CS 150 Project 1 Report

Tafita Rakotozandry & Khalid Almotaery

10/11/2020

1 Introduction

As the importance of datasets increases with the rapid human developments, understanding how to organize and analyze data has never been more important. Sorting information efficiently is one of the most foundational ideas in Computer and Data Science. The aim of this project was to have a clear realization of how sorting algorithms behave. This report will discuss different sorting algorithms and their performance with different datasets. Memory space and time are the usual variables used to measure the performance of an algorithm. However, this work will use time as the sole indicator of the best-fit algorithm. Because there are other factors that affect the running time of an algorithm such as the operating system and the speed of the device which is used to execute the algorithm in question. Moreover, the time of the algorithms will be viewed in it's "big picture." This point of view allows for ignoring all exterior factors by only considering the growth of time as more data is inserted. Thus, the time of each algorithm will be represented in it's "big-oh notation".

The sorting algorithms that were examined in this project were Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort (with random, first, and medium pivots). Because of the different logic behind each algorithm, each one has its own advantages and disadvantages as well as its best case, worst case, and average case. Best cases are datasets that are organized in a way that makes a given algorithm perform at its optimal performance.

Similarly, average cases and worst cases are data sets that make the algorithm in question perform at an average and worst performance respectively. The following is a detailed description of each algorithm:

1.1 Bubble Sort

Bubble sort is an algorithm that sorts a given data set by comparing each adjacent element to the other. It does so repeatedly until the large elements of the dataset "bubble" to the end of the data set. Thus, it sorts the data set from the end to the beginning. Although the method is simple and easy to implement, it is one of the worst algorithms in terms of time efficiency and has a big oh notation of o(n^2) for worst-case and average-case and o(n) as its best case. The reason for which has different time efficiently for the two cases is the logic of the algorithm. When the array is sorted, the algorithm is smart enough to go through all the elements once and realize that the data set is already sorted. This realization is made possible by a counter variable that checks if any swaps were made. This movement costs o(n), which makes it the best case for Bubble Sort. On the other hand, when the array is sorted reversely, the algorithm must go through the array many times and swap elements costing a worst-case timing of o(n^2). An example of an average case is if the first half of the array is sorted and the second half is reversely sorted. This case would still result in a big oh notation of o(n^2). Even though the average case should relatively take less time than the worst case, both of the cases have the same big-oh notation because they differ by a constant which can be ignored when looking at the "big picture." If the programmer knows that the given data set is sorted, Bubble Sort can efficiently prove that. However, in all other cases, this method is time costly.

1.2 Selection Sort

Selection Sort orders a given data set by finding the smallest element of the array and swapping it with the first unsorted element. This makes this method work oppositely in some ways to Bubble Sort as it sorts from smallest to largest. The approach costs time of o(n^2) for all data sets. This means that the algorithm does not have a best-case, average cases, nor worst cases. If we assume that a sorted list is the best data set that can be given, it will still have the same performance. That is true because the algorithm is designed to loop through the array assuming that further elements are smaller. An advantage to such a method is that no matter how the data set is arranged the big-oh notation is always going to be the same. The obvious disadvantage is that the algorithm is costly with all arrangements.

1.3 Insertion Sort

Insertion Sort inserts the element in its "right place." With Bubble Sort, the elements are only compared with their adjacent elements and swaps when the elements are out of order. Insertion Sort compares and swaps the element in question with all prior elements until the element in question is in its correct order. This makes the element perform very well with small arrays. Furthermore, Insertion Sort can be chosen over high-performance algorithms like

Quicksort when dealing with small data sets. Even with this fact in mind, the logic of Insertion Sort has a big-oh notation of o(n^2) for worst and average cases. When given a reversed ordered array (worst case) or a half sorted half reversely sorted array (average case) the methods would loop through for a notation of o(n^2). However when the array is already sorted (best case), the method would loop through the array once to know that the array is sorted, which gives it a running time of o(n).

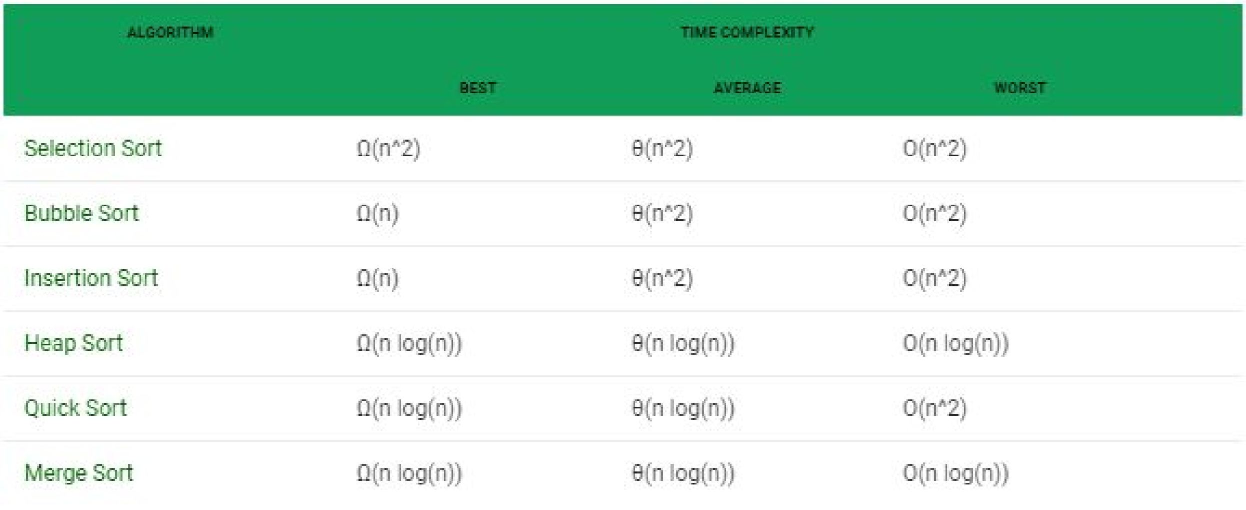
1.4 Merge Sort

The Merge Sort orders a given data set by following two routines: merging and splitting. The method starts by halving the data set repeatedly until each element is its own data set. Then it merges each sub-set together in an ordered way. It does so until all the data sets are merged together in a correct order. This method is considered to be one of the most efficient sorting methods for sorting a data set. Merge Sort has a big-oh notation of o(nlog(n)) for any data set arrangement. The outstanding performance is made possible by the logic in which this function follows. log(n) indicates the number of cuts that are possible for a given data set with size n. This expression is multiplied by n because of the number of elements that need to be merged together. This method is very useful for sorting large arrays at fast speeds. However, the method is not ideal for smaller arrays.

1.5 Quick Sort

Quick Sort is another efficient sorting algorithm that was examined in this project. The algorithm uses a pivot element to partition and break down the array into two sub-arrays. The first sub-array has values less than the value of the pivot and a second sub-array that has values greater than the pivot. The pivot is chosen repeatedly in each new sub-array until all the elements are considered as sub-arrays themselves. When all sub-arrays are merged together, a sorted data set is the result. The pivot element can be chosen anywhere in the array, but in the project, the choices of the pivot element were: the first element in the data set, a random element in the data set, and the median of the first, middle, and last element of the data set. Because of the logic of the algorithm, each pivot results in a different running time. Usually, the first pivot is the worst performance, followed by the median pivots and the best performance is with the random pivot**.** Looking at the big picture of Quick Sort, all pivot values should yield a best case and an average case big-oh notation of (nlo(n)). However, the worst case data set can lead to a running time of o(n^2). The resulting bad performance is due to choosing the pivot as the first or last element in an array that is sorted, reversely sorted, or an array that only contains duplicate values. In all other cases, quick sort is a high performance sorting method. Comparing Merge Sort and Quicksort running times will clearly favor Merge Sort because of Merge Sort's universal notation of o(log(n)). However, Quicksort has an advantage when it comes to memory space. Quicksort only uses O(log(n)) of extra memory, while Merge Sort uses O(n) of extra memory.

