Benchmarking Merge-Sort, Iterative v Recursive

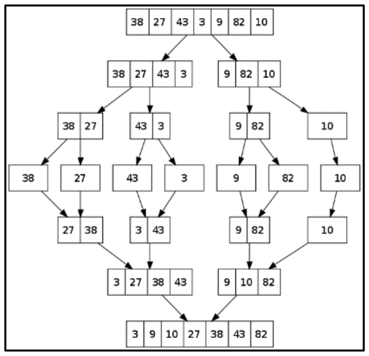
Trevon McHansen

10 March 2019

Introduction:

Merge sort is often considered one of the most universally effective traditional sorting algorithms due to its consistency. In fact, one of the more modern sorting algorithms known as Tim-sort utilizes merge sort principles in conjunction with quicksort and is now used as the default sorting method in the Python framework!

Merge-sort utilizes a divide and conquer algorithmic approach to break a large array down into smaller and smaller subcomponents until only a couple of values are being compared, and then the sub-arrays are then merged back together to form a total sort. The visual representation of this was retrieved from Software Engineering Stack Exchange (https://i.stack.imgur.com/lF95K.png).



Both the Iterative and Recursive variations of this algorithm follow this basic principle. The iterative variation contains two methods: merge and sort. The sort method breaks down the algorithm into smaller pieces while the merge method re-aligns all the elements into a single array. The recursive variation; however, requires only one method (as is the nature of recursion). It is cleaner and breaks the arrays into “left” and “right” arrays, recursively, until finally reaching a small enough sub-array to compare individual elements, and then merges the elements locally in a final loop of the method.

High-level Pseudocode for the sorting algorithms

```
//Iterative MergeSort

// Code retrieved from:

// parameters: array of elements to be sorted, temporary (empty) array, first element, middle element, last element

// merge

While (first element < middle element && middle element < last element)

    If first element (i) < mid element (j), add i to temp array

    Else, add j to temp array

// remaining elements added to temp array

While (i < length of array && i < the middle element location)

    Add element to temp array

// all elements are now ‘partially sorted’ in the “temp” array

For (first element (i); end element; i++)
```

```
        Array A at element i receives the value of array temp at element i
// sort
// parameter: partially sorted Array A
Copy all elements of Array A to (new) temp array of same length
For (m = 1; m<= array length – lowest index ; m = 2*m
        For (i = lowest index; I < array Length; i += 2*m)
                First element = i
                middle element = i + m – 1
                last element = minimum of (I + 2 * m – 1 , (and) array Length);
Merge Again
```

```
// Recursive MergeSort
// Parameters: Unsorted Array (a)
        If the array is null, return null
        If array contains any elements
                Middle of array = length of array / 2
                New array left has length of middle of array a //(half the array)
                For (0 index, length of left, increment index)
                        Array left at index i = array a at index i
                New array right has length of length – middle of array a //(the other half)
                For (middle index; length of Array; increment index)
                        Array right at index i-middle = array a at index i
                Recursively sort left array
                Recursively sort right array
                Initialize counting variables i, j, and k to 0
                While (i < length of left && j < length of right)
                        if ( left < right )
                                array index k = array left index i, increment i
                        else
```

```
                    array index k = array right index j, increment j
                increment k
            while (i < length of left)
                array a index k = array left index i
                increment i, increment k
            while (j < length of right)
                array a index k = array right index j
                increment j, increment k
        return array a
```

__Big-Θ analysis of the two versions of the algorithm__

One of the best parts about merge-sort, and the primary reason I selected it for my project 1 benchmark, is the fact that the best-case (Big-Ω), worst-case (Big-O), and average-case (Big-Θ) are all O(n*log(n)). Of course, the best-case scenario would contain a much lower constant coefficient value than the worst-case constant coefficient. That is, a Big-Ω determination of merge-sort could yield O(1*n*log(n)), but the Big-O of the equation could be O(255*n*log(n)).

This is due to the natural destroy and reconstruct, divide and concur nature of the algorithm. Both the iterative and recursive variations of the algorithm “break” the initial array into subcomponents (iterative “A” and “temp”, recursive “left” and “right”) repeatedly to form smaller and smaller subarrays which are then sorted and reconstructed.

First, we’ll look at the iterative version of the algorithm in the order that events occur upon method call: (all O values in table to be treated at Θ)

Int lowVal = 0, highVal = A.length-1;	//O(1), O(1)
Int[] temp = Arrays.copyOf(A, A.length);	(most likely) approximately O(n)
For (int m = 1; m<highVal – lowVal; m=2m)	O(m) where m = (highVal – lowVal)/2
For (int I = lowVal; i<highVal; i+=2m)	O(i) where i = (highVal – lowVal)/2
Int from = i;	//O(1)
Int mid = I + m – 1	O(1)
Int to = Integer.min(I + 2m -1, highVal)	O(n) because of the minimum method
Merge(A, temp, from, mid, to)	See Below

Int k = from, I = from, j = mid+1;	//O(1), O(1), O(1)
While i≤ mid && j≤ to	O(log(n))
If (A[i] < A[j]) { temp[k++] = A[i++]; }	O(1) { O(1) }
Else { temp[k++] = A[j++]; }	//O(1)
While (I < A.length && i≤ mid) { temp[k++] = A[i++];}	O(log(n))
For (I = from; i≤ to; i++)	O(n) where n is the length of the array
A[i] = temp[i]	O(1)

Assessment:

(italicized O(1) values do not have a direct impact on the efficiency of the algorithm)

O(1), O(1), O(n), O(log(n)), O(log(n)), *O(1),* O(1), O(n), *O(1), O(1), O(1),* O(log(n)), O(1), O(1), *O(1),* O(log(n)), O(n), O(1) => Θ(4*1), Θ(4*log(n)), Θ(3n) => **Θ(48*n*log(n))**

Next, we'll look at the Recursive variant of the algorithm:

If (array == null) return null;	O(1)
If(array.length > 1)	O(n-1)
Int mid = array.length / 2	O(1)
Int[] left = new int[mid];	O(1)
For(int I = 0; i<mid; i++)	O(n/2)
Left[i] = array[i]	O(1)
Int[] right = new int[array.length – mid]	O(1)
For (int I = mid; i<array.length; i++)	O(n/2)
RecursiveSort(left)	O(log(n))
RecursiveSort(right)	O(log(n))
While (i<left.length && j<right.length)	O(n/2)
If(left[i] < right[i]) { Array[k] = left[i]; i++ }	O(1), O(1), O(1)
Else { array[k] = right[j]; i++ }	O(1), O(1)
k++	O(1)
While (i<left.length) {}	O(n/2)
Array[k] = left[i]	O(1)
While(j < right.length) {}	O(n/2)
Array[k] = right[j];	O(1)

O(1), O(n-1), O(1), O(1), O(n/2), O(1), O(1), O(n/2), O(log(n)), O(log(n)), O(n/2), O(1), O(1), O(1), O(1), O(1), O(1), O(n/2), O(1), O(n/2), O(1) => Θ(n-1), Θ(5n/2), Θ(2log(n)), Θ(13*1) => Θ(3n), Θ(2log(n)), Θ(13) => **Θ(78n*log(n))**

Explanation of approach to avoiding problems with JVM warmup

I retrieved the basis for my JVM warmup from: (https://www.baeldung.com/java-jvm-warmup). I created a method called load() in the MergeSort.java class. The method iterates 10,000 times, each time creating a null object, converting the object to a String, and then, each iteration it will iterate 100,000 more times each time incrementing a fake long value.

