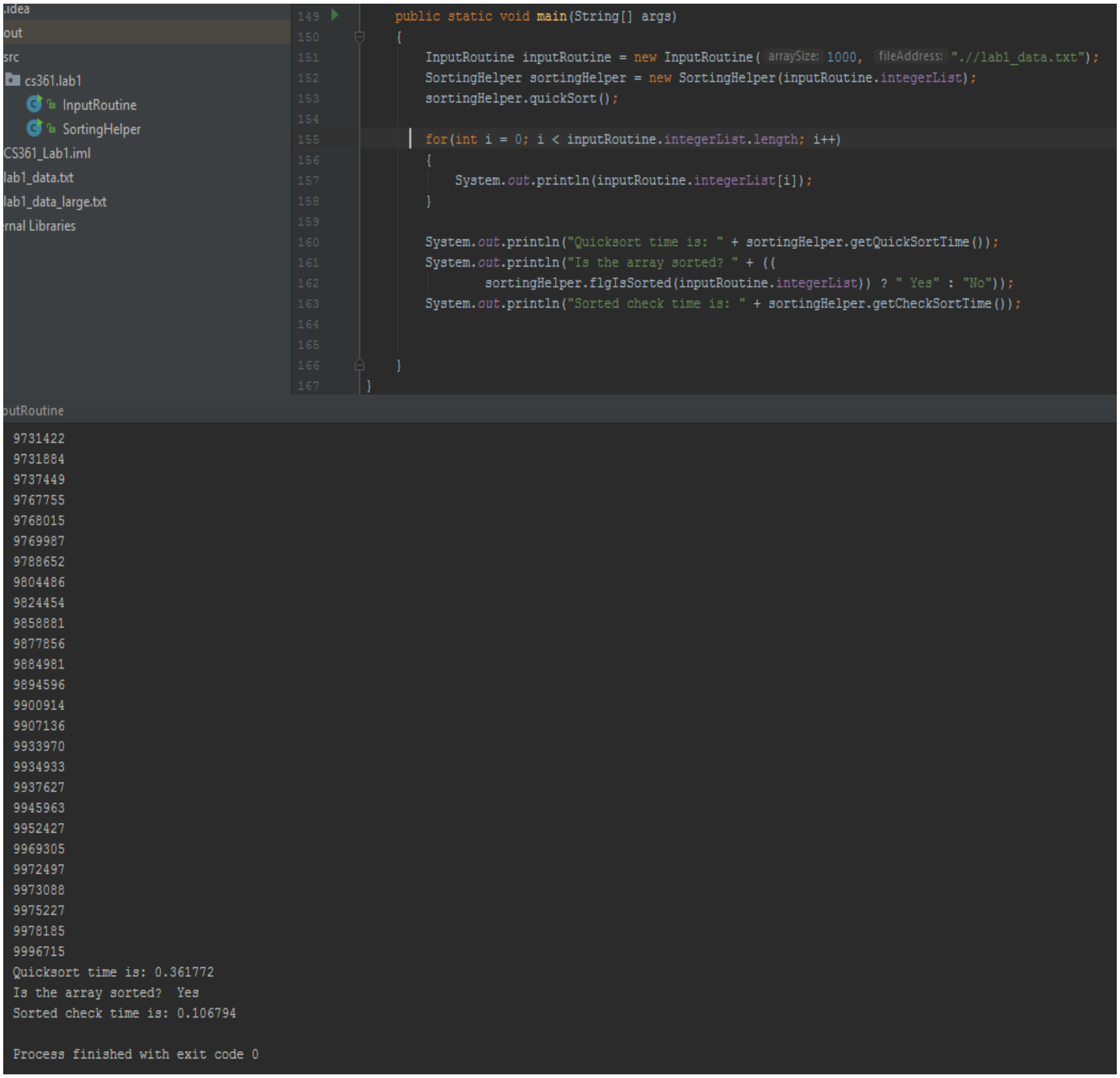
Author: Alexander Molodyh

Class: CS361

Date: April 13, 2017

While running both the Mergesort and Quicksort algorithms, I have found that the Quicksort algorithm is faster than the Mergesort algorithm in execution time. The difference between the average execution time for the Mergesort and the Quicksort isn't very big but it's a difference none the less. Even though the worst case complexity of Quicksort is O(n^2) as opposed to the Mergesort worst case complexity that is O(nlogn), Quicksort still has less of an execution time than Mergesort but only when the array is in random order. When running a Quicksort and a Mergesort on a sorted array, the Mergesort execution time is not affected by this, but the Quicksort can't even finish sorting the array without running in to a StackOverflowError on larger arrays. Even when sorting smaller arrays like 1000-2000 in size, Quicksort still racks up about 30 milliseconds as opposed to .6 milliseconds for the Mergesort. Unless you are completely certain that your data will always be completely random, I would not use the Quicksort for this reason. The Mergesort is slower than Quicksort by a small amount but it is guaranteed to take less time when sorting a partially or fully sorted list of data.

