**Introduction**

There are many different types of sorting algorithms that have their pros and cons depending on what situations you are using them for. Sorting algorithms, as the name implies, are meant to take an input of elements and sort them in a certain order. For large sets of data the most commonly used sorts today are heap sort, merge sort, and quicksort. For the purposes of this paper we will be focusing on quicksort. Quicksort is a divide and conquer algorithm. Divide and conquer algorithms break down a problem into two or more smaller problems usually through recursion until the problem is solved. Quicksort was developed by Tony Hoare and has become widely implemented in libraries for programming languages like the C family and in Java’s Arrays.sort() method which uses a dual-pivot version of quicksort. The way in which quicksort works is an element in the array is picked to be a pivot point. The array is reordered so that the elements which have higher values than the pivot are placed after the pivot and elements with lower values are placed before the pivot. The same process is then repeated for both sub-sets of the array on either side of the pivot until the array is in order. There are many different versions of quicksort that choose the pivot in different ways, such as picking the first element, the last element, a random element or the median as the pivot. In project 1 the method chosen was picking the last element as the pivot.

**Iterative vs. Recursive Discussion**

The quicksort algorithm can be implemented through iteration or recursively. Even though both perform the same tasks the way it is handled differs. The iterative method loops through a counter repeating the task until the problem is solved; the recursive method calls itself and breaks the problem into smaller pieces then combines the results as mentioned above. One main positive feature of the recursive implementation of quicksort is that the code is much simpler to write and easier to follow. This makes it less likely for the programmer to make mistakes. All recursive algorithms can be expressed in an iterative form. In the case of an iterative implementation of quicksort, you would use an explicit stack data structure instead of the recursive stack. Although in certain implementations one may be quicker than another, if the iterative version is written correctly the asymptotic time complexity is the same for both as will be shown below in the benchmarking section.

To discuss Big O of both the recursive and iterative implementations of quicksort we need to look at the best case, worst case, and average case. As mentioned above, if implemented correctly the time complexity should be the same for iterative and recursive implementations. The best case would be if the middle element was picked every time so that each time the array was split both sides would be equal, making the depth of the tree log n. The call at each level of the tree would be O(n) making the total time complexity O(n log n) for best case. The worst case for recursion would be if the pivot element was either the least or the greatest element in the array (an already sorted array). This would cause all elements to either be moved to the right or the left of the pivot accordingly making each comparison n – 1 in the worst case. This would give a time complexity of O(n^2) in the worst case. Although, using different implementations like choosing a random pivot point can effectively eliminate the worst case scenario. For the average case, if you use the master theorem you find that the time complexity would be O(n log n).

**Project 1 Benchmark Analysis**

Benchmarking was accomplished in Project 1 which involved inputting 10 data sets of a specific size, producing 50 data sets of said size. To calculate efficiency a critical operation was chosen to count the number of executions and also time measurements were calculated for the 50 runs of each data set size. I decided to set the critical counting operation for each time both functions set the pivot element at it’s proper position, in this case the last element of the array. This was done because the partition() method was the same for both the iterative and recursive implementations and would be a fair way to judge efficiency each time the method was called.

To begin our analysis we will look at the benchmarking results as shown in the diagram below.