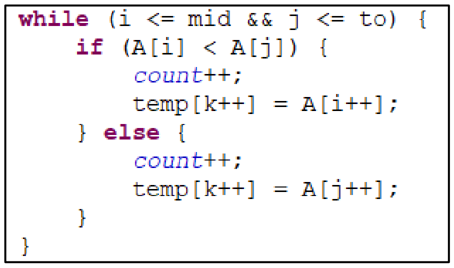
I found several different sites online each depicting a "right way" to warm up the JVM, but I found that the best way to do so was to perform an arbitrary event an arbitrary, but very large, number of times, to make sure that time is being spent getting the system up to speed. Much like pre-heating the oven before baking, I found that giving the JVM sufficient work to do beforehand made for much more consistent results. In order to avoid further issues, I stuck with these simple methods, although I had at one point considered writing a byte array output stream against the object value but determined that this would be an unnecessary step for the overall warmup process.

A discussion of the critical operations chosen to count with an explanation of why it was selected.

Iterative Critical Operations:

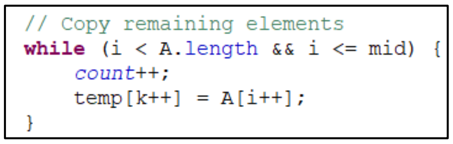
Merge Method:

For each iteration of the while loop, every time a value is added to the temporary array we count once. We count each iteration as we're doing a value check between variables as well as adding a variable to the temp array.



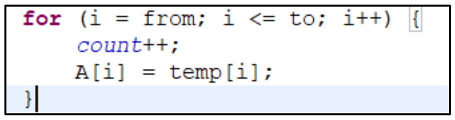
```
while (i <= mid && j <= to) {
    if (A[i] < A[j]) {
        count++;
        temp[k++] = A[i++];
    } else {
        count++;
        temp[k++] = A[j++];
    }
}
```

For each iteration of the while loop, every time we copy a value which has not already been added to the temporary array we count once. This ensures that we're counting every iteration in which we are adding values to the temp array and is counted, because we're repeatedly adding values to the array.



```
// Copy remaining elements
while (i < A.length && i <= mid) {
    count++;
    temp[k++] = A[i++];
}
```

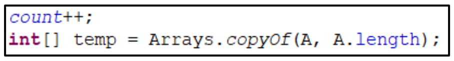
For each iteration of the for loop, wherein we copy all values from the partially-sorted temp array to the original array (A) we count once. This operation is coutned because it ensures that we're getting all values to the appropriate array.



```
for (i = from; i <= to; i++) {
    count++;
    A[i] = temp[i];
}
```

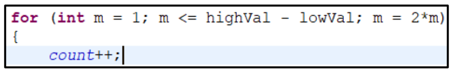
Sort Method:

Count once when we copy the array A to our new temporary Array. It is necessary to count here because we're creating a new array of values, which is necessary for the iterative Merge Sort method to work effectively.



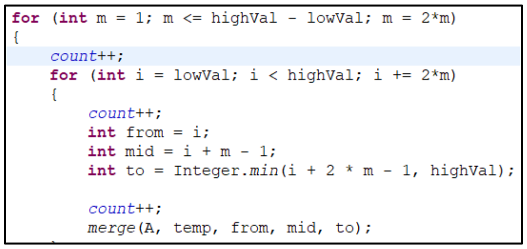
```
count++;
int[] temp = Arrays.copyOf(A, A.length);
```

For each iteration of the outer for loop, wherein we divide the array into blocks, we count one time. This outer-loop count is necessary to ensure that every time we re-enter the inner loop it is being counted.



```
for (int m = 1; m <= highVal - lowVal; m = 2*m)
{
    count++;
```

For each iteration of the inner for loop, wherein first determine the high, middle, and low points of the new array, and then call the merge method using the Array, temp array, low, middle, and high points, we count TWO times. Once for the mathematical determinations, and another time for calling the merge method. The mathematics here take some extra time and power for the processor, especially considering that it is going to happen m^2 times. Additionally, it is important to count every time we re-enter the merge method, as each time we enter we're requiring those parameters to be passed to new memory locations.



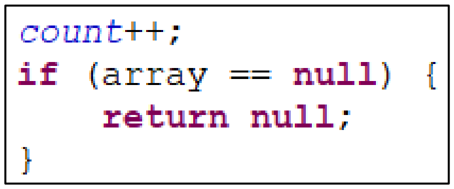
```
for (int m = 1; m <= highVal - lowVal; m = 2*m)
{
    count++;
    for (int i = lowVal; i < highVal; i += 2*m)
    {
        count++;
        int from = i;
        int mid = i + m - 1;
        int to = Integer.min(i + 2 * m - 1, highVal);

        count++;
        merge(A, temp, from, mid, to);
```

Total iterative critical operation counts: 7

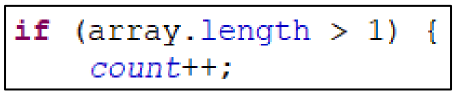
Recursive Critical Operations:

First, each time we check the recursive method to verify whether or not the array is null, we count once. This check is going to occur every time the iterative method is called, while it is typically false, it is imperative to denote the instances where we're going to be performing comparitive functions using boolean logic as it's a step that will be required every time we enter this method.



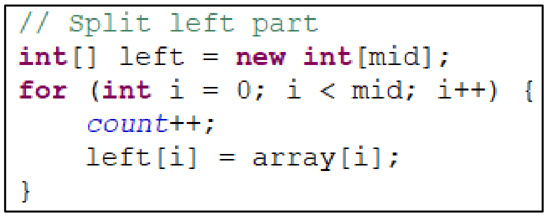
```
count++;
if (array == null) {
    return null;
}
```

After we verify that the array length is greater than one, if the condition is true we will count once. This count makes sure that we're including every time we begin to iterate inside of our recursive method. Even if only 2 elements are included, and later counts need not be determined, we've entered a new space for comparison and translation and should add to the count accordingly. This count also includes the creation of the new array and basic mathematical computation which immediately follow.



