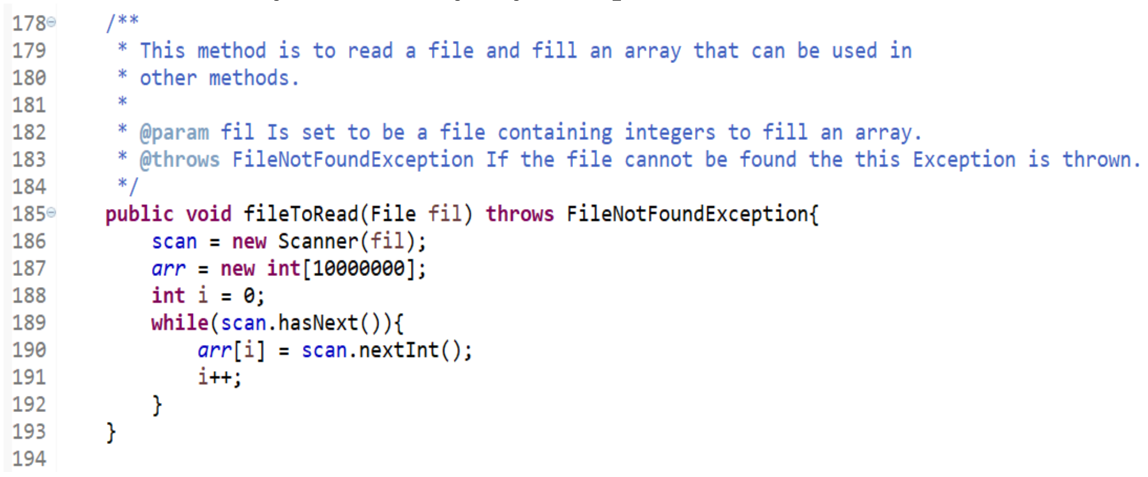
Nathan Stark

CS 361

Lab 1

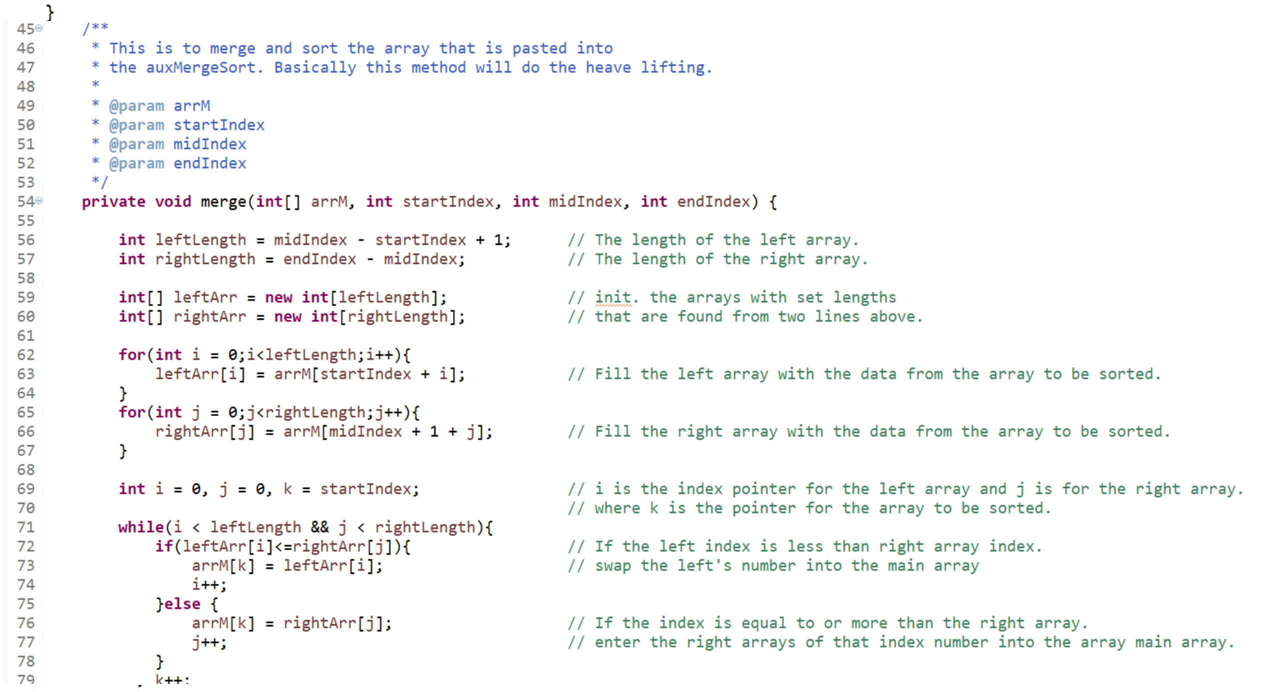
1. **Read the test data from the data file provided into an array of integers. The file contains 10,000,000 integers, one line per integer. If you read an empty line, then skip it. The sum of these integers is 49,999,995,000,000. Make sure you are using this information to verify the accuracy of your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