In summary, here is table describing the theoretical bigO of each of the mentioned algorithms above:



ALGORITHM	TIME COMPLEXITY		
	BEST	AVERAGE	WORST
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))

2 Approach

An experimental based approach was taken to deeply understand the behavior of the sorting algorithms in question. This meant that it was needed to test the sorting algorithms using different array sizes and characteristics. The team constructed test cases with the following characteristics:

1. Sorted Datasets
2. Reverse Sorted Datasets
3. Datasets with Random Order
4. Datasets with duplicate elements
5. Datasets that are Half Sorted and Half Reversely Sorted elements.

Here are examples of these different types of dataset:

- Sorted Datasets: 1,2,3,4,5,6,7,8,9,10 (increasing)
- Reverse Sorted Datasets : 10,9,8,7,6,5,4,3,2,1 (decreasing)
- Datasets with Random Order: 4,6,2,8,14,36,452 (random)
- Datasets with Duplicate Elements: 1,1,2,2,3,3,4,4 (element duplicated twice)
- HalfSorted Datasets: 1,2,3,4,5,68,423,744,33,789 (the first half is sorted and the rest is random)

When we generated the string, we decided to make a string with a length of 3 letters each. They are sorted the same way as if we were sorting an int.

Each dataset characteristic was split into three size categories: Small, Medium, and Large datasets. The Small datasets are arrays with sizes ranging from 100 elements to 900 elements. The medium data ets are arrays with sizes ranging from 1000 elements to 9000 elements. Finally, the large datasets are arrays with sizes ranging from 10000 elements to 90000 elements. The previous 5 characteristics are chosen because they test most of the theoretical best case, worst case, and average case scenarios. Sorted datasets are considered to be the best case for Bubble Sort and Insertion Sort and a worst case for Quicksort(with bad pivots). Even though sorted arrays are in some ways a best case for the other sorting methods, the big-oh notation of the other methods does not change. Reverse sorted arrays are the worst cases for all methods. All the methods (quicksort with bad pivots) except Merge Sort would have a big-oh notation of o(n^2)). This type of arrays will not make Merge Sort perform at its optimal speed, but the delay is ignored as it does not change the big-oh notation of the method. Arrays that are sorted in random order shows the average performance of the algorithm methods. Also, databases that contain duplicate values are chosen to display a unique performance of quicksort. The method results in a big-oh notation of o(n^2) with such an array and a bad choice of pivot. Moreover, all the other methods but Merge Sort will have the same notation. Although this case is not the best case for the other sorting algorithm, it does not change their big-oh notation. It was not practical to construct even the small arrays manually so the team built array generating methods containing “for loops” to generate the desired arrays. The same array generating methods were used for all sorting algorithms to ensure data consistency.

3 Methods

In order to test and analyze the performance of the algorithms, the team needed to translate the theoretical ideas of each sorting algorithm into practical code that is testable. Using the Java language, the team programmed the sorting algorithm in an Object Oriented Programming (OOP) approach. Thus, the team built an interface class called “Sorter”, which is a blueprint for all sorting algorithms and a class for each sorting algorithm. For the Quicksort method, an abstract class was built to combine the common functionalities that are present between Quicksort classes of different pivots. **Figure 1** shows the design of the program used to test the sorting algorithms.

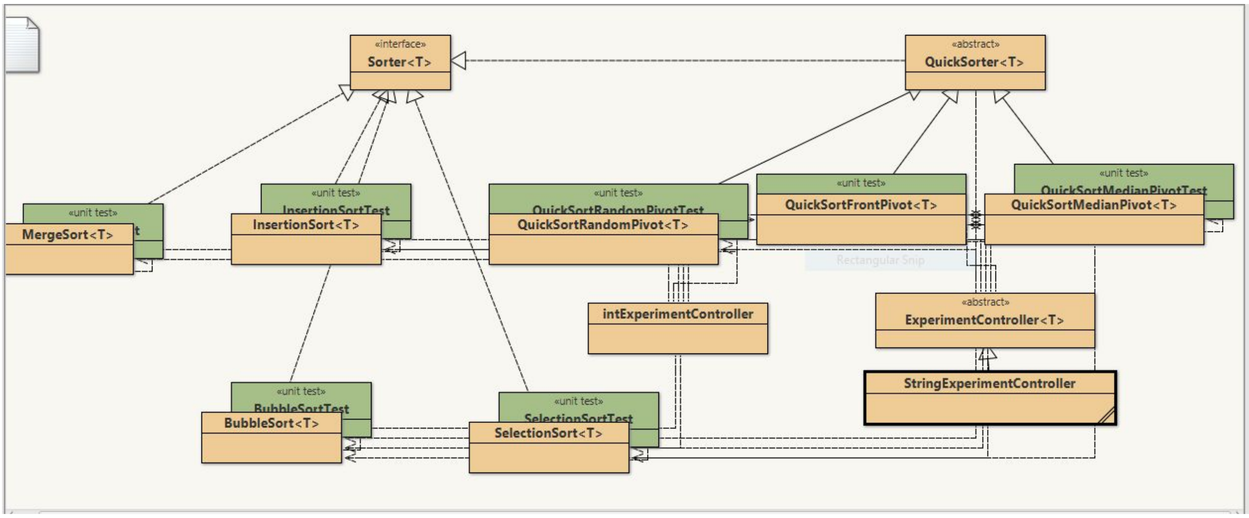
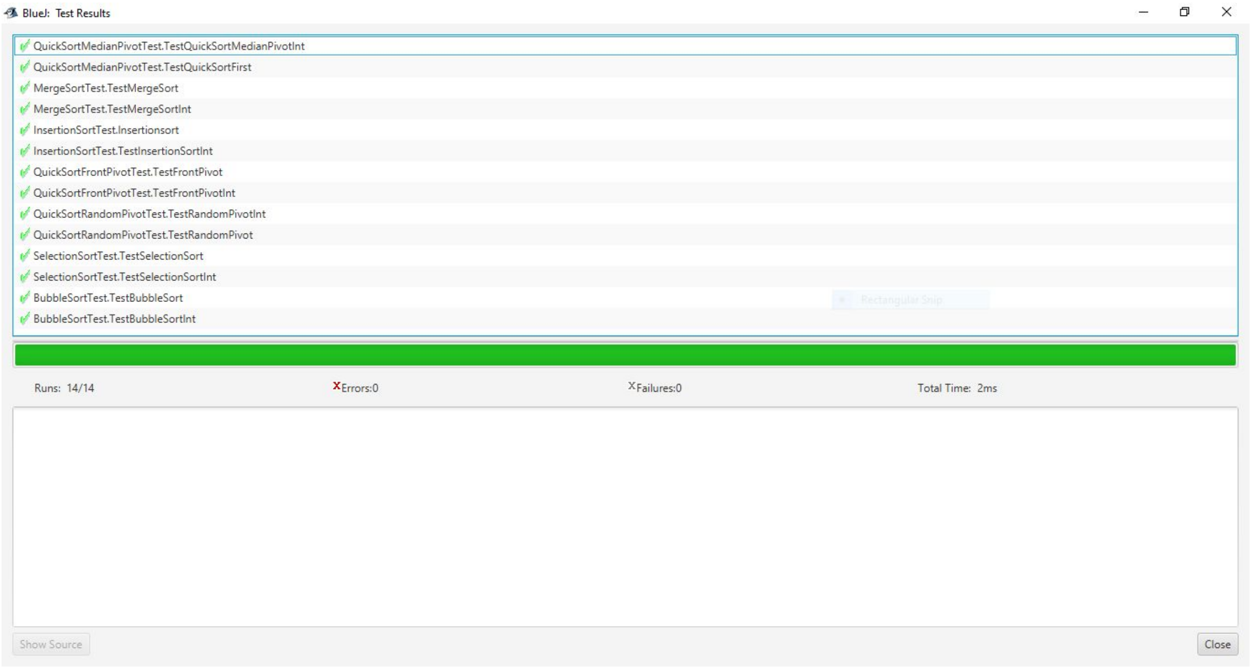


Figure 1

The methods were unit tested in order to check that they behave as they are meant to be.



```
✓ QuickSortMedianPivotTest.TestQuickSortMedianPivotInt
✓ QuickSortMedianPivotTest.TestQuickSortFirst
✓ MergeSortTest.TestMergeSort
✓ MergeSortTest.TestMergeSortInt
✓ InsertionSortTest.Insertionsort
✓ InsertionSortTest.TestInsertionSortInt
✓ QuickSortFrontPivotTest.TestFrontPivot
✓ QuickSortFrontPivotTest.TestFrontPivotInt
✓ QuickSortRandomPivotTest.TestRandomPivotInt
✓ QuickSortRandomPivotTest.TestRandomPivot
✓ SelectionSortTest.TestSelectionSort
✓ SelectionSortTest.TestSelectionSortInt
✓ BubbleSortTest.TestBubbleSort
✓ BubbleSortTest.TestBubbleSortInt

Runs: 14/14          ✗Errors:0          ✗Failures:0          Total Time: 2ms
```

Figure 2

We used a Dell Latitude E5570 to run our experiment. It has a processor INTEL(R) Core(TM0i5-6640HQ CPU 2.60GhZ,2601Mhz, 4 Core. It possesses a RAM of 16 GB. Multiple tests were performed to record time and view growth of the time graph using “currentTimeMills()” which capture the running time of the methods in milliseconds. The timing process was designed in a way such that it only captures the time spent from the moment the sorting algorithm was called to the moment where the method returns a sorted array. Therefore, other operations such as the generation of arrays are not affecting the time readings. Time recordings for a specific algorithm are repeated for all array sizes (27 sizes) and for all array characteristics (5 characteristics). This means that each algorithm had 135 readings. All the times captured were recorded, even if it does not follow our expectations.