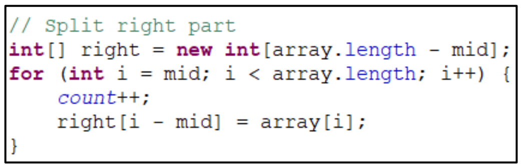
```
if (array.length > 1) {
    count++;
```

When we split the arrays into left and right, with the "left" being the lower half of the array and the "right" being the upper half of the array, a series of events occur. First, once the left array is constructed, we will iterate through every single value – for each iteration here we will count once, then add the left value at index i to the array at index i. This accounts for 1 count per iterative performance.



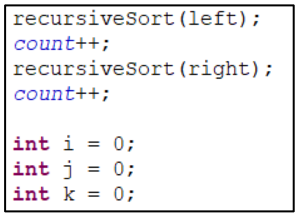
```
// Split left part
int[] left = new int[mid];
for (int i = 0; i < mid; i++) {
    count++;
    left[i] = array[i];
}
```

Immediately following, we'll iterate through all the values of the right array – for each iteration here we will count once again, and add the first value of the right array (0 index) [i – mid] to the middle value of the array i (where i = mid). This accounts for 1 counts per iterative performance.



```
// Split right part
int[] right = new int[array.length - mid];
for (int i = mid; i < array.length; i++) {
    count++;
    right[i - mid] = array[i];
}
```

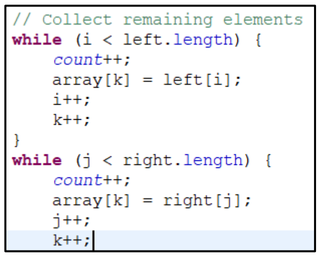
Once the arrays are split, we will perform the recursive sort on both the "left" array and the "right" array, we count one time each as we're re-entering the recursive method once per method call.



```
recursiveSort(left);
count++;
recursiveSort(right);
count++;

int i = 0;
int j = 0;
int k = 0;
```

Next, we need to merge the left and right arrays back together. At this point we assume that both the left and right arrays have sufficiently recursively sorted. So, for each array we create a while loop, while there is still a next index, we count once (in each loop = 2) and align the values of the array to the values of left and right respectively.



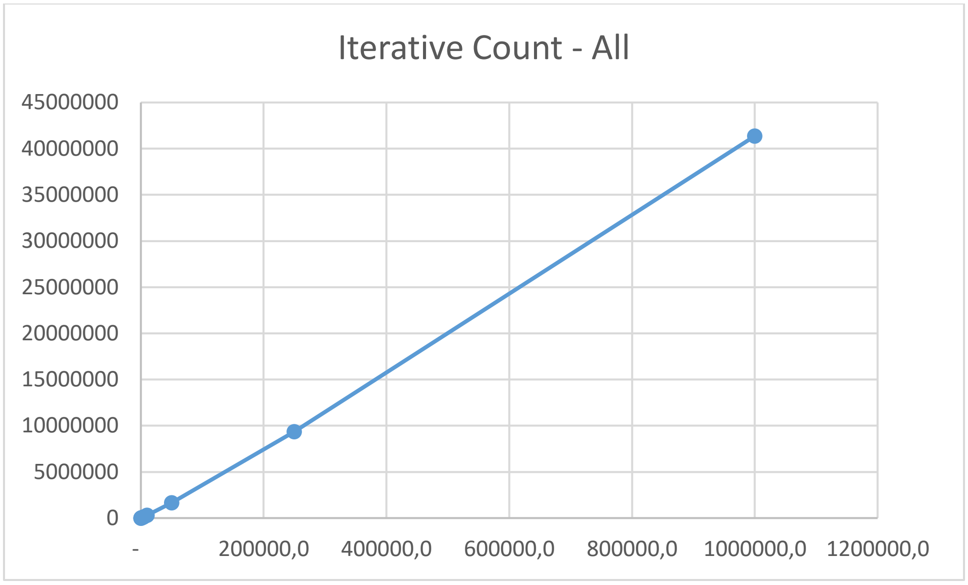
```
// Collect remaining elements
while (i < left.length) {
    count++;
    array[k] = left[i];
    i++;
    k++;
}
while (j < right.length) {
    count++;
    array[k] = right[j];
    j++;
    k++;
```

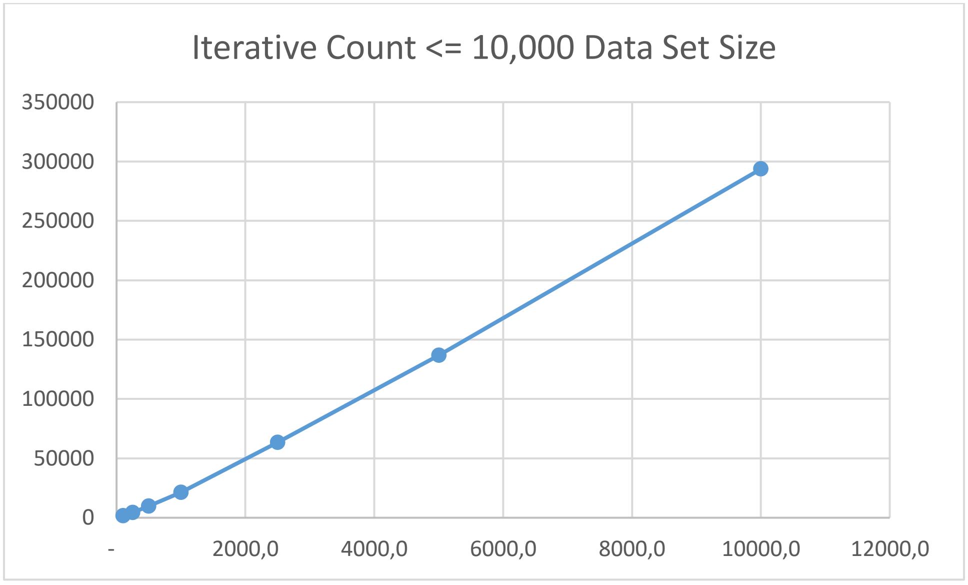
*Total recursive critical operation counts: **8***