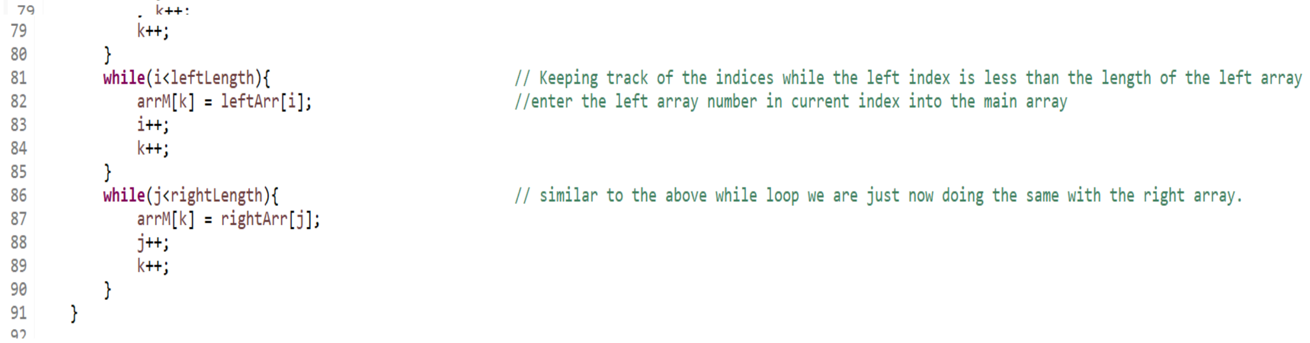
```
    /**
     * This method is to read a file and fill an array that can be used in
     * other methods.
     *
     * @param fil Is set to be a file containing integers to fill an array.
     * @throws FileNotFoundException If the file cannot be found the this Exception is thrown.
     */
    public void fileToRead(File fil) throws FileNotFoundException{
        scan = new Scanner(fil);
        arr = new int[10000000];
        int i = 0;
        while(scan.hasNext()){
            arr[i] = scan.nextInt();
            i++;
        }
    }
```

2. **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.**

```java
public class CS361Labs {
    //Scanner to read the lab files.
    private Scanner scan;
    //The array that is made from the lab file given.
    private static int[] arr;

    /* Much of the below code was helped and found on geeks for geeks https://www.geeksforgeeks.org/merge-sort/*/

    /**
     * Sort an array with merge sort.
     *
     * @param arrM The array that merge sort will be ran on.
     * @param startIndex the starting index of the array.
     * @param endIndex the length of the array minus one.
     */
    public void auxMergeSort(int[] arrM, int startIndex, int endIndex){

        int midIndex; // set an index for the mid point.

        if(startIndex < endIndex){
            midIndex = startIndex + (endIndex -startIndex)/2; // init. the midIndex marking the midpoint of the array.
                                                              // From geeks for geeks I have changed my midIndex equation
                                                              // as this should avoid over flow from large start and end
                                                              // indexes as that is what we will be using.

            auxMergeSort(arrM, startIndex, midIndex);         // Recursion. The method calls its self with
            auxMergeSort(arrM, midIndex + 1, endIndex);       // the midIndex in order to brake down the array.

            merge(arrM, startIndex, midIndex, endIndex);      //The arrays are then merged and sorted for this method.
        }
    }
```

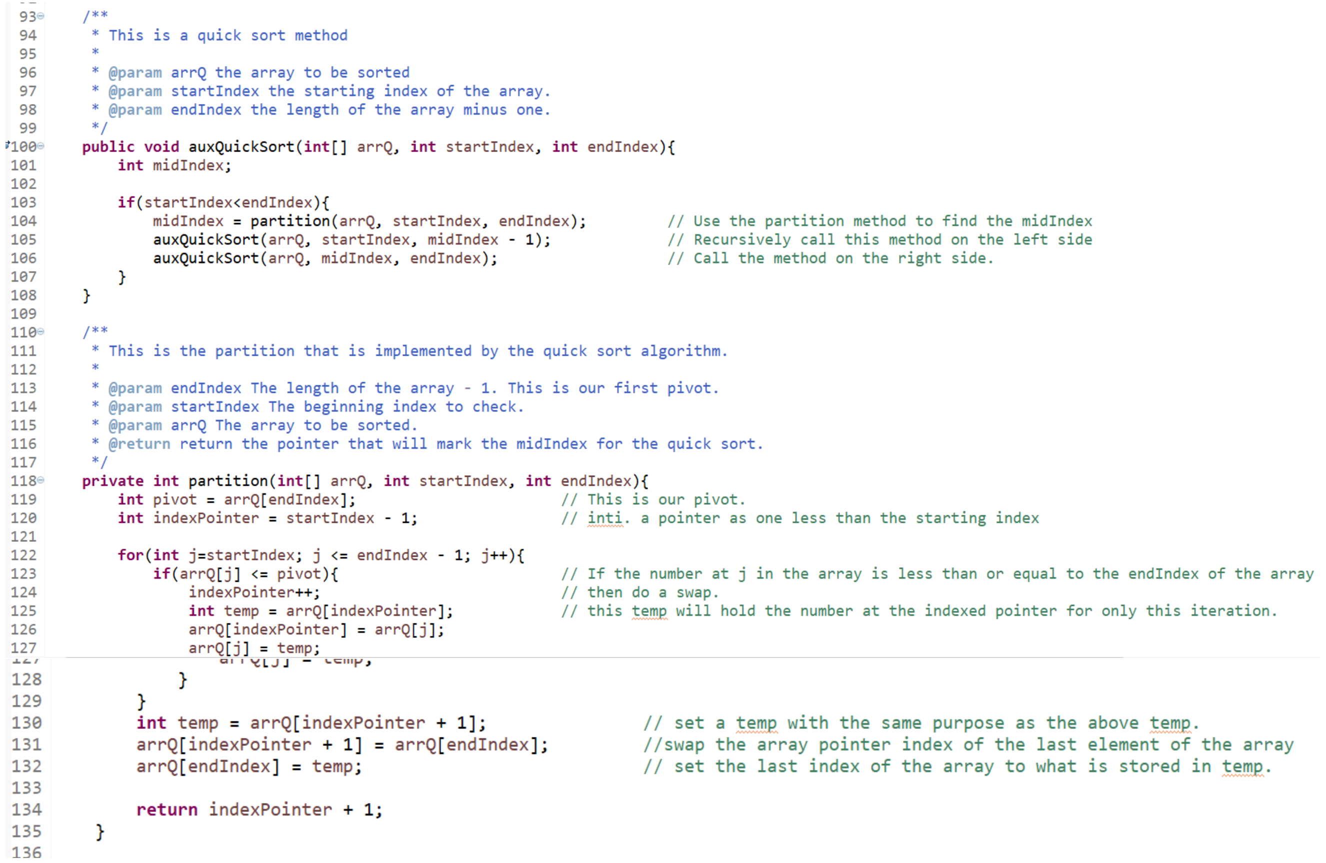


```java
    /**
     * This is to merge and sort the array that is pasted into
     * the auxMergeSort. Basically this method will do the heave lifting.
     *
     * @param arrM
     * @param startIndex
     * @param midIndex
     * @param endIndex
     */
    private void merge(int[] arrM, int startIndex, int midIndex, int endIndex) {

        int leftLength = midIndex - startIndex + 1;      // The length of the left array.
        int rightLength = endIndex - midIndex;           // The length of the right array.

        int[] leftArr = new int[leftLength];             // init. the arrays with set lengths
        int[] rightArr = new int[rightLength];           // that are found from two lines above.

        for(int i = 0;i<leftLength;i++){
            leftArr[i] = arrM[startIndex + i];           // Fill the left array with the data from the array to be sorted.
        }
        for(int j = 0;j<rightLength;j++){
            rightArr[j] = arrM[midIndex + 1 + j];        // Fill the right array with the data from the array to be sorted.
        }

        int i = 0, j = 0, k = startIndex;                // i is the index pointer for the left array and j is for the right array.
                                                         // where k is the pointer for the array to be sorted.
        while(i < leftLength && j < rightLength){
            if(leftArr[i]<=rightArr[j]){                 // If the left index is less than right array index.
                arrM[k] = leftArr[i];                    // swap the left's number into the main array
                i++;
            }else {
                arrM[k] = rightArr[j];                   // If the index is equal to or more than the right array.
                j++;                                     // enter the right arrays of that index number into the array main array.
            }
```