Based on Figure 1 above, we have two different experiment controllers, the one measures the time performance of **int** primitive type and the other one measures the time performance of the **reference** type. We decided to separate them because we could not make a generic class that can be used with primitive type.

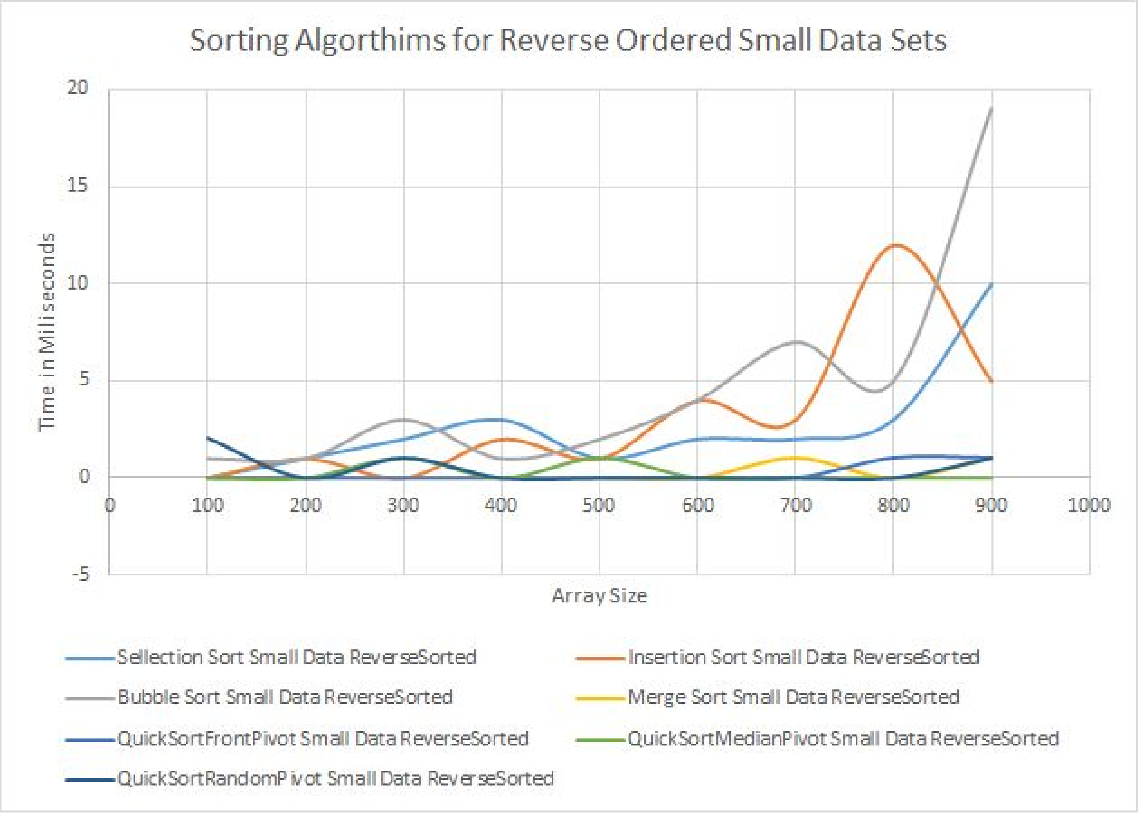
In order to have the most accurate measurements, we tried to space our measurement by at least 30 min before running another test. The idea was to let the computer cool down before doing any test. We also implemented our program in a way that the result will be printed on a csv file. It allowed us to graph the results directly and saved us a lot of time.

4 Data and Analysis

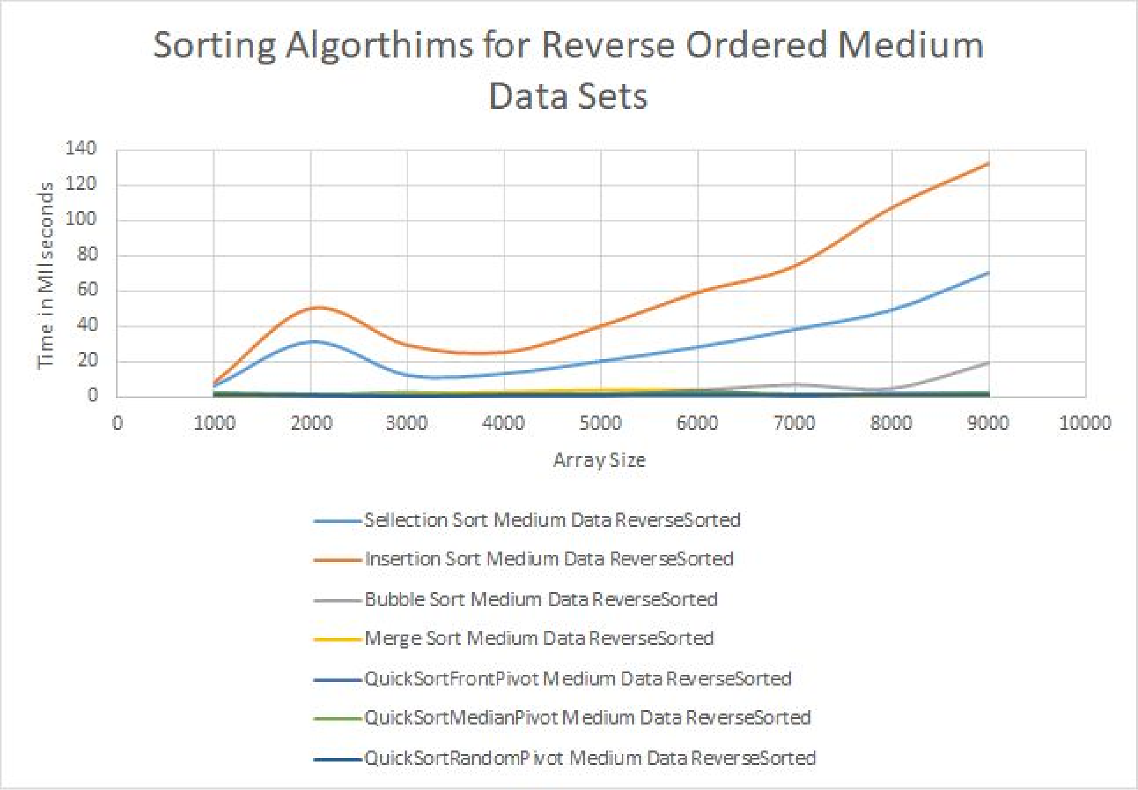
The following are the results of our conducted experiment. The analysis of the data is broken down into the characteristics of the arrays. Additionally, the analysis will look closely at the Quicksort method with different pivots, at Merge Sort and Quicksort's performances with primitive types and reference types, and at the difference between the team's implementation of Quicksort and Merge versus the same methods from the Java standard library.