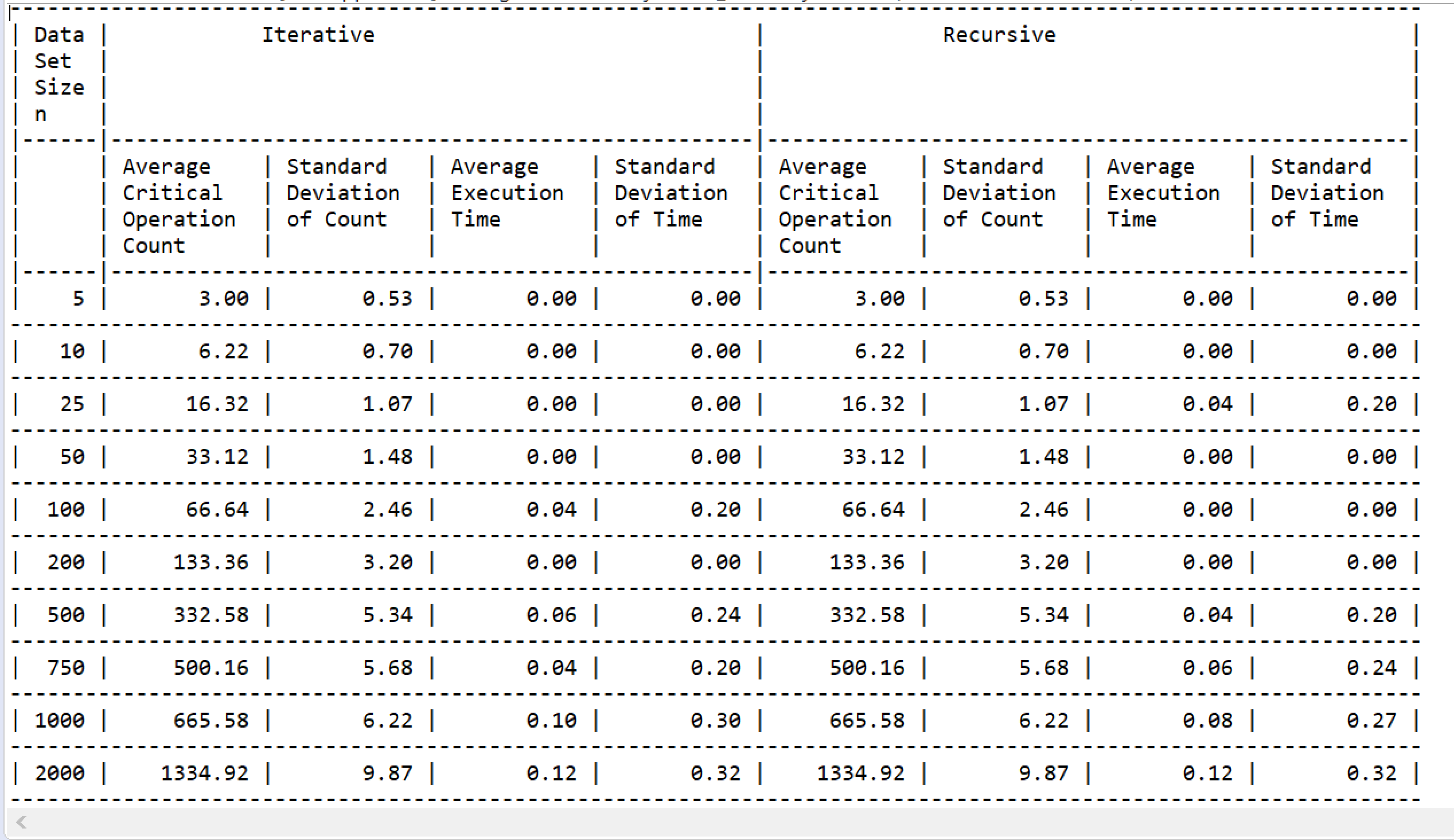
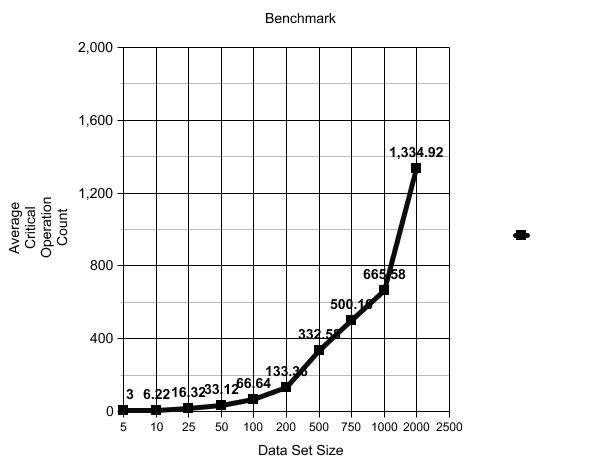
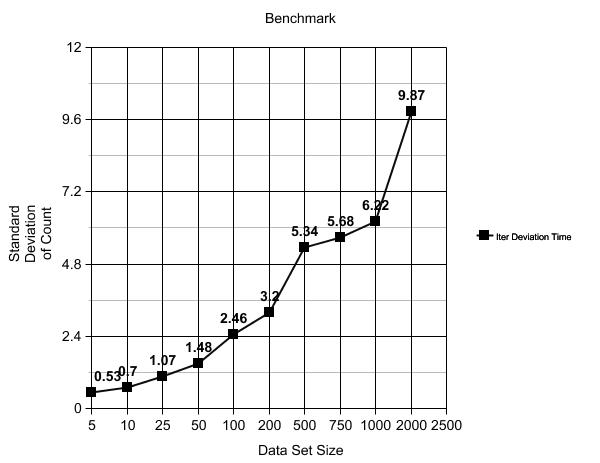
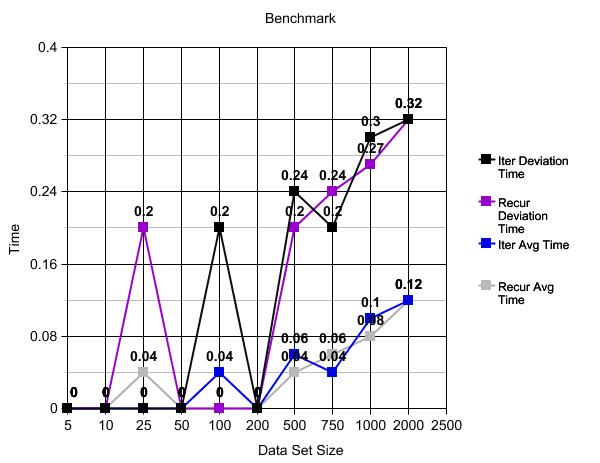