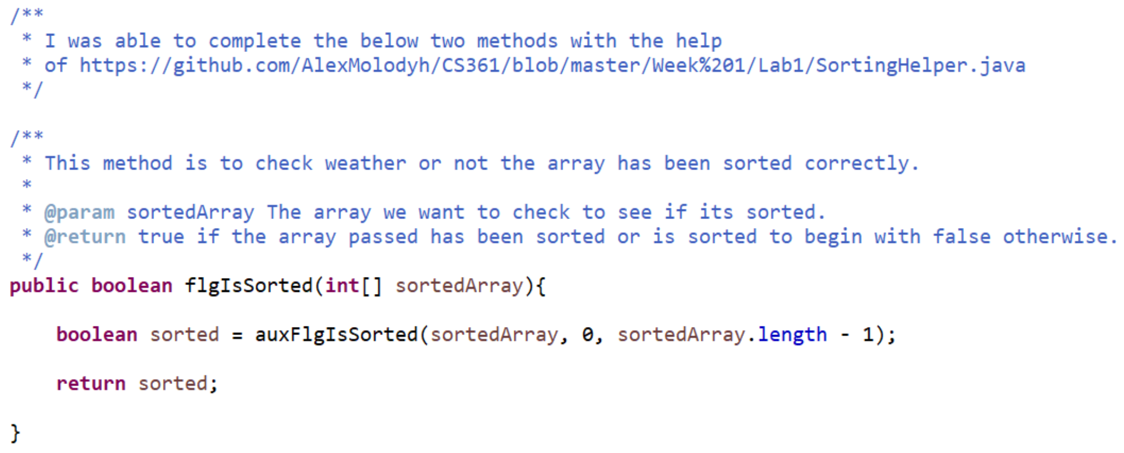
```java
            k++;
        }
        while(i<leftLength){                         // Keeping track of the indices while the left index is less than the length of the left array
            arrM[k] = leftArr[i];                    //enter the left array number in current index into the main array
            i++;
            k++;
        }
        while(j<rightLength){                        // similar to the above while loop we are just now doing the same with the right array.
            arrM[k] = rightArr[j];
            j++;
            k++;
        }
    }
```