4.1 Data and Analysis for Different Characteristics

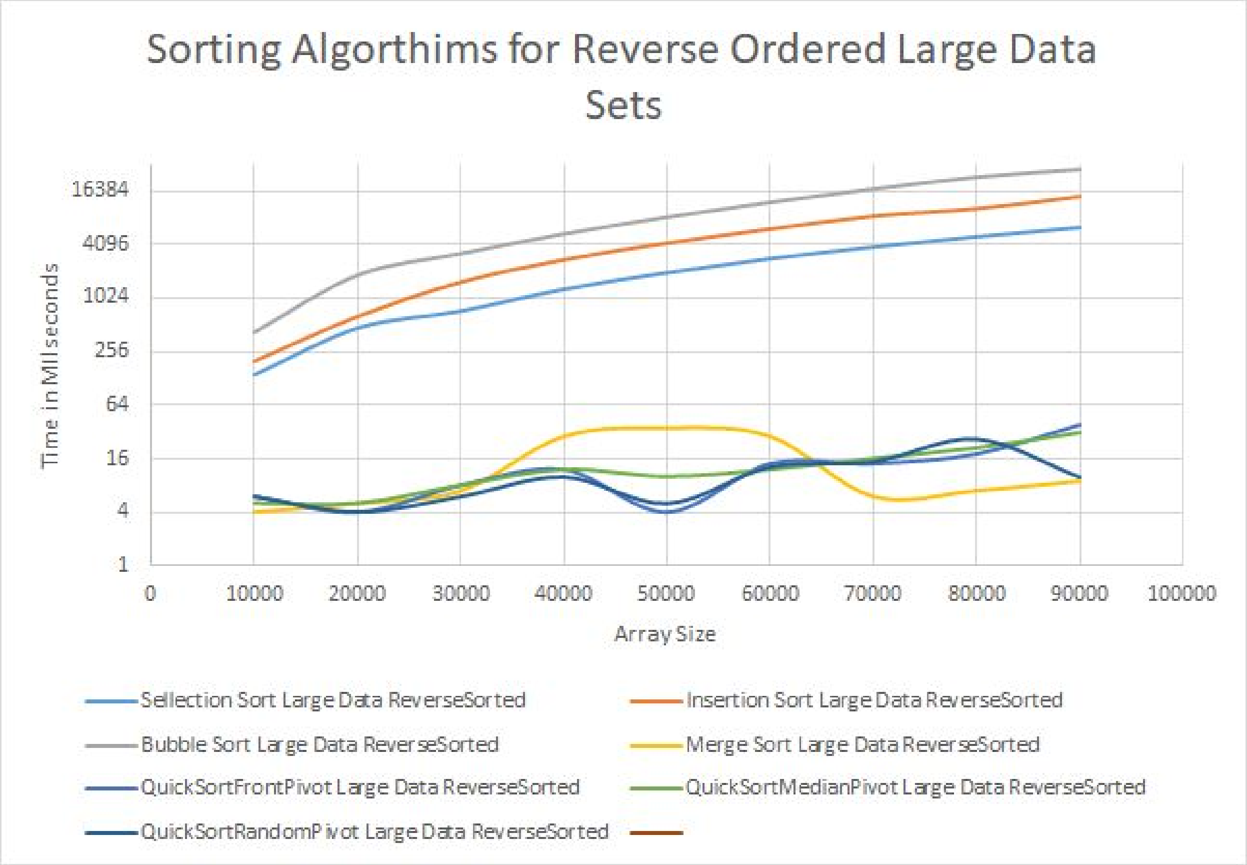
4.1.1 Reverse Order Data:



Graph 1


Sorting Algorthims for Reverse Ordered Medium Data Sets
Time in MIlseconds
0
20
40
60
80
100
120
140
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Array Size
Sellection Sort Medium Data ReverseSorted
Insertion Sort Medium Data ReverseSorted
Bubble Sort Medium Data ReverseSorted
Merge Sort Medium Data ReverseSorted
QuickSortFrontPivot Medium Data ReverseSorted
QuickSortMedianPivot Medium Data ReverseSorted
QuickSortRandomPivot Medium Data ReverseSorted