<u>Analysis:</u>

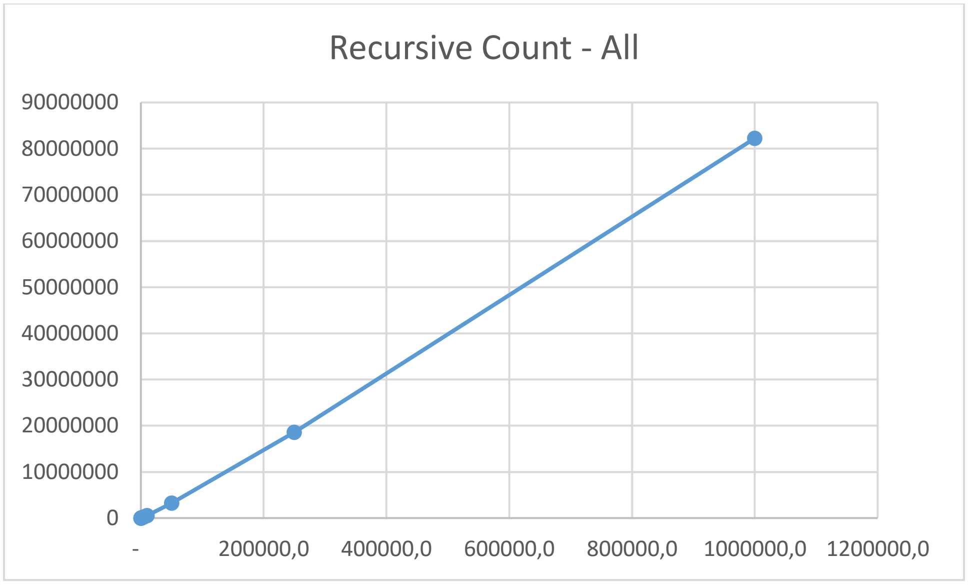
<u>Graph of critical operations for both algorithms and one for execution times</u>

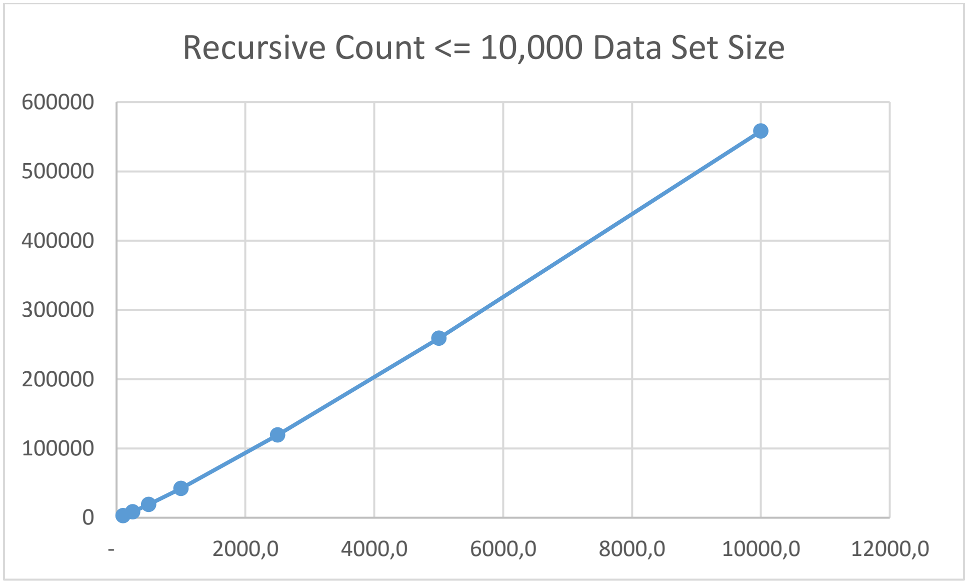
The first two images below display the critical operations count for the Iterative algorithm, the first displays all data set sizes up to 1,000,000 while the second only shows data set sizes up to 10,000 in order to show the continued linear growth of the count.



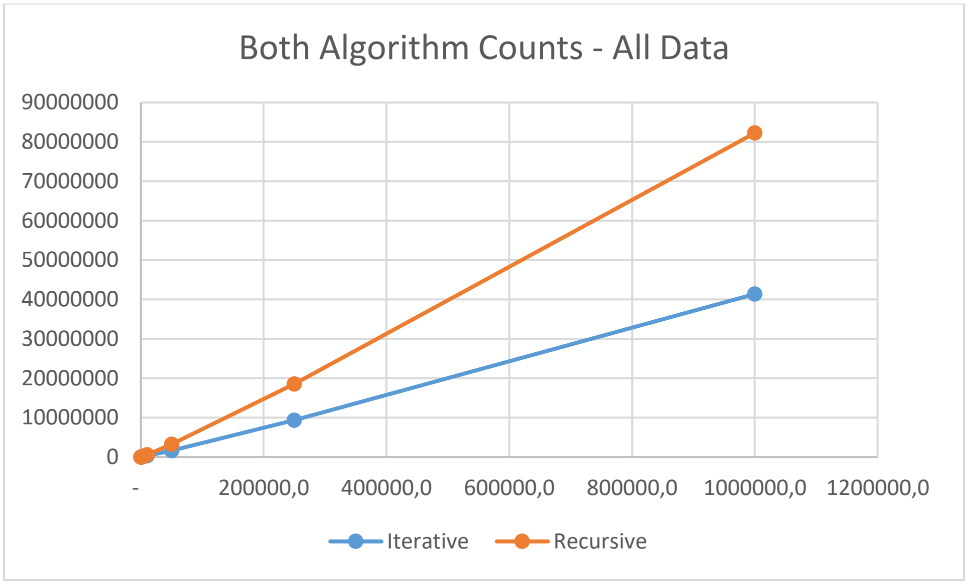


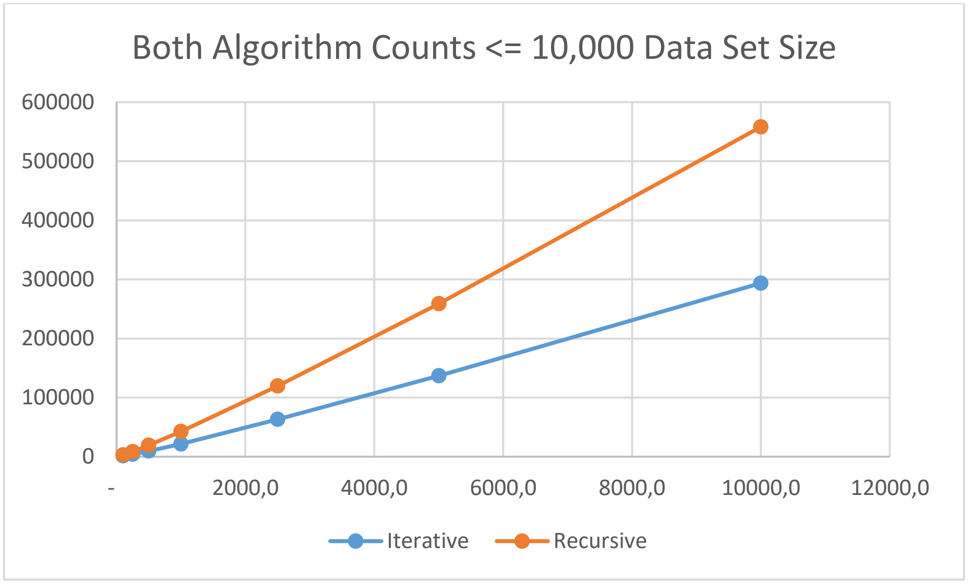
Following are the Recursive algorithm graphs, the same principles apply.