3. **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.**

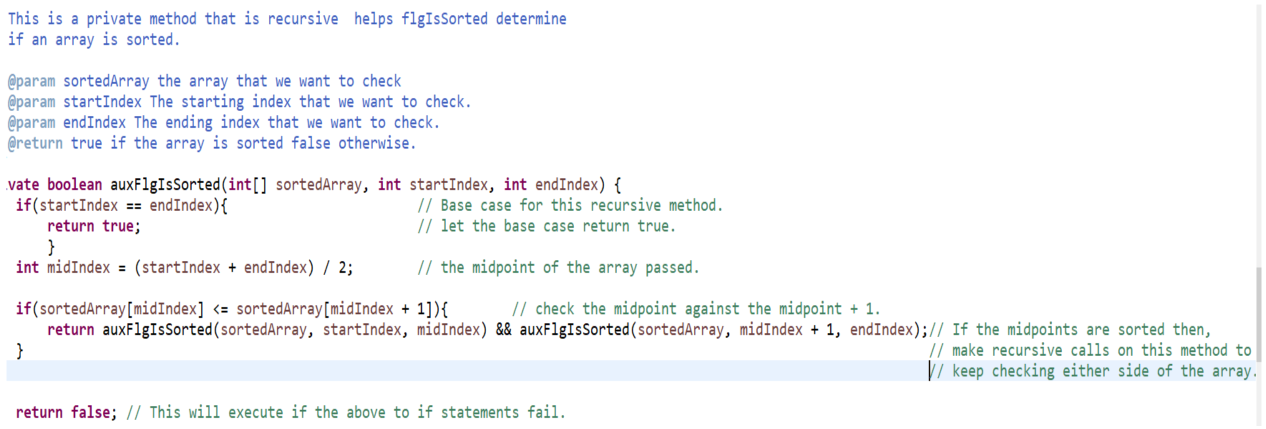


```java
/**
 * This is a quick sort method
 *
 * @param arrQ the array to be sorted
 * @param startIndex the starting index of the array.
 * @param endIndex the length of the array minus one.
 */
public void auxQuickSort(int[] arrQ, int startIndex, int endIndex){
    int midIndex;

    if(startIndex<endIndex){
        midIndex = partition(arrQ, startIndex, endIndex);         // Use the partition method to find the midIndex
        auxQuickSort(arrQ, startIndex, midIndex - 1);             // Recursively call this method on the left side
        auxQuickSort(arrQ, midIndex, endIndex);                   // Call the method on the right side.
    }
}

/**
 * This is the partition that is implemented by the quick sort algorithm.
 *
 * @param endIndex The length of the array - 1. This is our first pivot.
 * @param startIndex The beginning index to check.
 * @param arrQ The array to be sorted.
 * @return return the pointer that will mark the midIndex for the quick sort.
 */
private int partition(int[] arrQ, int startIndex, int endIndex){
    int pivot = arrQ[endIndex];                         // This is our pivot.
    int indexPointer = startIndex - 1;                  // inti. a pointer as one less than the starting index

    for(int j=startIndex; j <= endIndex - 1; j++){
        if(arrQ[j] <= pivot){                           // If the number at j in the array is less than or equal to the endIndex of the array
            indexPointer++;                             // then do a swap.
            int temp = arrQ[indexPointer];              // this temp will hold the number at the indexed pointer for only this iteration.
            arrQ[indexPointer] = arrQ[j];
            arrQ[j] = temp;
        }
    }
    int temp = arrQ[indexPointer + 1];                  // set a temp with the same purpose as the above temp.
    arrQ[indexPointer + 1] = arrQ[endIndex];            //swap the array pointer index of the last element of the array
    arrQ[endIndex] = temp;                              // set the last index of the array to what is stored in temp.

    return indexPointer + 1;
}
```

4. **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.**

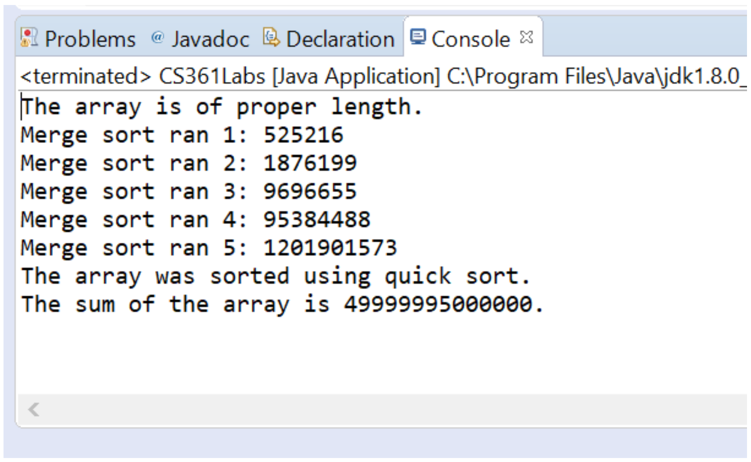


```java
/**
 * I was able to complete the below two methods with the help
 * of https://github.com/AlexMolodyh/CS361/blob/master/Week%201/Lab1/SortingHelper.java
 */

/**
 * This method is to check weather or not the array has been sorted correctly.
 *
 * @param sortedArray The array we want to check to see if its sorted.
 * @return true if the array passed has been sorted or is sorted to begin with false otherwise.
 */
public boolean flgIsSorted(int[] sortedArray){

    boolean sorted = auxFlgIsSorted(sortedArray, 0, sortedArray.length - 1);

    return sorted;

}
```



```java
This is a private method that is recursive  helps flgIsSorted determine
if an array is sorted.

@param sortedArray the array that we want to check
@param startIndex The starting index that we want to check.
@param endIndex The ending index that we want to check.
@return true if the array is sorted false otherwise.

vate boolean auxFlgIsSorted(int[] sortedArray, int startIndex, int endIndex) {
 if(startIndex == endIndex){                        // Base case for this recursive method.
     return true;                                   // let the base case return true.
     }
 int midIndex = (startIndex + endIndex) / 2;        // the midpoint of the array passed.

 if(sortedArray[midIndex] <= sortedArray[midIndex + 1]){        // check the midpoint against the midpoint + 1.
     return auxFlgIsSorted(sortedArray, startIndex, midIndex) && auxFlgIsSorted(sortedArray, midIndex + 1, endIndex);// If the midpoints are sorted then,
 }                                                                                                                    // make recursive calls on this method to
                                                                                                                      // keep checking either side of the array.

 return false; // This will execute if the above to if statements fail.
```