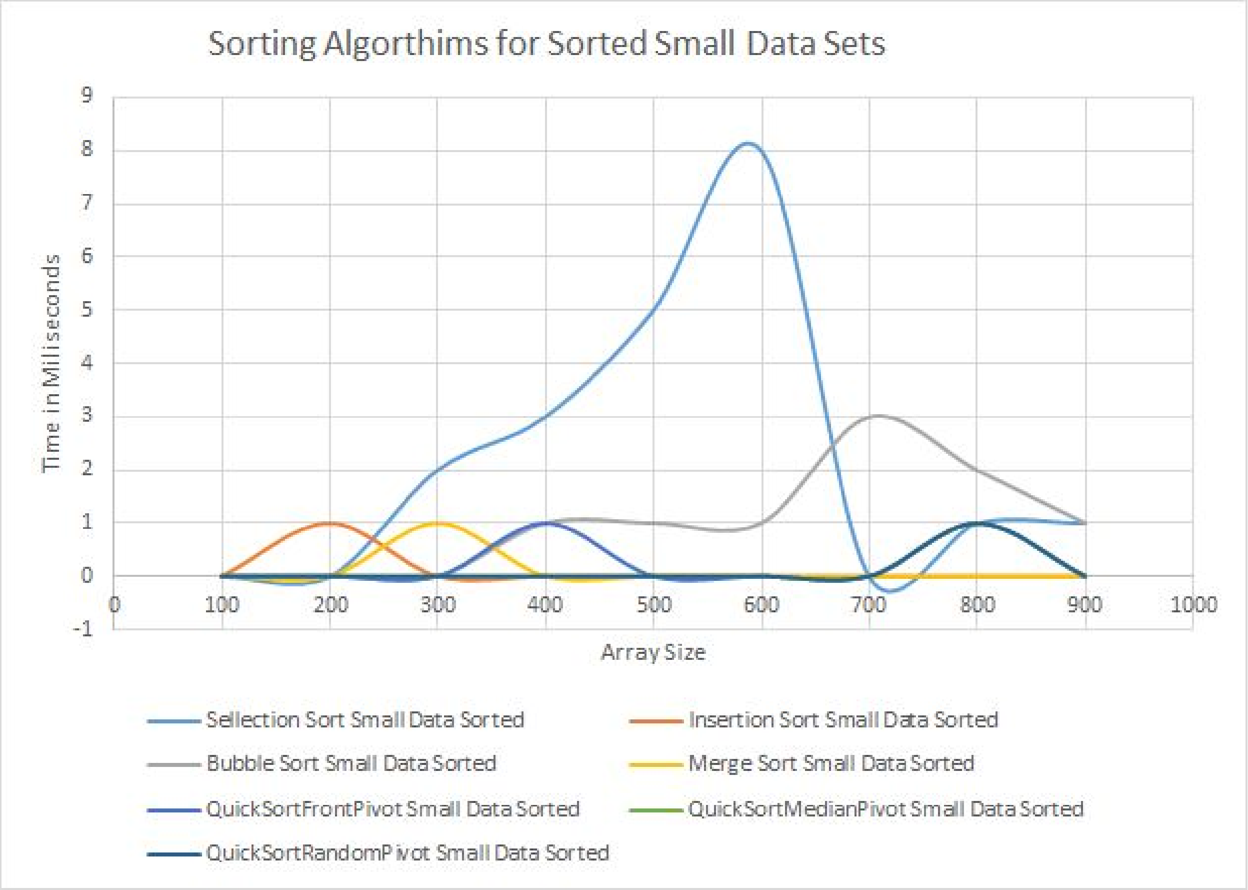
Graph 2



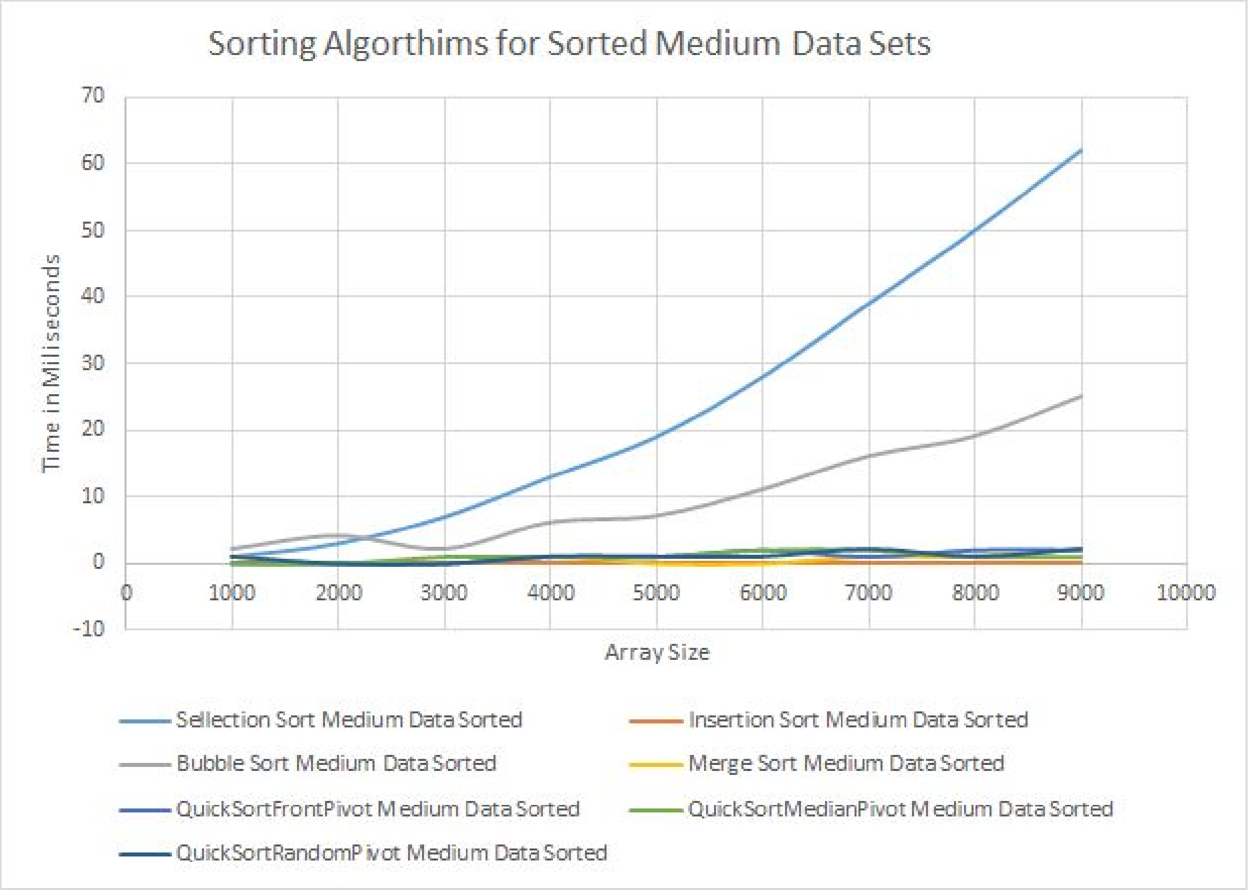
Graph 3

The three graphs above show the running time of the sorting algorithms when given a reverse sorted array. Graph 1 is for small datasets, graph 2 is medium data sets and graph 3 is for large datasets. Because graph 1 displays information for small data sets, the graph does not show an accurate representation of the algorithm performance. When dealing with small data, the readings are mixed up with other operations undertaken by the computer. However, Bubble Sort's curve is merely showing an exponential which closely resembles the expected growth of Bubble Sort. All the other curves are showing sine like trends, which does not represent their theoretical growth. For graphs 2 and 3, the curves are reflecting the performance of the algorithms more accurately. In graph 2, array sizes ranging between 1000 to 4000 are displaying unexpected trends. In this range, the time curves of Insertion Sort and Selection Sort are showing a parabolic trend instead of an exponential one. However, as soon as the size of the arrays goes beyond the mentioned region, the desired curves are drawn. The reason for which the graph acts unexpectedly at the initial stage is because the array sizes are still small to show accurate values. This means that the team should have made the small array sizes extend from 100 arrays to 4000. Finally, graph 3 is a logarithmic graph and it shows the best representation of the growth functions of each sorting algorithm. The graph shows Bubble Sort, Insertion Sort, and Selection Sort having the expected trend of n^2. Due to the large disparity between the running times of the algorithms with a trend of n^2(Bubble Sort, Insertion Sort, and Selection Sort) and the algorithms with the trend of nlog(n) (Merge Sort and Quicksort), it was necessary to make the graph logarithmic to have a clearer view of the lower running time values. Thus, it can still be noted that Merge Sort and Quicksort are having extreme low running times compared to the other methods and that they somewhat follow their expected trend.

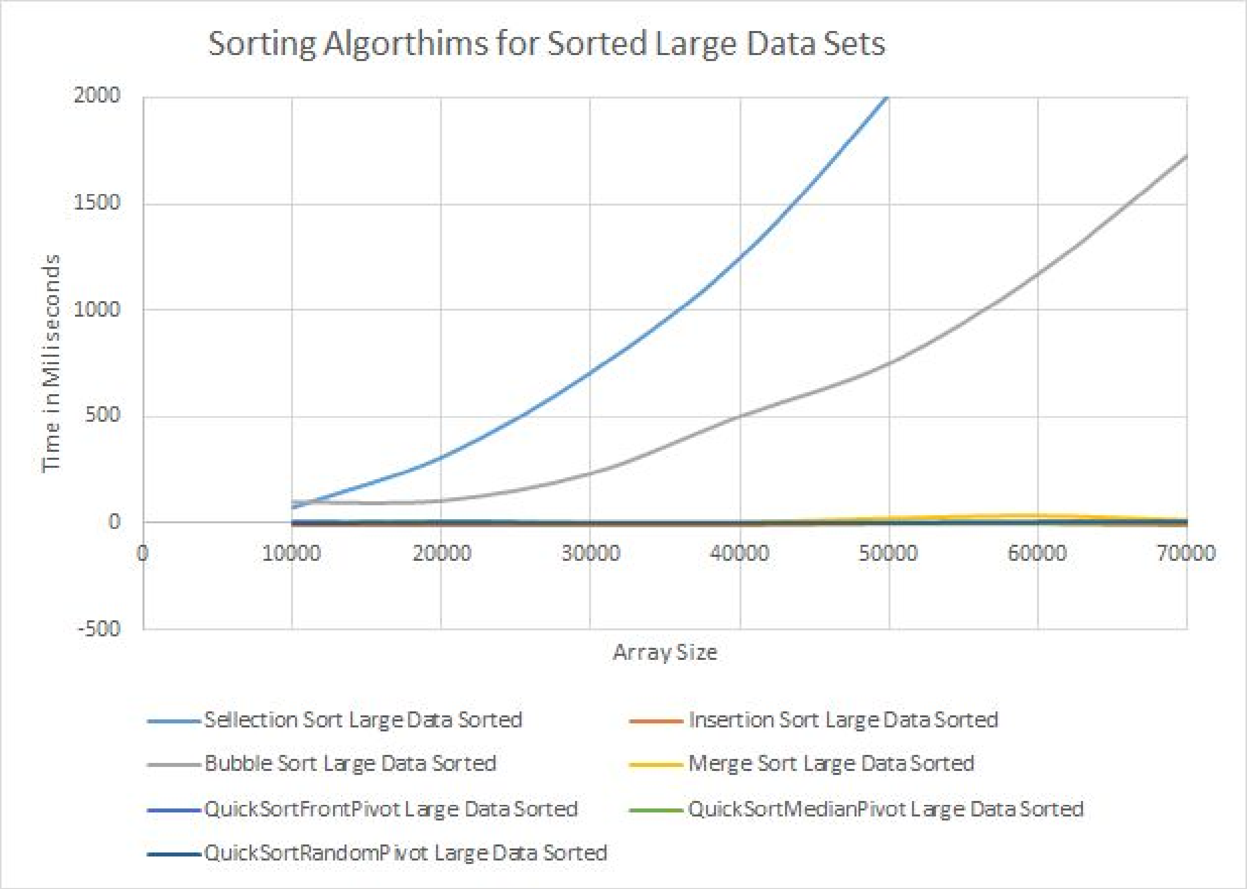
4.1.2 Sorted Data:



Graph 4


Sorting Algorthims for Sorted Medium Data Sets
Time in Miliseconds
70
60
50
40
30
20
10
0
-10
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Array Size
Sellection Sort Medium Data Sorted
Insertion Sort Medium Data Sorted
Bubble Sort Medium Data Sorted
Merge Sort Medium Data Sorted
QuickSortFrontPivot Medium Data Sorted
QuickSortMedianPivot Medium Data Sorted
QuickSortRandomPivot Medium Data Sorted

