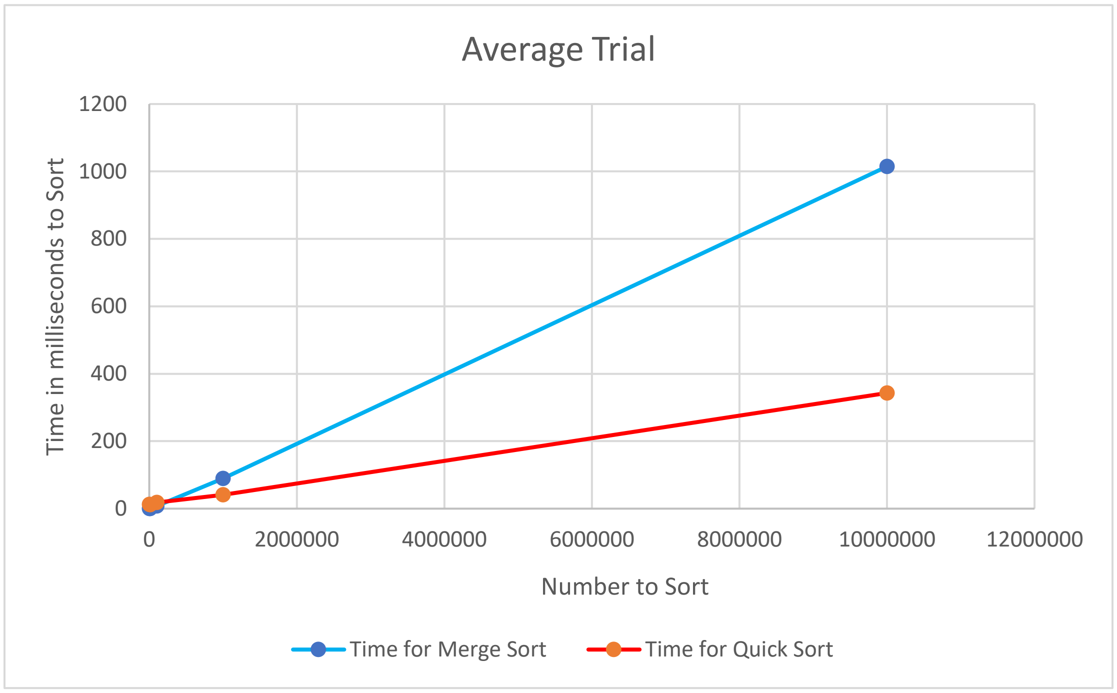
Trial 2	Merge Sort	Quick Sort
1000	0	13
10000	0	12
100000	7	27
1000000	90	43
10000000	997	349



Trial 3	Merge Sort	Quick Sort
1000	0	13
10000	0	12
100000	8	14
1000000	86	43
10000000	1009	337



Number to Sort	Time for Merge Sort	Time for Quick Sort
1000	0	12.66666667
10000	0.333333333	12
100000	8	18.66666667
1000000	89	41.33333333
10000000	1014.666667	343



Analysis of Data: It looks like for the smaller numbers merge sort is faster, but as the number of elements increases, quick sort has a faster sorting time. This is probably because quick sort has a smaller constant than merge sort as “n” gets large.

Final Thoughts: For most of this lab I used the pseudocode from the book. For the reading file part, I used my “Objects First with BlueJ” book from CS 161 and 162. Everything else I used is cited in the comments of the code. I thought this lab was fun and not too challenging. The flgIsSorted method was super fun to think about. I figured that one out on my own using knowledge from merge sort and thinking about what the hints said. The testing stuff was fun to write too. I talked to some people outside of our class about formatting stuff. (I basically just asked how they did it…). Stacia Fry also helped me find a bug in my Quick Sort method that was causing a StackOverflowError. Other than that, I basically did the lab by myself. I really enjoyed it. (That’s why I got it done like 2 weeks early.) Hopefully this is what you are looking for.