5. **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.**



```
<terminated> CS361Labs [Java Application] C:\Program Files\Java\jdk1.8.0_
The array is of proper length.
Merge sort ran 1: 525216
Merge sort ran 2: 1876199
Merge sort ran 3: 9696655
Merge sort ran 4: 95384488
Merge sort ran 5: 1201901573
The array was sorted using quick sort.
The sum of the array is 49999995000000.
```

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.
```

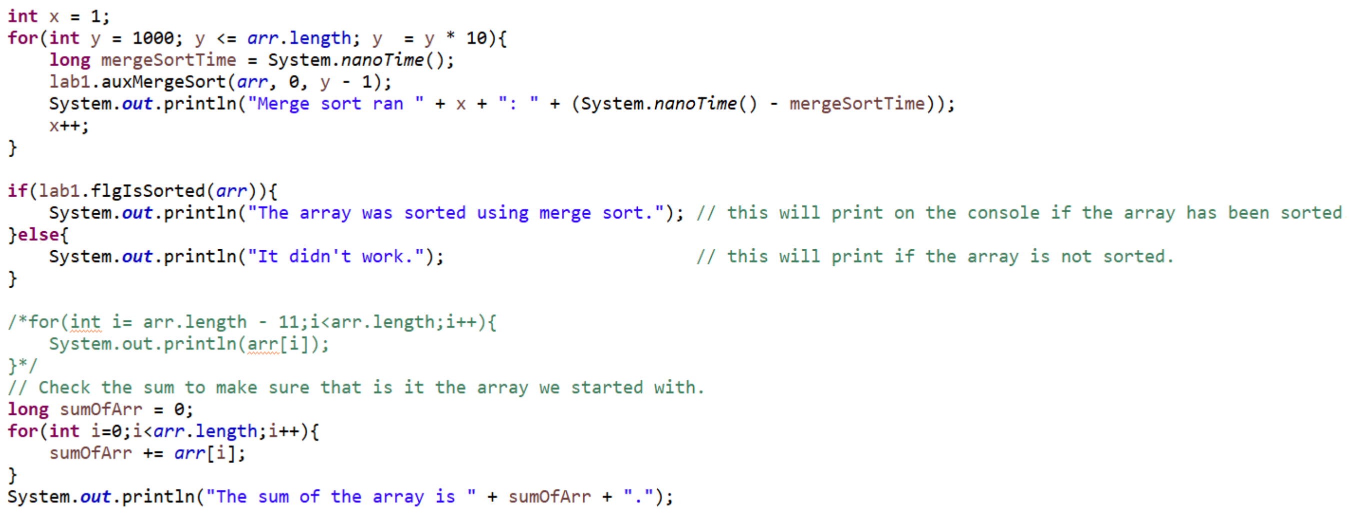
The second time that I ran for merge sort through the loop.

```
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.



```
int x = 1;
for(int y = 1000; y <= arr.length; y  = y * 10){
    long mergeSortTime = System.nanoTime();
    lab1.auxMergeSort(arr, 0, y - 1);
    System.out.println("Merge sort ran " + x + ": " + (System.nanoTime() - mergeSortTime));
    x++;
}

if(lab1.flgIsSorted(arr)){
    System.out.println("The array was sorted using merge sort."); // this will print on the console if the array has been sorted
}else{
    System.out.println("It didn't work.");                        // this will print if the array is not sorted.
}

/*for(int i= arr.length - 11;i<arr.length;i++){
    System.out.println(arr[i]);
}*/
// Check the sum to make sure that is it the array we started with.
long sumOfArr = 0;
for(int i=0;i<arr.length;i++){
    sumOfArr += arr[i];
}
System.out.println("The sum of the array is " + sumOfArr + ".");
```

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.
```

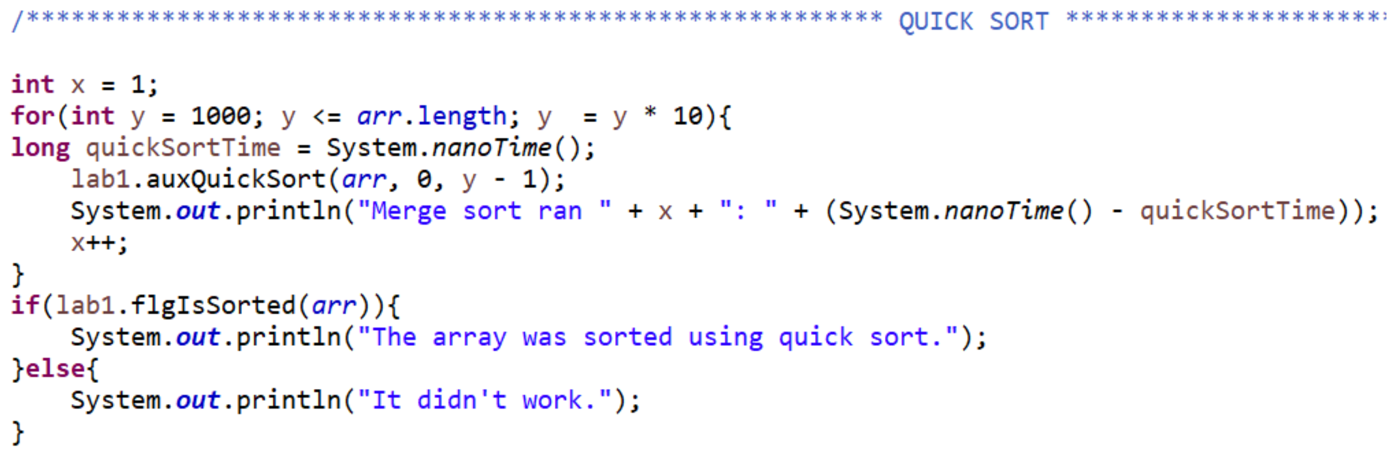
The second time that I ran for quick sort through the loop.