Graph 5

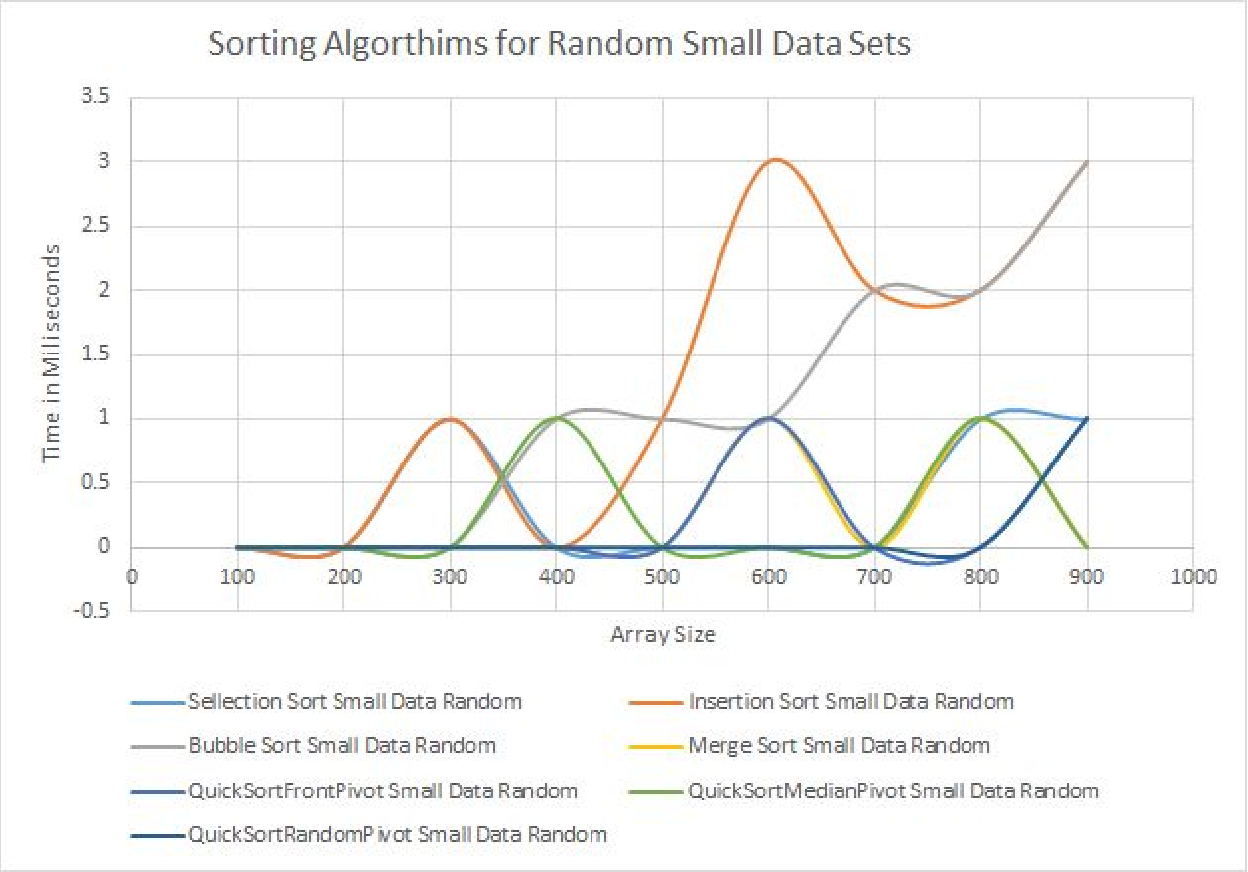


Graph 6

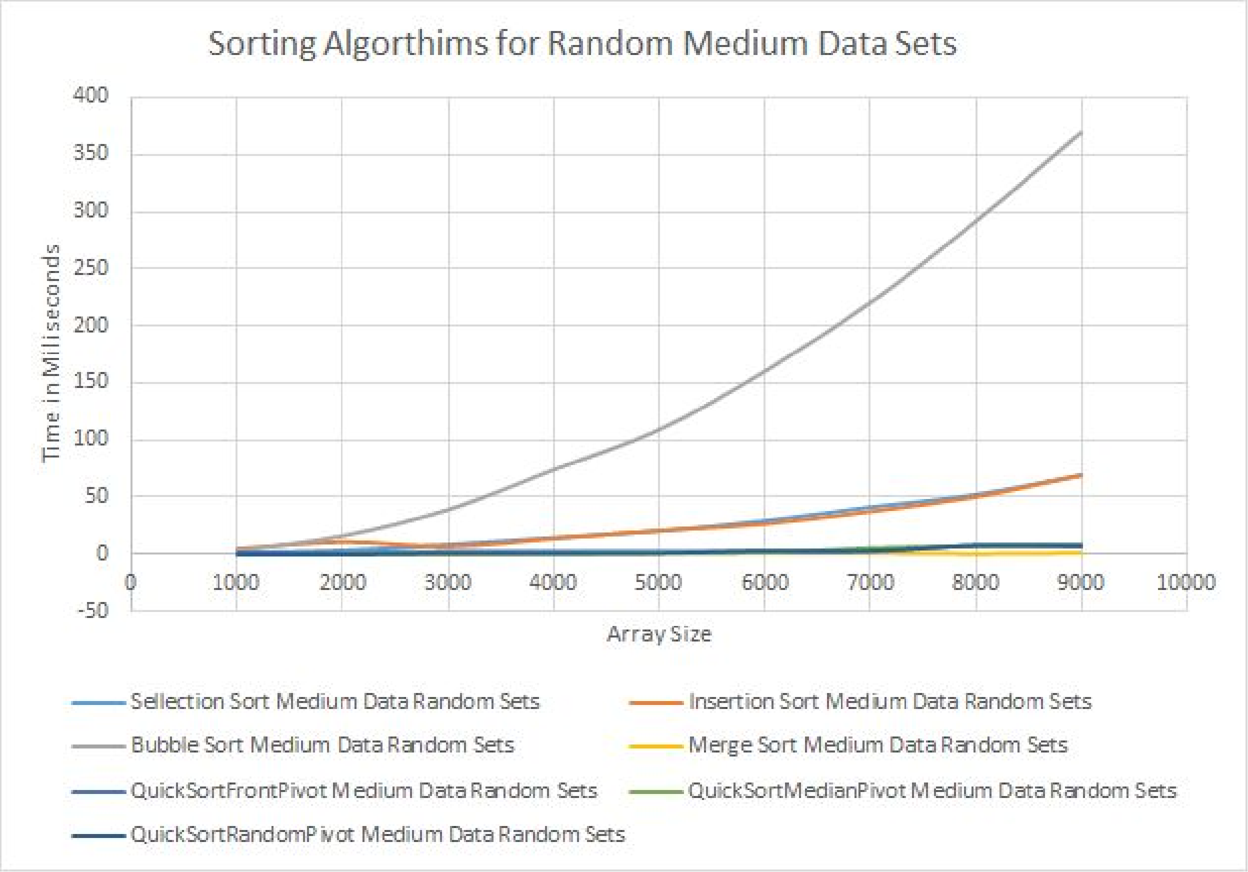
The three graphs above show the running time of the sorting algorithms when given a sorted array. Graph 1 is for small datasets, graph 2 is medium data sets and graph 3 is for large datasets. Because graph 1 displays information for small data sets, the graph does not show an accurate representation of the algorithm performance. When dealing with small data, the readings are mixed up with other operations undertaken by the computer. All the curves are showing sine like trends, which does not represent their theoretical growth. Theoretically, sorted arrays are best cases for all algorithms. Thus, algorithms such as Bubble Sort and Insertion Sort, will have a big-oh notation of o(n) when dealing with such datasets.In addition, the running time of Selection Sort is always o(n^2) and Quicksort and Merge Sort will still maintain their o(nlog(n)) notation. Graphs 5 and 6 prove the exceptions for each algorithm, but not quit for

Bubble Sort. Although Bubble Sort is not showing sharp exponential growth in graph 6, the curve is still not linear as expected. However for medium array sizes, Bubble Sort is showing a more linear trend. The cause of the unexpected result in graph 6 is related to the use of large datasets.

