# Introduction – Merge Sort

For this assignment, I opted to benchmark merge sorting. Merge sorting happens when a list of data is split into smaller lists that are broken down to their smallest components, typically a sub list of one element. In here, there would be three types of lists: the list we are working on (active list), the list above it (parent list), and any lists that spawn from it (children lists). Typically, at most two children list belong to a single active list, and the children list compare their elements together to see which of their elements need to be sorted. The act of combining the children lists together is called merging.

Merge sorting can be done in an iterative manner as well as a recursive manner, and as such, both have their own efficiencies. What is shared between them is that, assuming the same list is used for both iterative and recursive approach, the list will be at most N number of elements long, so in the case of any looping, N number of elements will have to be looked at.

When it comes to time, both algorithms should be the same efficiency: O(N\*logN). The reason for this is because in either case, the list being sorted is being split in half, with each sub list is also being split in half. It should take about the same amount of time to handle this operation.

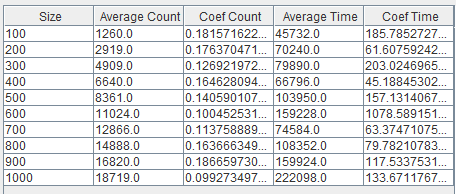
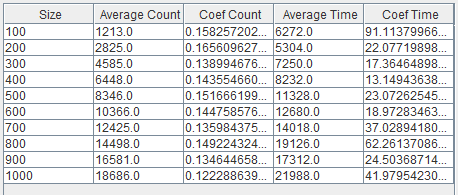
 For this assignment, I had to benchmark both iterative and recursive merge sorting algorithms to determine some statistics for each sort. I track the amount of operations each algorithm takes as well as tracking the average time each algorithm is expected to take in nanoseconds. I generated a report sheet that describes the results of each sorting algorithm. The data is displayed below:

Figure - Recursive Merge Sort Statistics

Figure - Iterative Merge Sort Statistics

# Pseudo Code – Iterative Approach

1. Calculate the middle of the array
2. Loop through the array up until the middle of the array and put the data inside of an array (leftArray)
3. Loop through the ray from the middle of the array to the end and put the data inside of an array (rightArray)
4. Compare the elements of leftArray and rightArray and add the lesser element to a new array
5. End

# Pesudo Code – Recursive Approach

1. Get the index of the middle of the array
2. Loop through the array and get all the entries from the left side of the array (up until the middle index) and set them to an array (leftArray)
3. Loop through the array and get all the entries from the right side of the array (from the middle of the array to the end) and set them to an array (rightArray)
4. Sort leftArray with recursion
5. Sort rightArray with recursion
6. Loop while two custom counters are less than the size of leftArray and rightArray
   1. If the entry at the current index of leftArray is less than or equal to the entry at the current index of rightArray
      1. Set the next element in leftArray to the next element of the main array
   2. If the entry at the current index of leftArray is greater than the entry at the current index of rightArray
      1. Set the next element in rightArray to the next element of the main array
7. While a counter is less than the size of leftArray
   1. Add that entry to the main array
8. While a counter is less than the size of rightArray
   1. Add that entry to the main array
9. End

