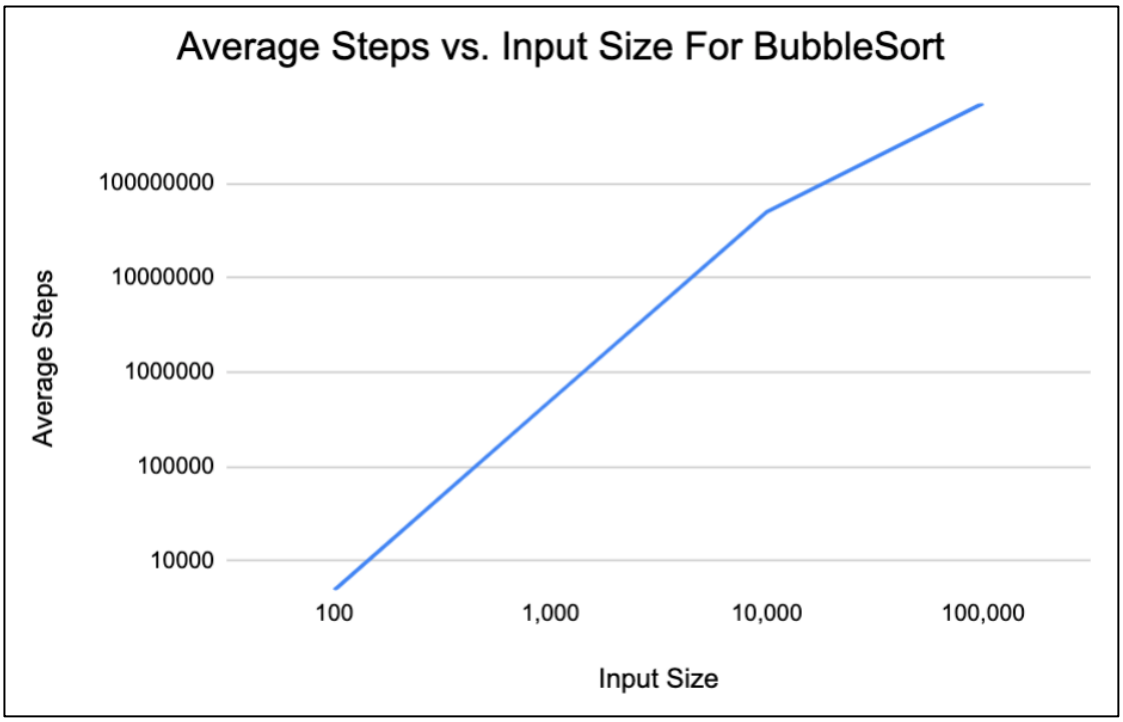
Q5 PDF Report

*****Y-AXIS ON GRAPH IS LOG SCALED!! (for the excel screenshot with data, please look at Q2 Folder)*

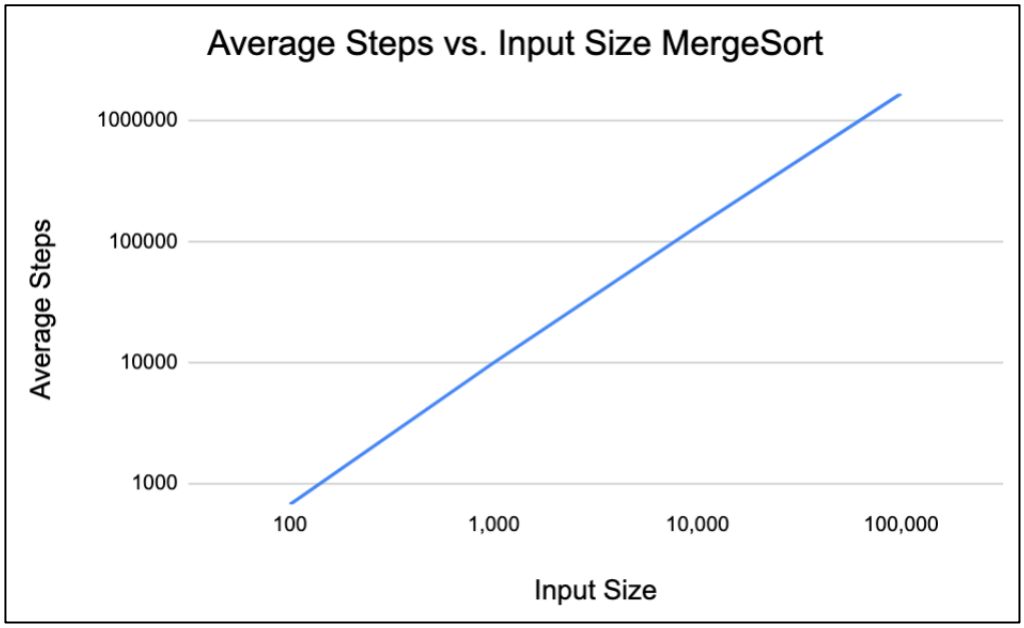
The three algorithms which were explored were BubbleSort, MergeSort, and QuickSort. Each algorithm takes an array of Comparable type and in my code, I sorted an array of Integers. The purpose of each algorithm is to sort the array in ascending order. Each program also has a step counter to count the number of times the program compares two items in the array. Each program was tested with different input sizes of arrays and run 3 times to get an average of the number of steps. This data was plotted and used to compare the algorithms to each other in terms of efficiency.