The code that I used to print out my results.



```java
    public static void main(String[] args)
    {
        InputRoutine inputRoutine = new InputRoutine( arraySize: 1000,  fileAddress: ".//lab1_data.txt");
        SortingHelper sortingHelper = new SortingHelper(inputRoutine.integerList);
        sortingHelper.quickSort();

        for(int i = 0; i < inputRoutine.integerList.length; i++)
        {
            System.out.println(inputRoutine.integerList[i]);
        }

        System.out.println("Quicksort time is: " + sortingHelper.getQuickSortTime());
        System.out.println("Is the array sorted? " + ((
                sortingHelper.flgIsSorted(inputRoutine.integerList)) ? " Yes" : "No"));
        System.out.println("Sorted check time is: " + sortingHelper.getCheckSortTime());


    }
}
```

```
9731422
9731884
9737449
9767755
9768015
9769987
9788652
9804486
9824454
9858881
9877856
9884981
9894596
9900914
9907136
9933970
9934933
9937627
9945963
9952427
9969305
9972497
9973088
9975227
9978185
9996715
Quicksort time is: 0.361772
Is the array sorted?  Yes
Sorted check time is: 0.106794

Process finished with exit code 0
```

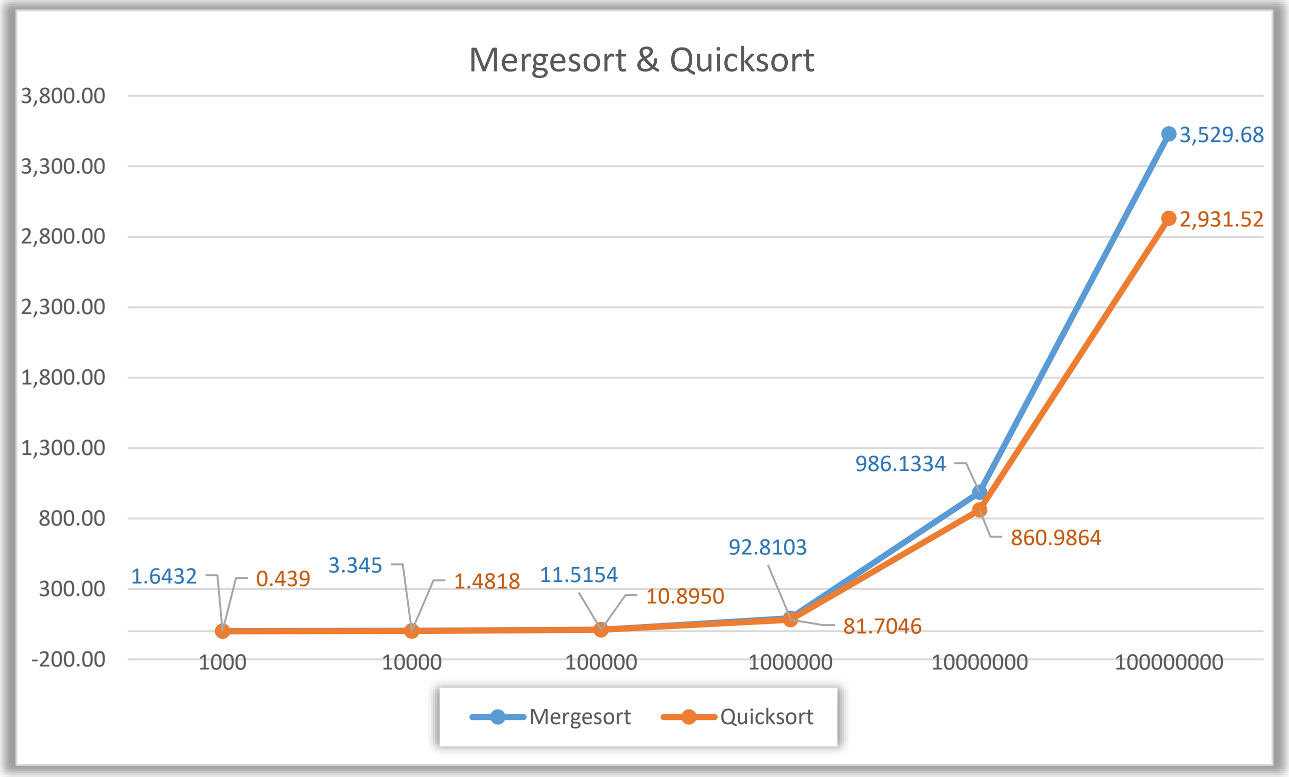
Mergesort & Quicksort algorithm data table: The following table displays the Mergesort and Quicksort algorithm execution data.