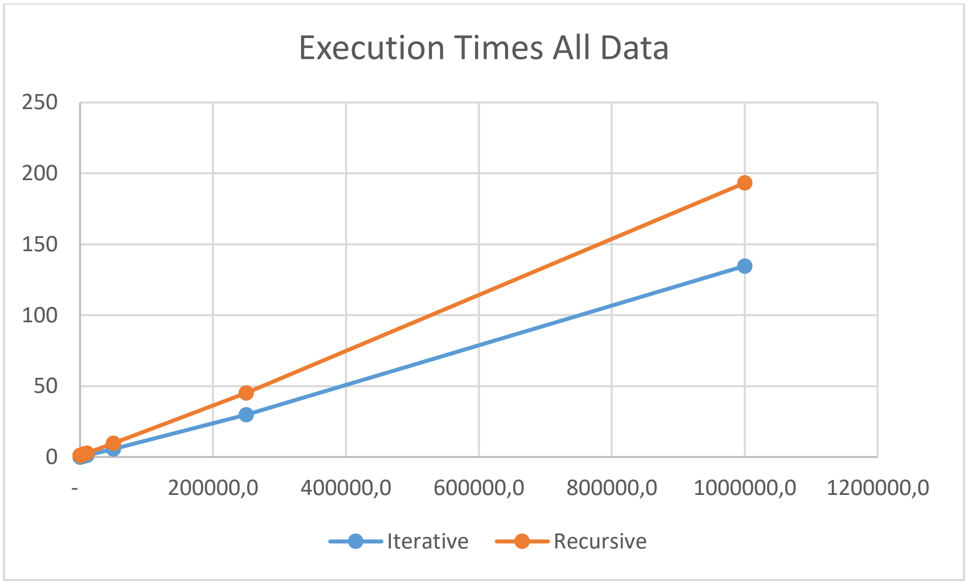


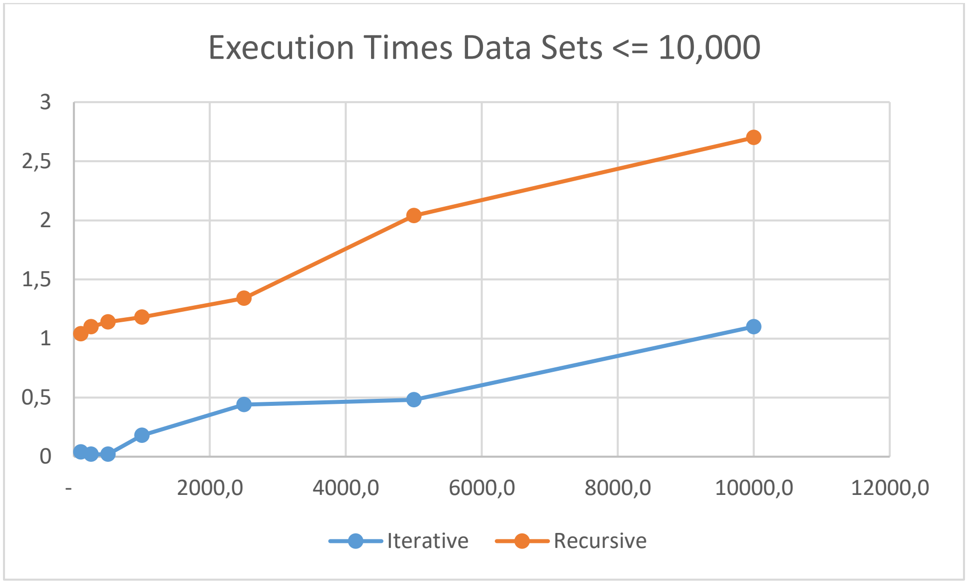
Following contains both algorithms counts, same principles apply.





Following is the graph of execution times for both algorithms, same principles apply.

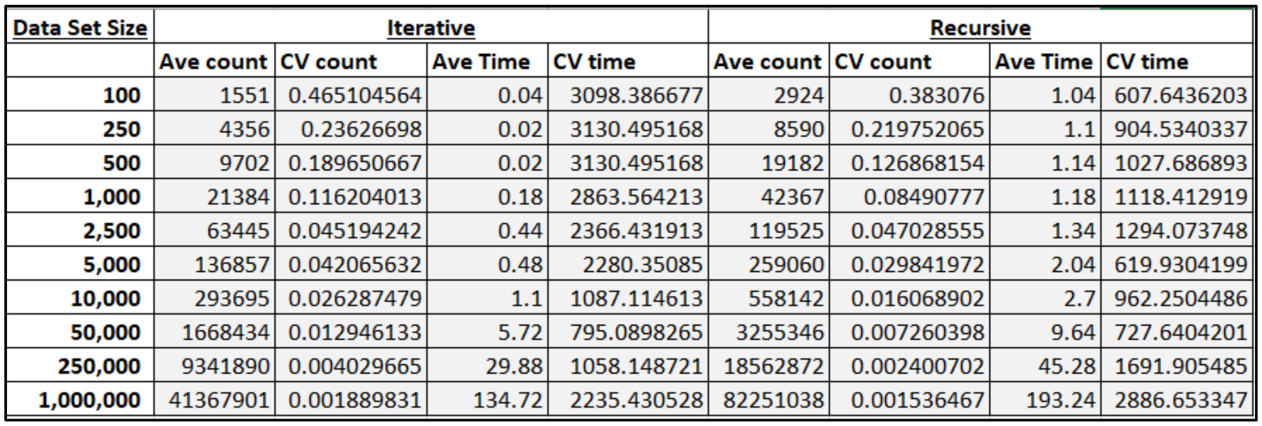




Comparison of performance of the two versions:

Overall, it is very clear that the iterative algorithm performed faster, with significantly fewer critical operations counted for each data set. This assessment is further compounded throughout the next sections. The most significant point of interest is that while it is clear that the Iterative performed better than the Recursive, by a linear constant. That is, the recursive count was typically 88% to 100% greater than the iterative count. The output used for this assessment is as follows from the following .txt output file.