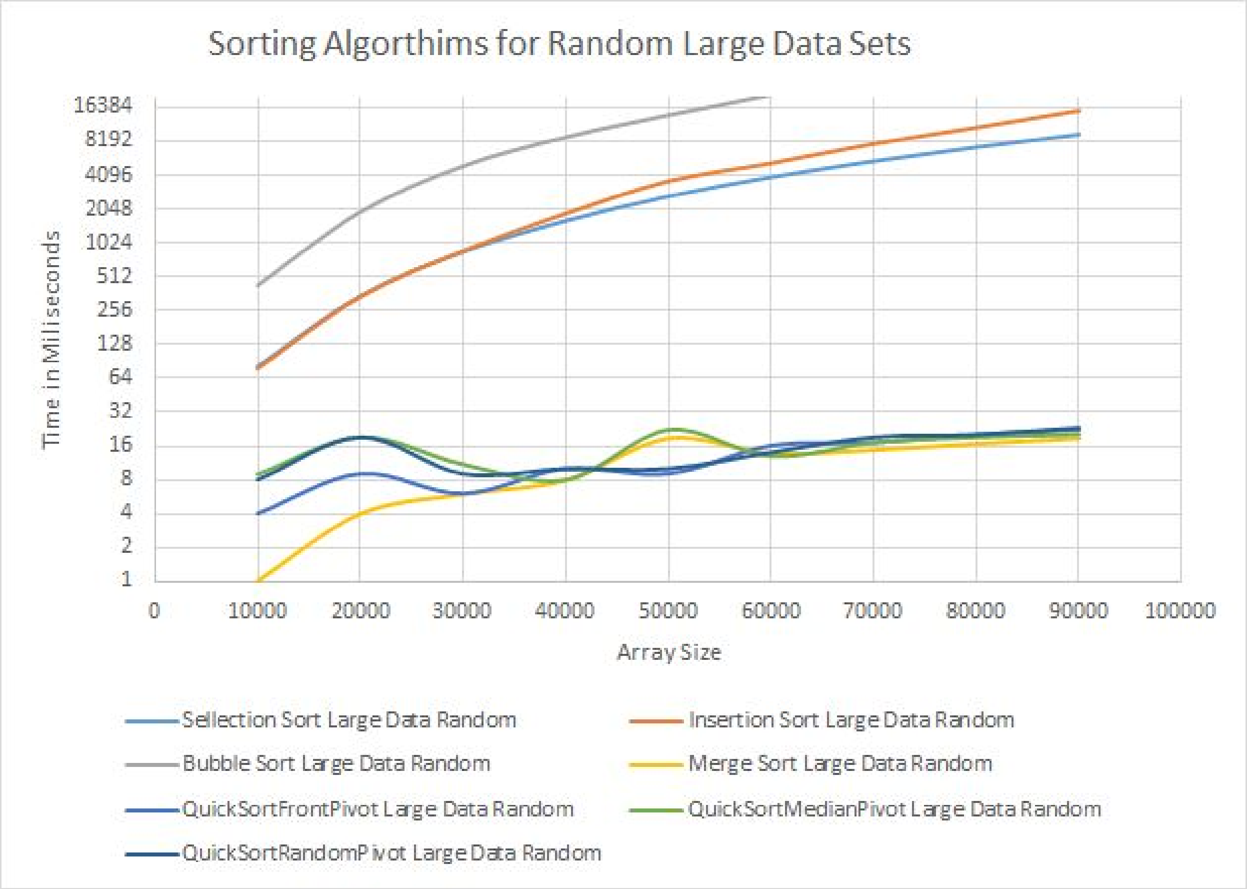
4.1.3 Array with Elements in Random Order Data:



Graph 7


Sorting Algorthims for Random Medium Data Sets
Time in Miliseconds
400
350
300
250
200
150
100
50
0
-50
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Array Size
Sellection Sort Medium Data Random Sets
Insertion Sort Medium Data Random Sets
Bubble Sort Medium Data Random Sets
Merge Sort Medium Data Random Sets
QuickSortFrontPivot Medium Data Random Sets
QuickSortMedianPivot Medium Data Random Sets
QuickSortRandomPivot Medium Data Random Sets