- **Algorithm:** Displays the name of the algorithm
- **Array Size:** Displays the size of the array that was used
- **Run #1:** Displays the first time the sorting algorithm was run
- **Run #2:** Displays the second time the sorting algorithm was run
- **Run #3:** Displays the third time the sorting algorithm was run
- **Avg Runtime:** Displays the average runtime of the 3 runs
- **Sorted Check:** Displays the result from the flgIsSorted method
    - **Yes:** Means the array was sorted
    - **No:** Means the array was not sorted
- **Sorted Runtime:** Displays the time it took to check if the array was sorted or not

Algorithm	Array Size	Run #1	Run #2	Run #3	Avg Runtime	Sorted Check	Sorted Runtime
Mergesort	1,000	1.5229	1.8385	1.5682	1.6432	Yes	0.104495
Mergesort	10,000	3.0273	3.6617	3.3461	3.3450	Yes	0.283338
Mergesort	100,000	13.2525	10.7321	10.5616	11.5154	Yes	1.205654
Mergesort	1,000,000	92.6341	92.7789	93.0178	92.8103	Yes	4.417154
Mergesort	10,000,000	996.7418	983.2936	978.3650	986.1334	Yes	35.376574
Mergesort	100,000,000	3,575.8095	3,630.1499	3,383.0655	3,529.6750	Yes	343.999669
QuickSort	1,000	0.4750	0.3656	0.4765	0.4390	Yes	0.106028
QuickSort	10,000	1.3658	1.4435	1.6362	1.4818	Yes	0.252678
QuickSort	100,000	10.7487	10.2221	11.7142	10.8950	Yes	1.282045
QuickSort	1,000,000	81.0372	79.7171	84.3596	81.7046	Yes	4.584755
QuickSort	10,000,000	865.2762	859.2916	858.3913	860.9864	Yes	35.172694
QuickSort	100,000,000	2,937.4495	2,921.9124	2,935.2058	2,931.5225	Yes	344.147087

Mergesort & Quicksort algorithms: This chart displays the average running time for each algorithm on the different sized arrays.