```
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.
```

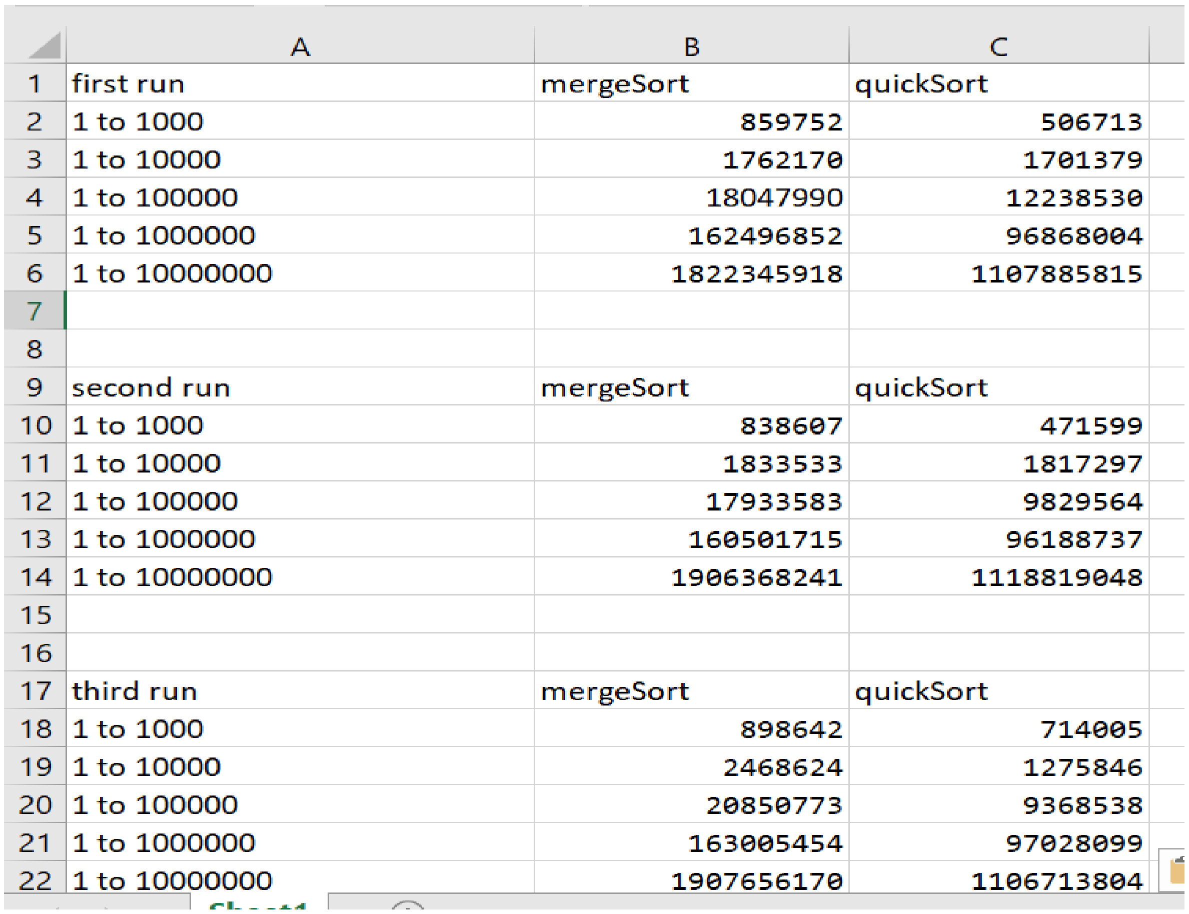
The third time that I ran for quick sort through the loop.

```
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.



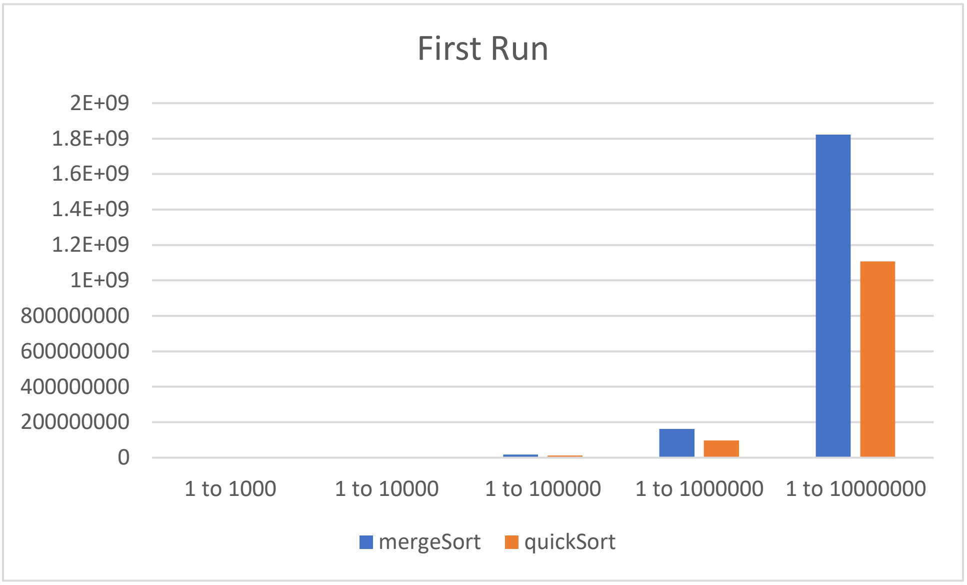
```java
/****************************************************** QUICK SORT *********************

int x = 1;
for(int y = 1000; y <= arr.length; y  = y * 10){
long quickSortTime = System.nanoTime();
    lab1.auxQuickSort(arr, 0, y - 1);
    System.out.println("Merge sort ran " + x + ": " + (System.nanoTime() - quickSortTime));
    x++;
}
if(lab1.flgIsSorted(arr)){
    System.out.println("The array was sorted using quick sort.");
}else{
    System.out.println("It didn't work.");
}
```