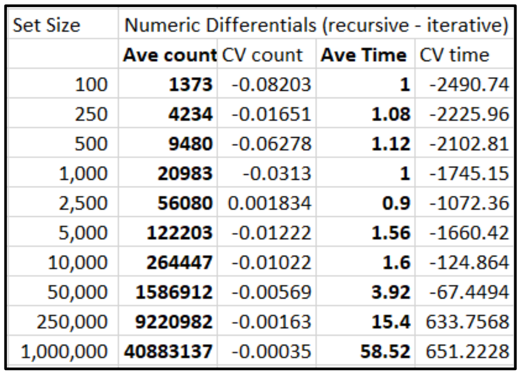
output.txt



Data Set Size	Iterative				Recursive			
	Ave count	CV count	Ave Time	CV time	Ave count	CV count	Ave Time	CV time
100	1551	0.465104564	0.04	3098.386677	2924	0.383076	1.04	607.6436203
250	4356	0.23626698	0.02	3130.495168	8590	0.219752065	1.1	904.5340337
500	9702	0.189650667	0.02	3130.495168	19182	0.126868154	1.14	1027.686893
1,000	21384	0.116204013	0.18	2863.564213	42367	0.08490777	1.18	1118.412919
2,500	63445	0.045194242	0.44	2366.431913	119525	0.047028555	1.34	1294.073748
5,000	136857	0.042065632	0.48	2280.35085	259060	0.029841972	2.04	619.9304199
10,000	293695	0.026287479	1.1	1087.114613	558142	0.016068902	2.7	962.2504486
50,000	1668434	0.012946133	5.72	795.0898265	3255346	0.007260398	9.64	727.6404201
250,000	9341890	0.004029665	29.88	1058.148721	18562872	0.002400702	45.28	1691.905485
1,000,000	41367901	0.001889831	134.72	2235.430528	82251038	0.001536467	193.24	2886.653347

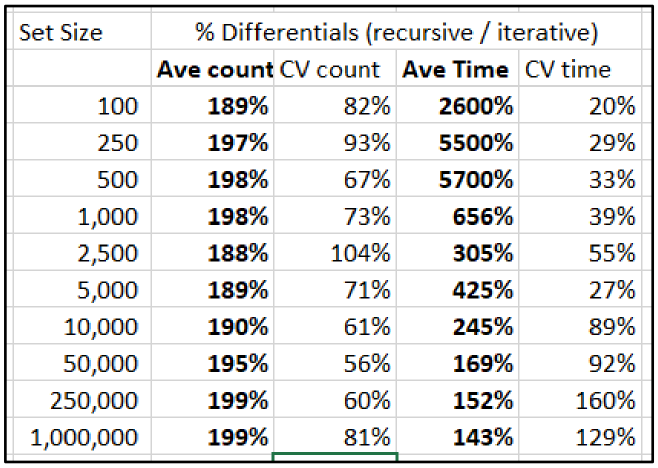
<u>Comparison of the critical operation results and the actual execution time measurements</u>

Using the graphs above, we can clearly see that there were more critical operation counts per data set by a constant number. This is likely to indicate that the big-Θ assessment earlier was nearly accurate, as we had the recursive function growing at approximately (78/48 = 162.5% - 1 =>) 62.5% faster growth. This constant growth ratio appears to be present in the graphs at almost all points given by the linear similarities between the two counts. This is less obvious; however, in execution times. At Data Set Size = 2,500, we see an unusually small difference in time to execute compared to the other values, this difference is representative of the smallest difference in execution time of all. (See column 4, row 7 of the next image for the mathematical proof of this assessment):



Set Size	Numeric Differentials (recursive - iterative)			
	Ave count	CV count	**Ave Time**	CV time
100	**1373**	-0.08203	**1**	-2490.74
250	**4234**	-0.01651	**1.08**	-2225.96
500	**9480**	-0.06278	**1.12**	-2102.81
1,000	**20983**	-0.0313	**1**	-1745.15
2,500	**56080**	0.001834	**0.9**	-1072.36
5,000	**122203**	-0.01222	**1.56**	-1660.42
10,000	**264447**	-0.01022	**1.6**	-124.864
50,000	**1586912**	-0.00569	**3.92**	-67.4494
250,000	**9220982**	-0.00163	**15.4**	633.7568
1,000,000	**40883137**	-0.00035	**58.52**	651.2228

One thing of particular interest is the grade by which the average time increased in comparison to the average count. Over the course of all the data sets, our average count varied by between 189% and 199% between recursive and iterative (i.e. at Data Set Size 500, iterative count was 9,702 and recursive count was 19,182. This indicates that the Recursive count value was 198% of the iterative count value). However, Time to execute did not possess that same consistent trend. For Average time, we have a sudden spike from Data Set Size (DSS) 100 to DSS 250, but then after DSS 500 the ratio difference between times gradually becomes less and less until at the end, with DSS 1,000,000 the Iterative algorithm was only 43% faster than the Recursive algorithm! That's