- **X-Axis:** The size of the array used
- **Y-Axis:** The average runtime for the algorithm it took to sort the array of size **x**



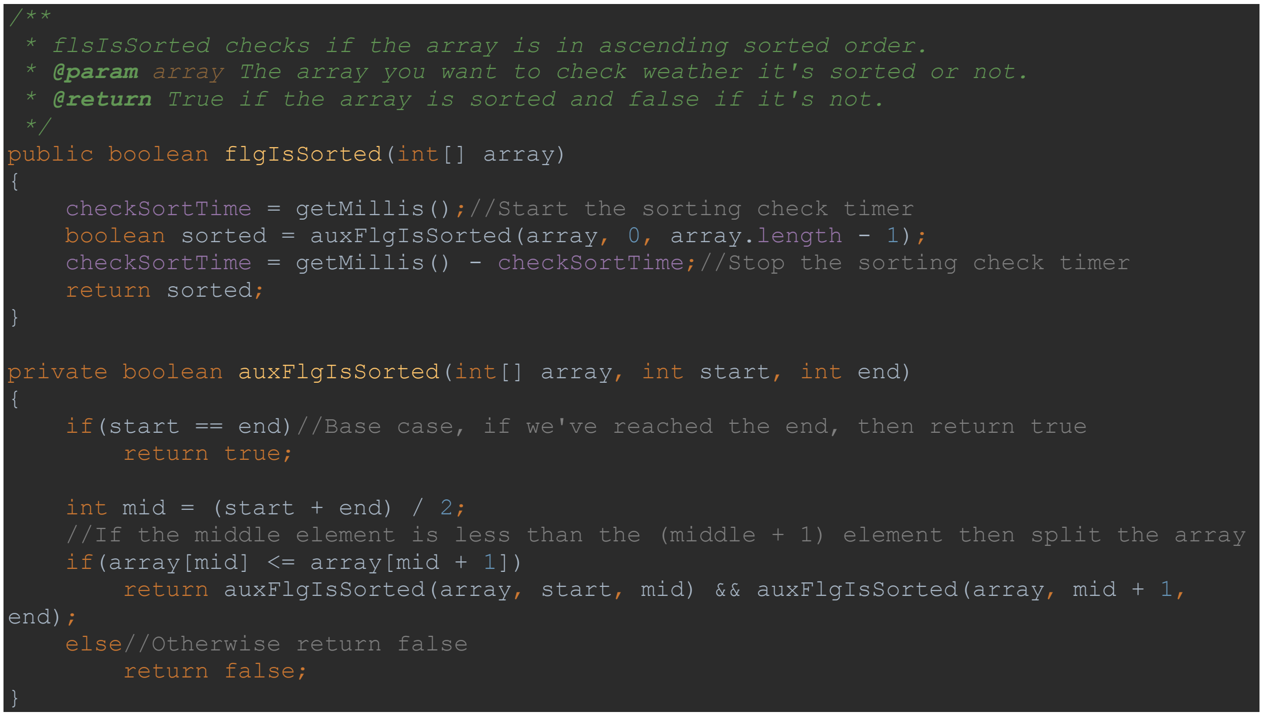
Mergesort algorithm code: The following code is for the Mergesort algorithm.

```
/**
 * mergeSort performs a Mergesort sorting algorithm on the array that was passed in to
 * the SortingHelper object.
 */
public void mergeSort()
{
    mergeSortTime = getMillis();//Start the mergesort clock
    auxMergeSort(arrayContainer, 0, arrayContainer.length - 1);
    mergeSortTime = getMillis() - mergeSortTime;//Store the time it took to sort the
array
}

/**
 * auxMergeSort takes and array and splits it in to two arrays and calls on itself on
 * each of the array half's.
 *
 * @param start The start of the array.
 * @param end   Indicates the end of the array.
 */
private void auxMergeSort(int[] array, int start, int end)
{
    if(start < end)
    {
        //Get the midpoint of the array
        int mid = (int) Math.floor((start + end) / 2);

        auxMergeSort(array, start, mid);//Start Mergesort on left side array
        auxMergeSort(array, mid + 1, end);//Start Mergesort on right side array
        merge(array, start, mid, end);//Start merging the arrays
    }
}

/**
 * merge takes two arrays and merges them together and at the same time sorts
 * it in ascending order.(It doesn't take two arrays, it just uses the starting, mid,
and ending points
 * to traverse through the array).
 *
 * @param start The start of the first array.
 * @param mid   The end of the first array and the beginning of the second array(mid +
1) is the beginning of
 *              the second array.
 * @param end   The end of the second array.
 */
private void merge(int[] array, int start, int mid, int end)
{
    //Variables to travers left and right side of array
    int k = start;
    int g = start;
    int j = mid + 1;

    //Populate arrayHelper with the section of values from arrayContainer
    for(int i = start; i <= end; i++)
    {
        arrayHelper[i] = array[i];
    }

    /*
    This section traverses through the left array and the right array and stores the
    lowest values from the arrayHelper in to the arrayContainer.
     */
    while(k <= mid && j <= end)//loop through left and right array
    {
        if(arrayHelper[k] <= arrayHelper[j])//Check if left side element is smaller
                                            //than right side element
        {
            array[g] = arrayHelper[k];//Read element from arrayHelper and store it in
                                      //arrayContainer
            k++;
        }
        else//Right side is smaller, store right side element in to arrayContainer
        {
            array[g] = arrayHelper[j];
            j++;
        }
        g++;
    }

    /*
     * Loop through the rest of the remaining elements and store them in the array
     */
    while(k <= mid)
    {
        array[g] = arrayHelper[k];
        k++;
        g++;
    }
}
```