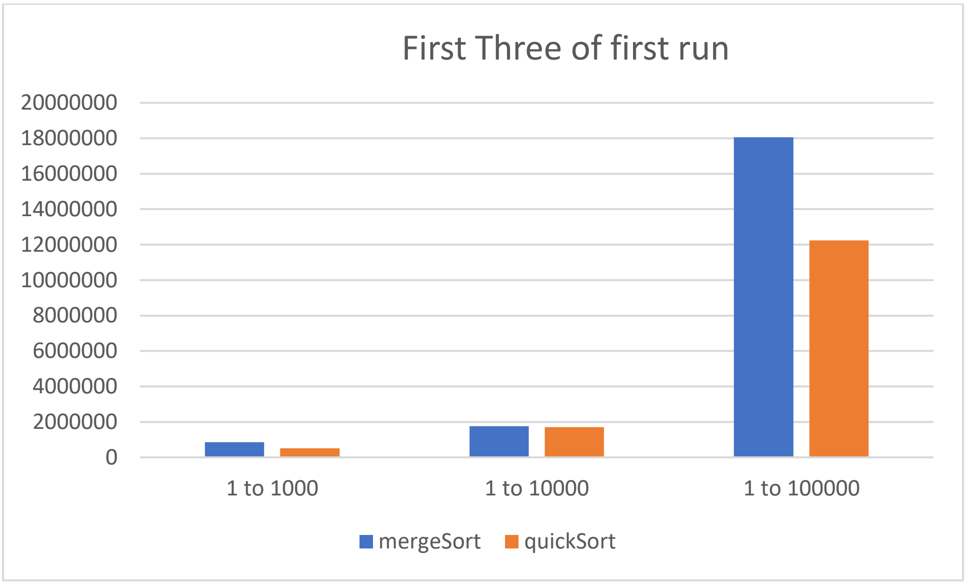


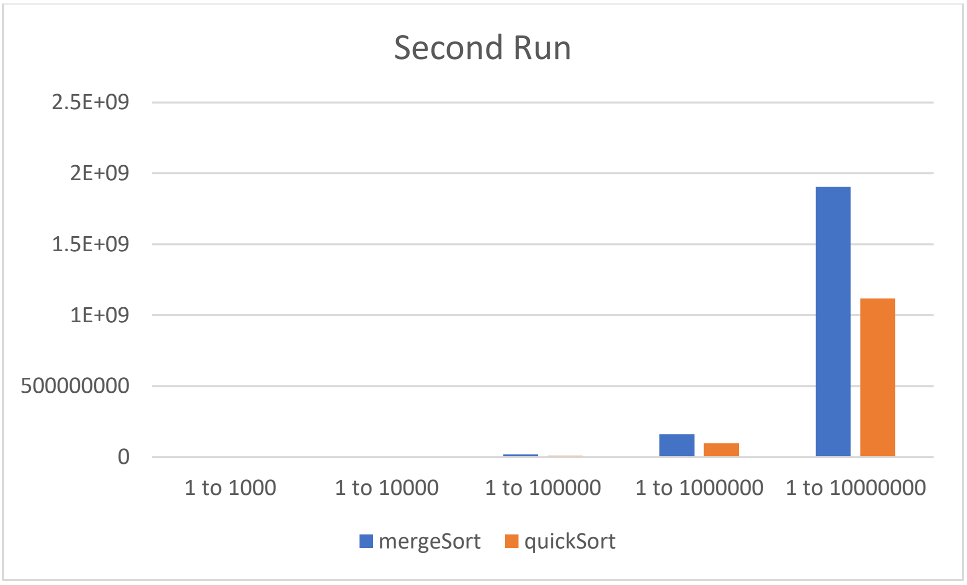
	A	B	C
1	first run	mergeSort	quickSort
2	1 to 1000	859752	506713
3	1 to 10000	1762170	1701379
4	1 to 100000	18047990	12238530
5	1 to 1000000	162496852	96868004
6	1 to 10000000	1822345918	1107885815
7			
8			
9	second run	mergeSort	quickSort
10	1 to 1000	838607	471599
11	1 to 10000	1833533	1817297
12	1 to 100000	17933583	9829564
13	1 to 1000000	160501715	96188737
14	1 to 10000000	1906368241	1118819048
15			
16			
17	third run	mergeSort	quickSort
18	1 to 1000	898642	714005
19	1 to 10000	2468624	1275846
20	1 to 100000	20850773	9368538
21	1 to 1000000	163005454	97028099
22	1 to 10000000	1907656170	1106713804