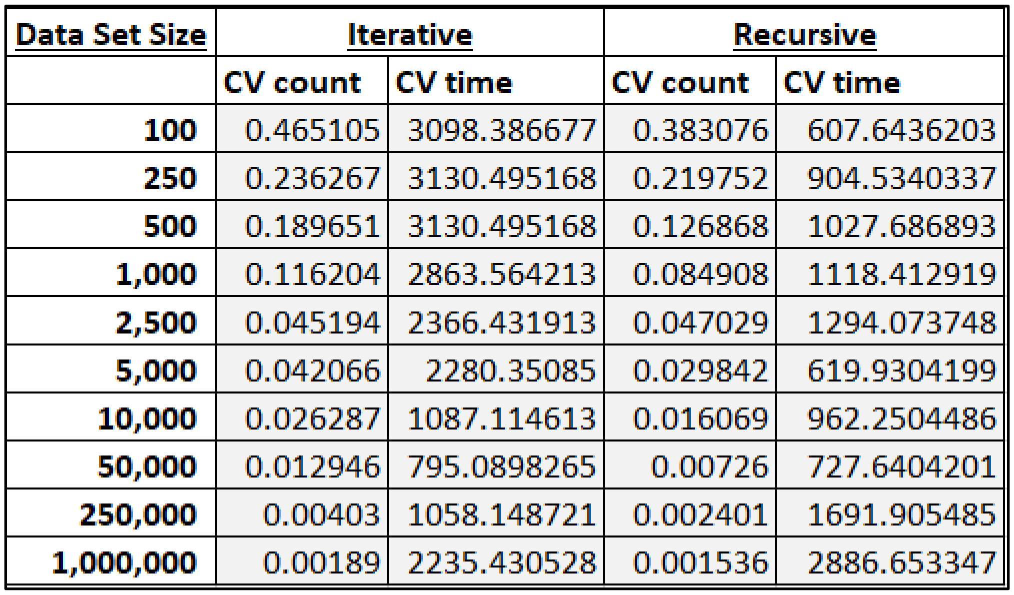


Set Size	% Differentials (recursive / iterative)			
	Ave count	CV count	**Ave Time**	CV time
100	**189%**	82%	**2600%**	20%
250	**197%**	93%	**5500%**	29%
500	**198%**	67%	**5700%**	33%
1,000	**198%**	73%	**656%**	39%
2,500	**188%**	104%	**305%**	55%
5,000	**189%**	71%	**425%**	27%
10,000	**190%**	61%	**245%**	89%
50,000	**195%**	56%	**169%**	92%
250,000	**199%**	60%	**152%**	160%
1,000,000	**199%**	81%	**143%**	129%

nearly 40 times less than DSS 500! From this information, and the gradual descent of values in the average time to execute column, we can assess that it is likely that the larger and larger the data set size becomes, the closer and closer the execution times between algorithms will get until they are eventually equal!

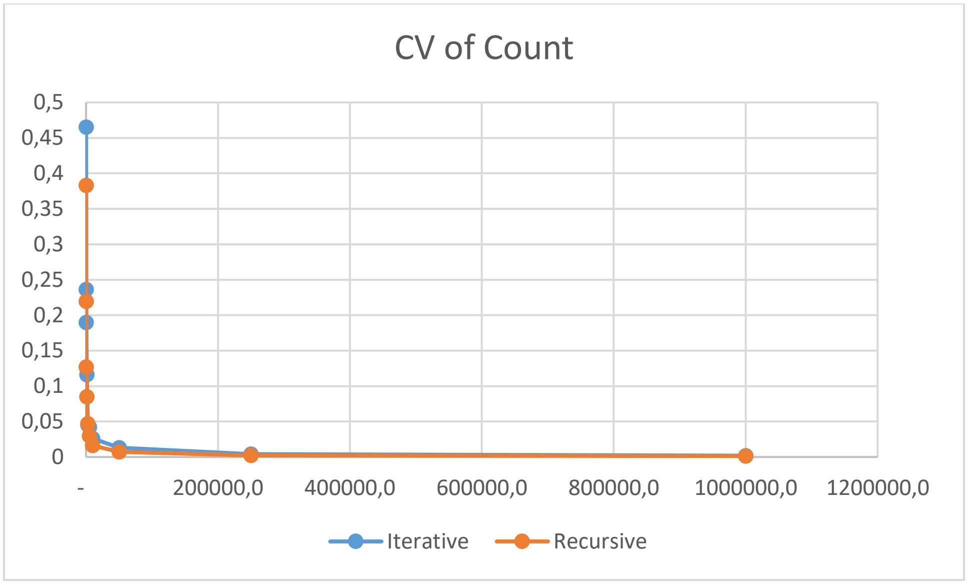
Discussion of the significance of the coefficient of variance results and how it reflects data sensitivity

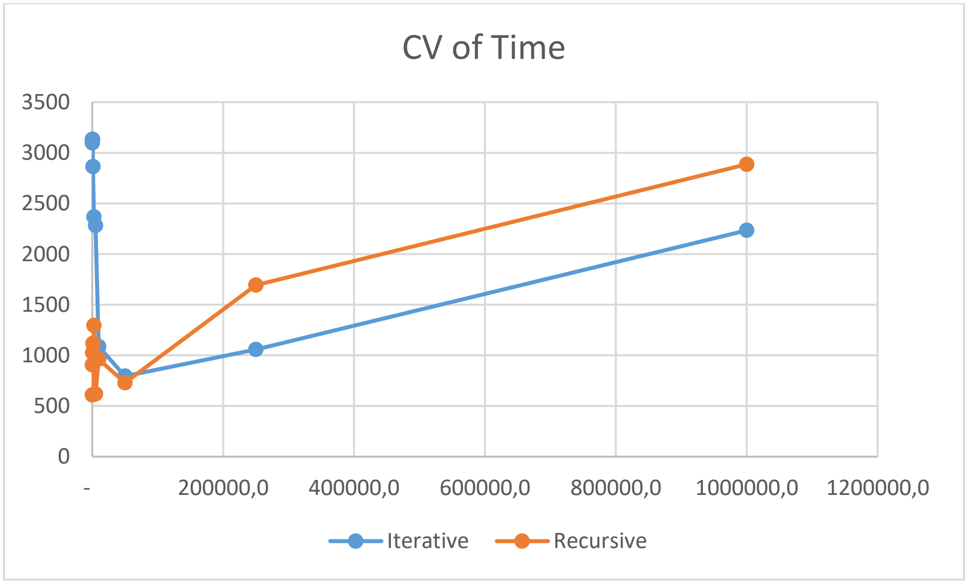
The coefficient of variance (CV) tells us the ratio of comparison between the standard deviation and the average result. Essentially, standard deviation is the degree to which one set of data differed from another, and the average tells us the value of the total sum of all values divide by the number of elements, giving us a generalized idea of where the data most commonly occurred. We use coefficient of variance because it tells us just how inconsistent our data is, which is thereby a direct reflection of data sensitivity. By using the above output data, we can see that the CV for execution time is incredibly smaller than the CV for count. (For easier viewing I've placed a revised table blow which only shows CV values.)