Figure 1 – Benchmarking Stats

As you can see in the data set in figure 1, both the iterative and recursive implementations are roughly identical. The critical operation that was used as the counter was exactly the same, as expected. This is because as mentioned earlier, the functionality of both implementations is the same, the method in how you sort is just slightly different. Because the average critical operation count is the same this explains why the standard deviation of count calculations are identical as well. In both cases, the array size is the same and the space complexity would also be the same. When comparing the critical operation results to the measurements of execution time it is fair to say that both iterative and recursive implementations are equally as efficient. The variation in average execution time has more to do with the fact that the last element was chosen as the pivot and data is randomized for each data set versus one implementation being more efficient than the other. The variation in execution time and standard deviation of time between iterative and recursive implementations also seem to decrease as the size of the data sets increase. Standard deviation of time also plays an important role for both versions since it shows accuracy of results and reflect data sensitivity. As the size of the data set increases so does the standard of deviation for time. The standard deviation of time was most noticeable when the data set size was greater than 200 for both iterative and recursive implementations. Below are a set of graphs shows the comparison of the data between iterative and recursive implementations.







**Conclusion**

Based on my research and the analysis of the results it is fairly obvious to me that if the logic of the recursive implementation is implemented iteratively with an explicit stack data structure that both versions are equally as efficient. The average critical operation count and its standard deviation are the same for both versions of the quicksort algorithm. The average execution time also stays relatively the same as the size of the data set increases. Because I chose to use the last element as the pivot the algorithm was not as refined as it could be. Randomizing the pivot could decrease the chance of a worst case situation happening and decrease average run time. Because the average case of the quicksort algorithm is O(n log n) it is a very useful and efficient solution for sorting in many different cases.

**References**

Analysis of quicksort (article) | Quick sort. (n.d.). Retrieved October 15, 2017, from <https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort>

Barnwal, A. (2016, December 20). Iterative Quick Sort. Retrieved October 15, 2017, from http://www.geeksforgeeks.org/iterative-quick-sort/