The BubbleSort algorithm sorts by taking an unordered array and comparing each item in the array to the item next to it and swapping them if they are in the wrong order. I placed the step counter right after most inner for loop as that is where the comparison is being made and that is the most time-consuming part of the method. The following is the plot generated from the data collected:

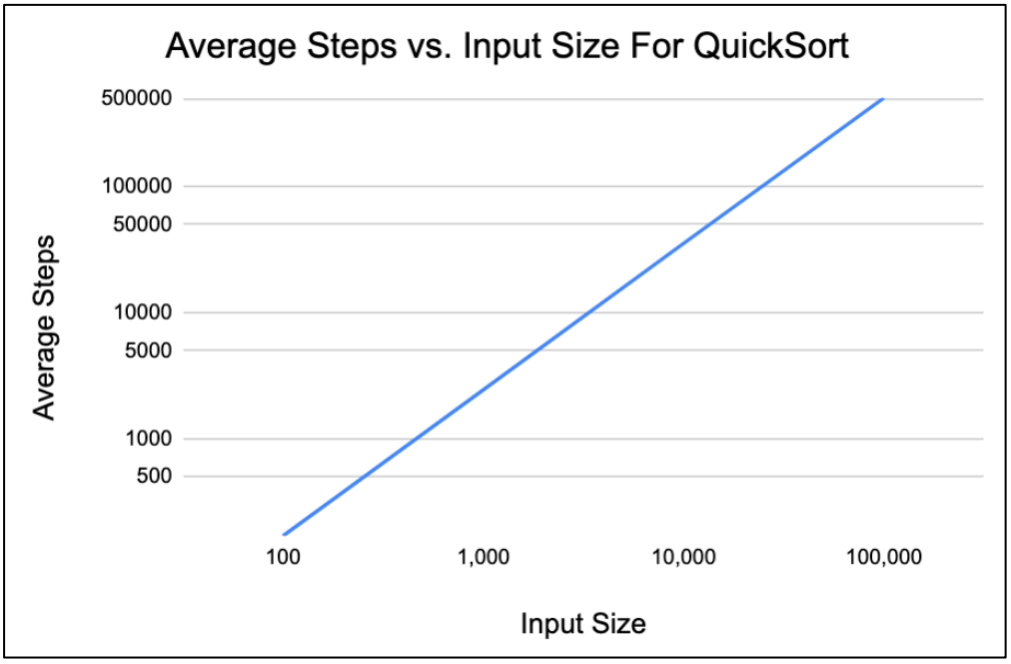


The MergeSort algorithm sorts by dividing and conquering. The array is split up in half, then each of those halves are split, and this keeps going until each element is singled out; hence, the divide portion of “divide and conquer”. Then, each item is compared to the item next to it and any switches which need to be made are done; hence, the conquer portion. Because the method in this algorithm is the divide each item in the array and then compare each item to the next item, the number of steps is the same each time for an input size regardless of how many times the program is run. I placed the step counter in the merge() method right after the second for loop where all the compares are occurring

because that is the most complicated part of the algorithm which takes the most time. The following is the plot generated from the data collected:



The QuickSort algorithm is similar to MergeSort as it also sorts by dividing and conquering. The algorithm chooses an item as a pivot and splits or partitions the array around the pivot item. The partition() method is the key of this algorithm as its purpose is to put the given pivot item in the correct position in the array and put all items smaller than the given partition and put all the items larger than the given partition after the given partition. I placed the step counter right after the while loop which is part of the partition method as that is the most complicated part of the algorithm which takes up the most time. The following is the plot generated from the data collected:



After collecting, plotting, and analyzing the data, the conclusions are very clear. The average number of steps was always significantly higher for BubbleSort than the average number of steps for MergeSort and BubbleSort. This is because BubbleSort has to go through the entire array and compare each time without any splitting of any times. BubbleSort compares items in pairs without overlapping, causing it to start at the beginning of the array to do the same thing until the array is fully sorted. Comparing the average number of steps of MergeSort and QuickSort, QuickSort was the clear winner. Even though both algorithms have O(nlogn) runtime, QuickSort's worst-case runtime is $O(n^2)$ and average case runtime is O(nlogn). QuickSort also requires little additional space where as MergeSort uses extra space proportional to n. It is also easy to avoid the worst-case scenario for QuickSort based on the pivot chosen. QuickSort also is recursive and randomizes because it randomly shuffles the array before sorting it. All of these factors make QuickSort the most efficient algorithm over MergeSort and BubbleSort.