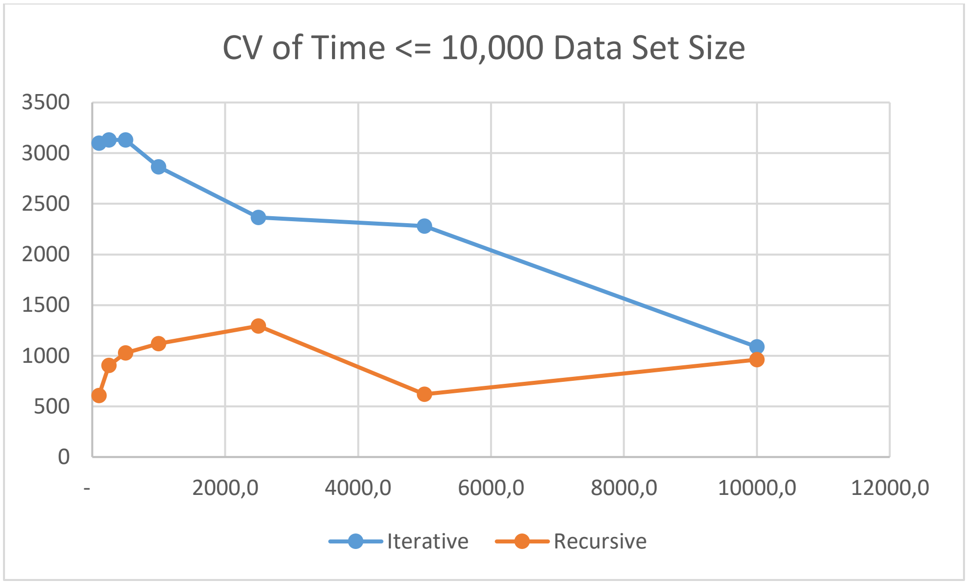


Data Set Size	Iterative		Recursive	
	CV count	CV time	CV count	CV time
100	0.465105	3098.386677	0.383076	607.6436203
250	0.236267	3130.495168	0.219752	904.5340337
500	0.189651	3130.495168	0.126868	1027.686893
1,000	0.116204	2863.564213	0.084908	1118.412919
2,500	0.045194	2366.431913	0.047029	1294.073748
5,000	0.042066	2280.35085	0.029842	619.9304199
10,000	0.026287	1087.114613	0.016069	962.2504486
50,000	0.012946	795.0898265	0.00726	727.6404201
250,000	0.00403	1058.148721	0.002401	1691.905485
1,000,000	0.00189	2235.430528	0.001536	2886.653347

As the data sets grow larger, we can see that the CV of Count for both iterative and recursive grow smaller and smaller, this indicates that the data is less sensitive in larger data sets. On the contrary, we can see that the CV of Time has a less obvious coefficient of growth. See Below:


CV of Count
0,5
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
-
200000,0
400000,0
600000,0
800000,0
1000000,0
1200000,0
Iterative
Recursive


CV of Time
3500
3000
2500
2000
1500
1000
500
0
-
200000,0
400000,0
600000,0
800000,0
1000000,0
1200000,0
Iterative
Recursive



From these charts we can also see that the CV of time is most closely, as in there is the most common amount of variance between the recursive and iterative algorithms when the data set size was 10,000.

How your results compare to your Big-Θ analysis

Are the actual results synonymous, or similar to the expected performance?

Big-Θ depicts the performance complexity of the execution time required by an algorithm. Therefor, we will look at the execution times of the algorithms based on the size of the data sets to determine of the performance complexity matches our earlier assessments. We assessed earlier that the both the iterative and recursive algorithms would perform at Θ(n*log(n)) while the iterative has a coefficient of 48 and the recursive has coefficient 78.

Where n is the number of elements, we can insert that number to determine the actual values of data set size * log(data set size) as well as coefficient * data set size * log(data set size). [Contrary to output data text document, the values in the table for average execution times is displayed in μs] I've done this in excel as follows (values adjusted to account for unit conversion).

Data Set Size	Iterative				Recursive			
	Ave Time	CV time	n*log(n)	48*n*log(n)	Ave Time	CV time	n*log(n)	78*n*log(n)
100	0.04	3098.386677	200	0.010	1.04	607.6436203	200	0.016
250	0.02	3130.495168	599.485	0.029	1.1	904.5340337	599.485	0.047
500	0.02	3130.495168	1349.485	0.065	1.14	1027.686893	1349.485	0.105
1,000	0.18	2863.564213	3000	0.144	1.18	1118.412919	3000	0.234
2,500	0.44	2366.431913	8494.85	0.408	1.34	1294.073748	8494.85	0.663
5,000	0.48	2280.35085	18494.85	0.888	2.04	619.9304199	18494.85	1.443
10,000	1.1	1087.114613	40000	1.920	2.7	962.2504486	40000	3.120
50,000	5.72	795.0898265	234948.5	11.278	9.64	727.6404201	234948.5	18.326
250,000	29.88	1058.148721	1349485	64.775	45.28	1691.905485	1349485	105.260
1,000,000	134.72	2235.430528	6000000	288.000	193.24	2886.653347	6000000	468.000

As can be seen by the data above, ((n*log(n) values not adjusted for units), for our iterative assessment, our actual values and our expected values are fairly similar. In some instances, the values were nearly identical (see data set size 1,000) but in others the actual output was less than half of the expected output (data set sizes 250,000 and 1,000,000). In other instances, the actual output was nearly identical to the expected output (see data set size 1,000 and 2,500).

Contrarily, the actual recursive values were nowhere near the actual outputs. At times the expected time was nearly 150% more than the actual time, and the closest we got to an actual assessment was data set size 500 where the difference was only 0.9. Due to the nature of the differences in both algorithms, it is likely that there was an error when computing the Big-Θ assessment, and that the logs should have been impacted more and the coefficient less.

Alternatively, given the Coefficient of Variance of Time, it is also possible that there were instances wherein the actual time was equivalent to the expected time given the small degree of variation between values. The CV of time demonstrates that there is between a 795% and 2,886% variance. Given the degree of data sensitivity, it is difficult to determine the true degree of accuracy, but we can assess that it is possible that the original Big-Θ assessment was not completely accurate.

Conclusion:

One of the primary things that we were able to see from this project was that, while the recursive method had cleaner code and was arguably easier to read, the iterative method performed more effectively both in critical operation count and in overall execution time. Additionally, we were able to assess that the best-case and worst-case scenarios based on the output data from the program showed that there was still a linear performance which indicates that the overall performance was consistent (there were no major deviations) throughout all data sets of both methods.

We were also able to see that the time to execute was the primary varying factor. In some instances, we saw CV of time as high as 2886%! This value was indicative that our JVM warmup was reasonably effective in terms of the time to execute. However, a more intensive

JVM warmup may have reduced the CV of count for both methods overall and may have yielded a more consistent, smaller CV of count.

Overall, we can assess that the Merge-sort method is a very consistent sort method. As data set size increased, count and time increased proportionately with very few outliers. This gives us a greater understanding for why the Java framework utilizes merge-sort for non-primitive data types as the more complex values are effectively sorted with very little data loss over very large quantities of data.

<u>Data Excel Sheet:</u>

sortedData.xlsx

<u>Merge Program:</u>

MergeBenchmark.zip