Quicksort algorithm code: The following code is for the Quicksort algorithm.

```java
/**
 * quickSort performs a Quicksort sorting algorithm on the array that was passed in to
 * the SortingHelper object.
 */
public void quickSort()
{
    quickSortTime = getMillis();
    auxQuickSort(arrayContainer, 0, arrayContainer.length - 1);
    quickSortTime = getMillis() - quickSortTime;
}

private void auxQuickSort(int[] array, int start, int end)
{
    if(start < end)
    {
        int pivot = partition(array, start, end);

        auxQuickSort(array, start, pivot);
        auxQuickSort(array, pivot + 1, end);
    }
}

private int partition(int[] array, int start, int end)
{
    int pivot = array[start];//Pick the pivot to be the first element in array
    boolean done = false;
    int sIndex = start - 1;
    int eIndex = end + 1;

    while(!done)//Loop until low index and high index meet each other
    {
        do{//While element in start index is less than pivot increment start index
            sIndex++;
        }
        while(array[sIndex] < pivot);

        do{//While element in end index is greater than pivot decrement end index
            eIndex--;
        }
        while(array[eIndex] > pivot);

        //If start and end are the same index then return the start index
        if(sIndex >= eIndex)
            return eIndex;

        swap(array, sIndex, eIndex);//Swap the start and end index elements
    }
    return eIndex;//Return the midpoint
}
```

Sorting check method: The image on the very bottom is a screen dump from the flgIsSorted() method. After I sorted the array I ran it through a for loop to output the elements in ascending order just to show that they are all sorted. The image below is the code for the flgIsSorted() method.



```
/**
 * flsIsSorted checks if the array is in ascending sorted order.
 * @param array The array you want to check weather it's sorted or not.
 * @return True if the array is sorted and false if it's not.
 */
public boolean flgIsSorted(int[] array)
{
    checkSortTime = getMillis();//Start the sorting check timer
    boolean sorted = auxFlgIsSorted(array, 0, array.length - 1);
    checkSortTime = getMillis() - checkSortTime;//Stop the sorting check timer
    return sorted;
}

private boolean auxFlgIsSorted(int[] array, int start, int end)
{
    if(start == end)//Base case, if we've reached the end, then return true
        return true;

    int mid = (start + end) / 2;
    //If the middle element is less than the (middle + 1) element then split the array
    if(array[mid] <= array[mid + 1])
        return auxFlgIsSorted(array, start, mid) && auxFlgIsSorted(array, mid + 1,
end);
    else//Otherwise return false
        return false;
}
```