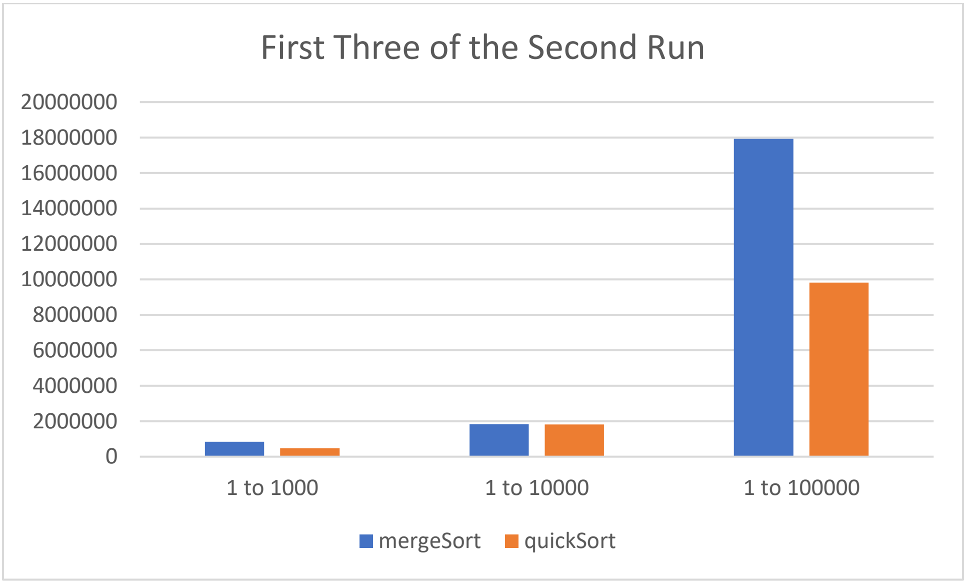
Because it's hard to read the first three results from merge and quick sort being ran against all five I added a second chart so that we can more clearly see what's happening.

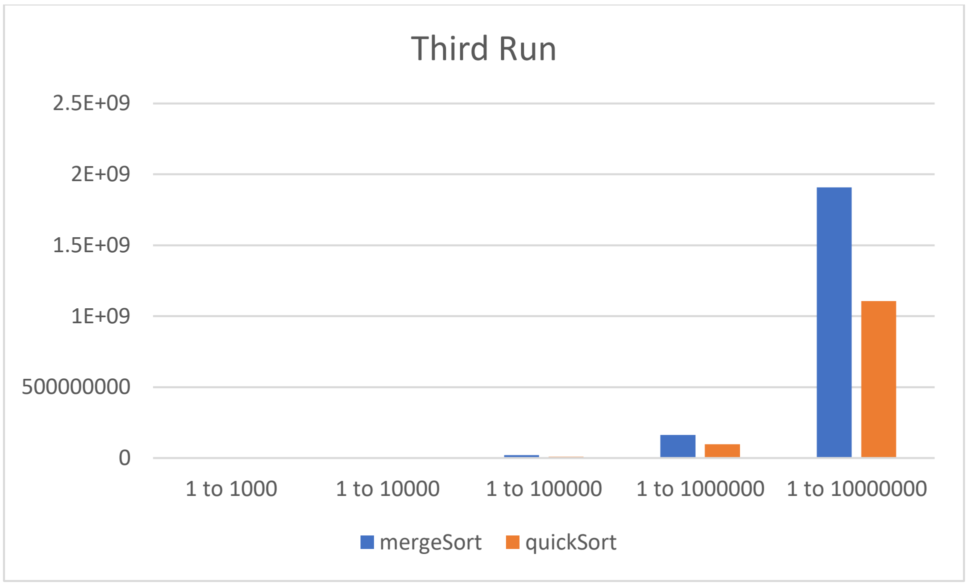
Form this first run we can clearly see that quicksort is faster than merge sort for all iterations.

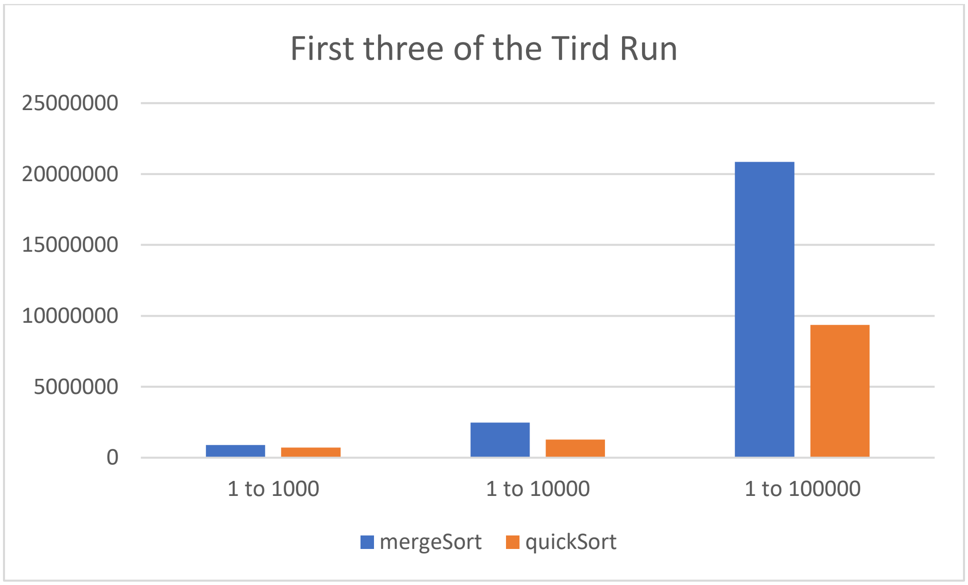





Second Run
2.5E+09
2E+09
1.5E+09
1E+09
500000000
0
1 to 1000
1 to 10000
1 to 100000
1 to 1000000
1 to 10000000
mergeSort
quickSort


First Three of the Second Run
20000000
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
1 to 1000
1 to 10000
1 to 100000
mergeSort
quickSort





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