# Critical Operations, Time, and Analysis

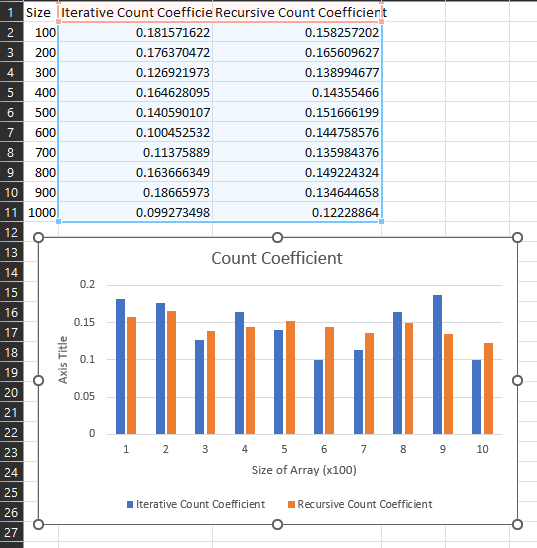
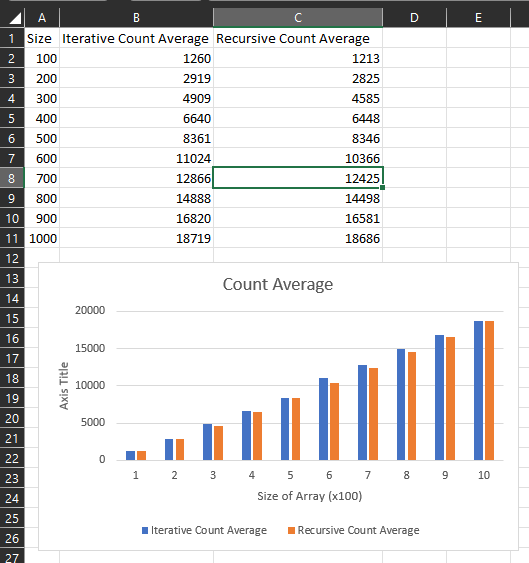
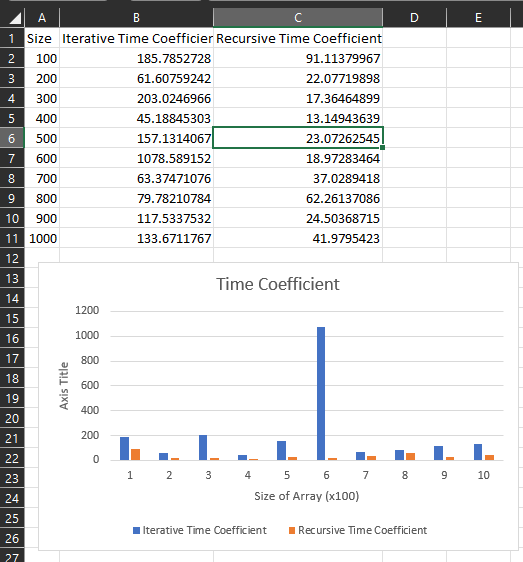
When thinking about any sort of critical operation for the two merge sort algorithms, I felt as though that it would apply to the instances in which any merging is done. What I figured is that the best way to determine this is to look at the endings of each sort where the merging happens. This is where the comparison of elements are taking place. I count each comparison as their own operation. 

Figure - Count Coefficient Comparison

Figure - Count Average Comparison



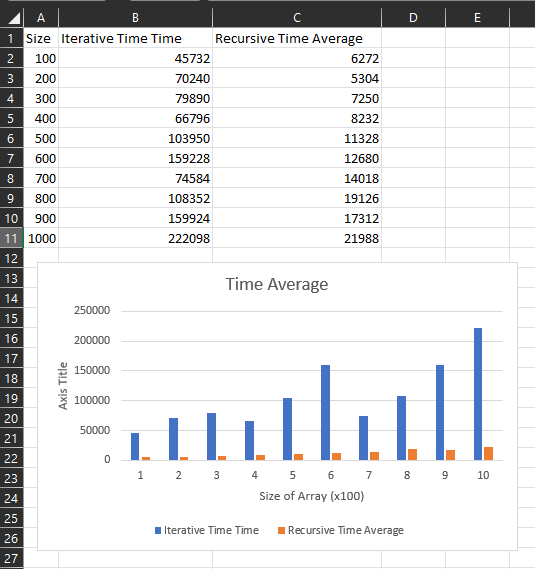


Figure - Time Coefficient Comparison

Figure - Time Average Comparison

According to the above data, recursive merge sorting performs slightly less operations on average compared to iterative merge sorting, even though they are very close together. However, it also appears that recursive merge sorting takes a significantly less amount of time to complete compared to iterative. Due to this, I want to believe that recursive sorting is a lot more efficient than iterative sorting.

There also might be some level of scrutiny in this report though. I also analyzed the coefficient for both the count and time. The lower a coefficient is to zero, the more accurate calculation will appear to be. While the coefficient for counting seems to be low, suggesting that the calculation for counting the operations is close to accurate, the coefficient for time seems to be all over the place, suggesting some level of inaccuracy. I have tried to work on fixing the time conversion by implementing a warm up function to the MergeSort class, but nothing seems to get the time coefficient to be consistent.

# Conclusion

It is important to understand the different benefits and trade offs between different implementations of an algorithm to know when to best use them. I find it interesting that a recursive merge sort appears to be more efficient to use than an iterative merge sort, but I would also suspect that there is some conditions as to what makes a recursive merge sort faster. I would have expected the time averages to be close to each other to support the idea that their Big-O analysis was correct, but further testing and diagnosing on my code should allow me to see that they are both withing some variance of each other.