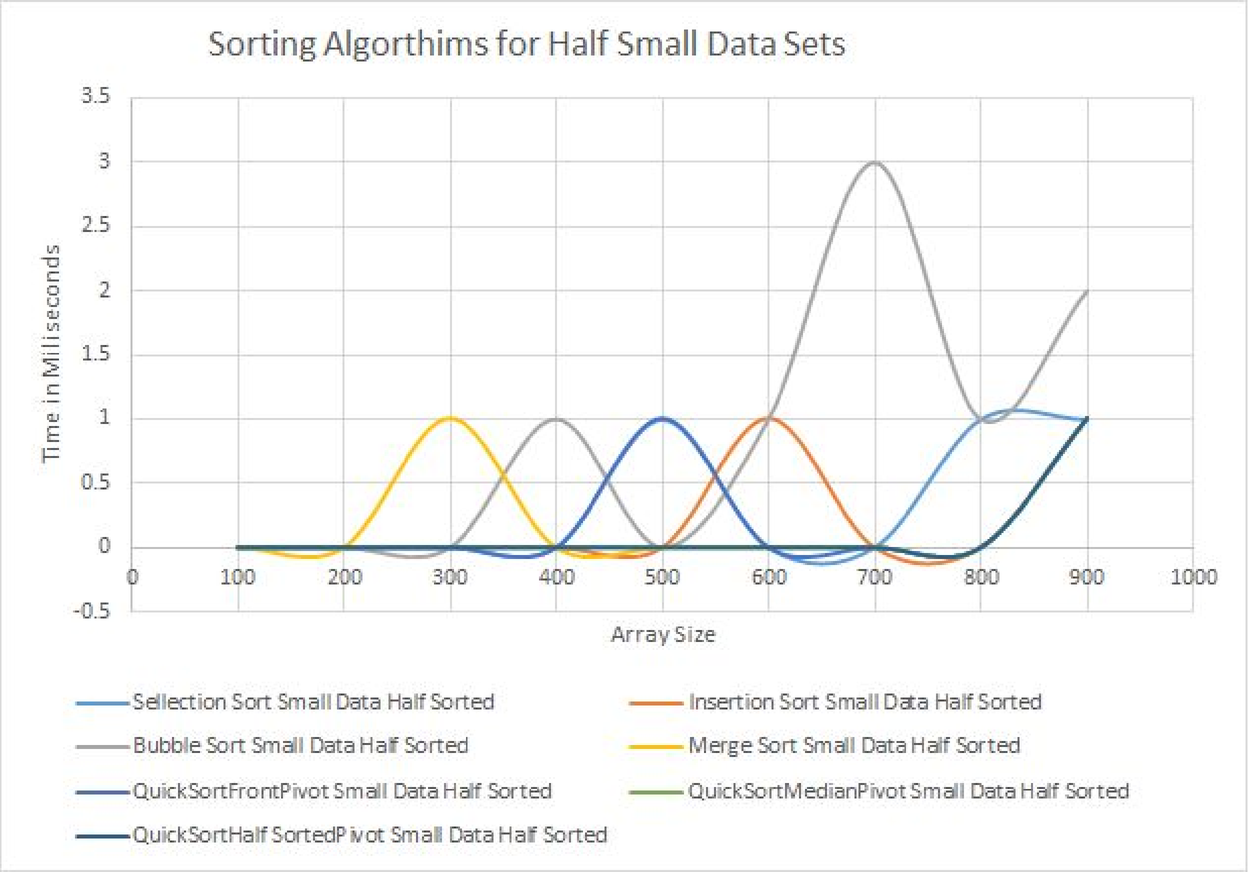
Graph 8



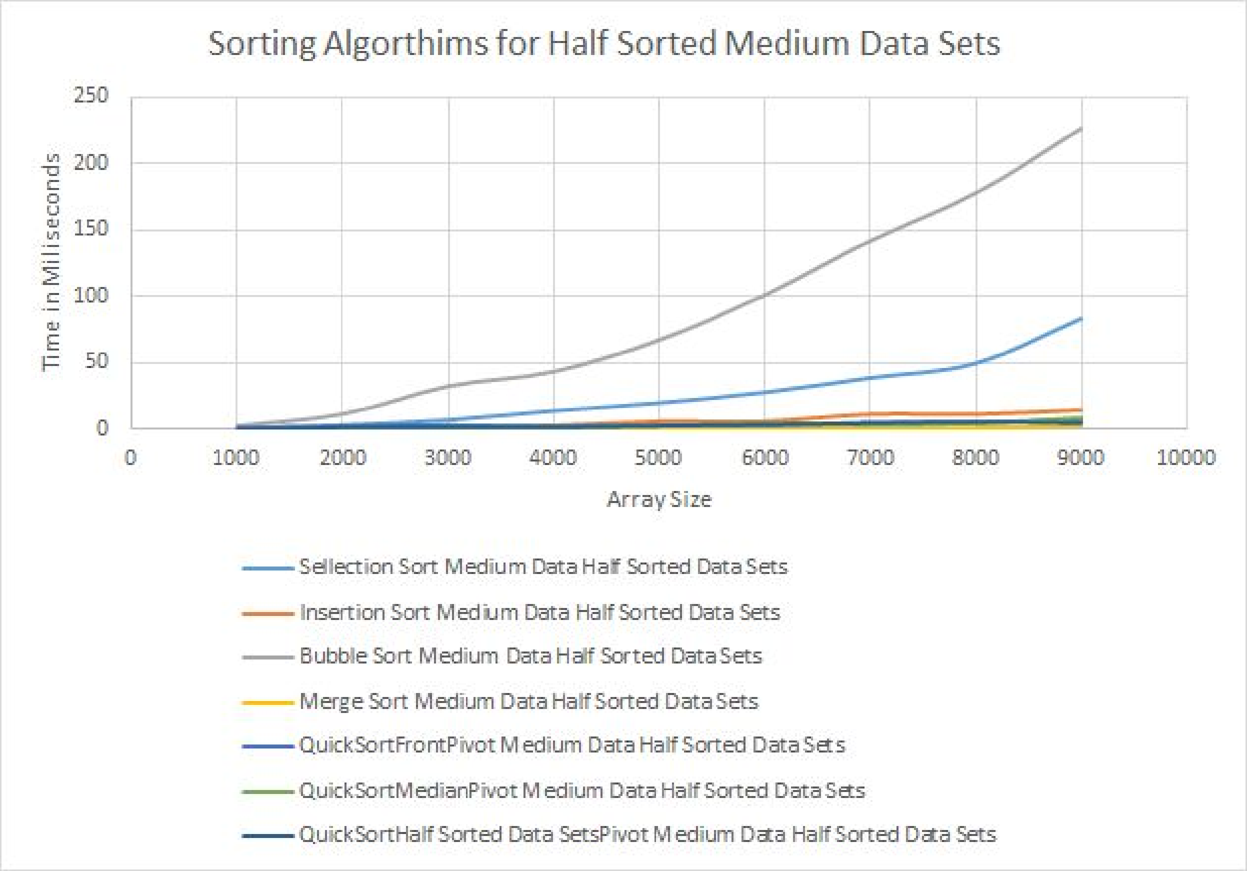
Graph 9

The three graphs above show the running time of the sorting algorithms when given a random ordered array. Graph 7 is for small datasets, graph 8 is medium data sets and graph 9 is for large datasets. Because graph 7 displays information for small data sets, the graph does not show an accurate representation of the algorithm performance. When dealing with small data, the readings are mixed up with other operations undertaken by the computer. However, Bubble Sort's curve is merely showing an exponential which closely resembles the expected growth of Bubble Sort. All the other curves are showing sine like trends, which does not represent their theoretical growth. Having the elements in random order triggers the average performance for all sorting algorithms. For graphs 8 and 9, the curves are reflecting the performance of the algorithms more accurately. In graph 8, Bubble Sort, Selection Sort, and Insertion Sort are showing n^2 trends which follows the expected results. Moreover, all the Quicksort methods and Merge Sort are displaying their excepted o(nlog(n)). Graph 9 is a logarithmic graph which shows the same characteristics discussed for graph 8, but with larger values.

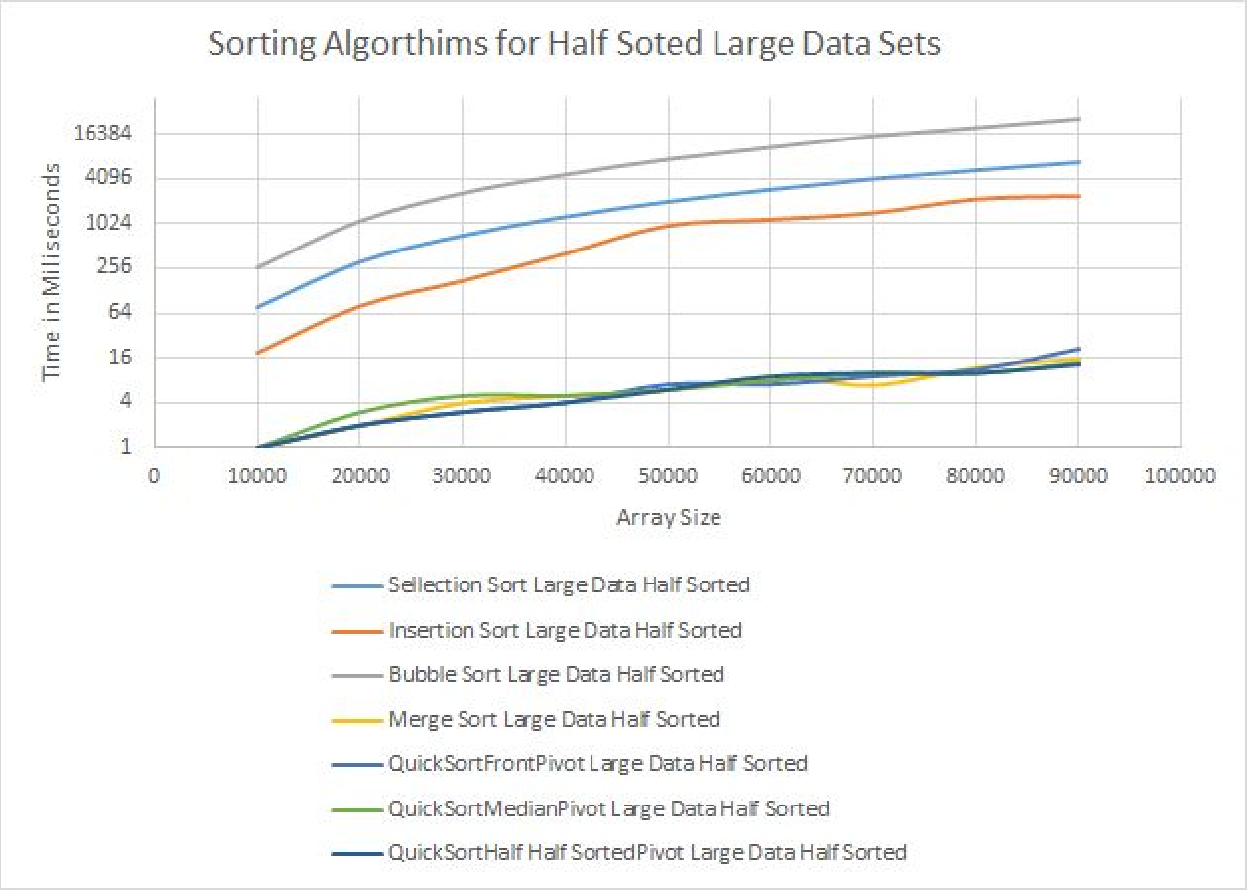
4.1.4 Half Sorted Element Data:



Graph 10


Sorting Algorthims for Half Sorted Medium Data Sets
250
200
150
100
50
0
Time in Miliseconds
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Array Size
Sellection Sort Medium Data Half Sorted Data Sets
Insertion Sort Medium Data Half Sorted Data Sets
Bubble Sort Medium Data Half Sorted Data Sets
Merge Sort Medium Data Half Sorted Data Sets
QuickSortFrontPivot Medium Data Half Sorted Data Sets
QuickSortMedianPivot Medium Data Half Sorted Data Sets
QuickSortHalf Sorted Data SetsPivot Medium Data Half Sorted Data Sets

Graph 11

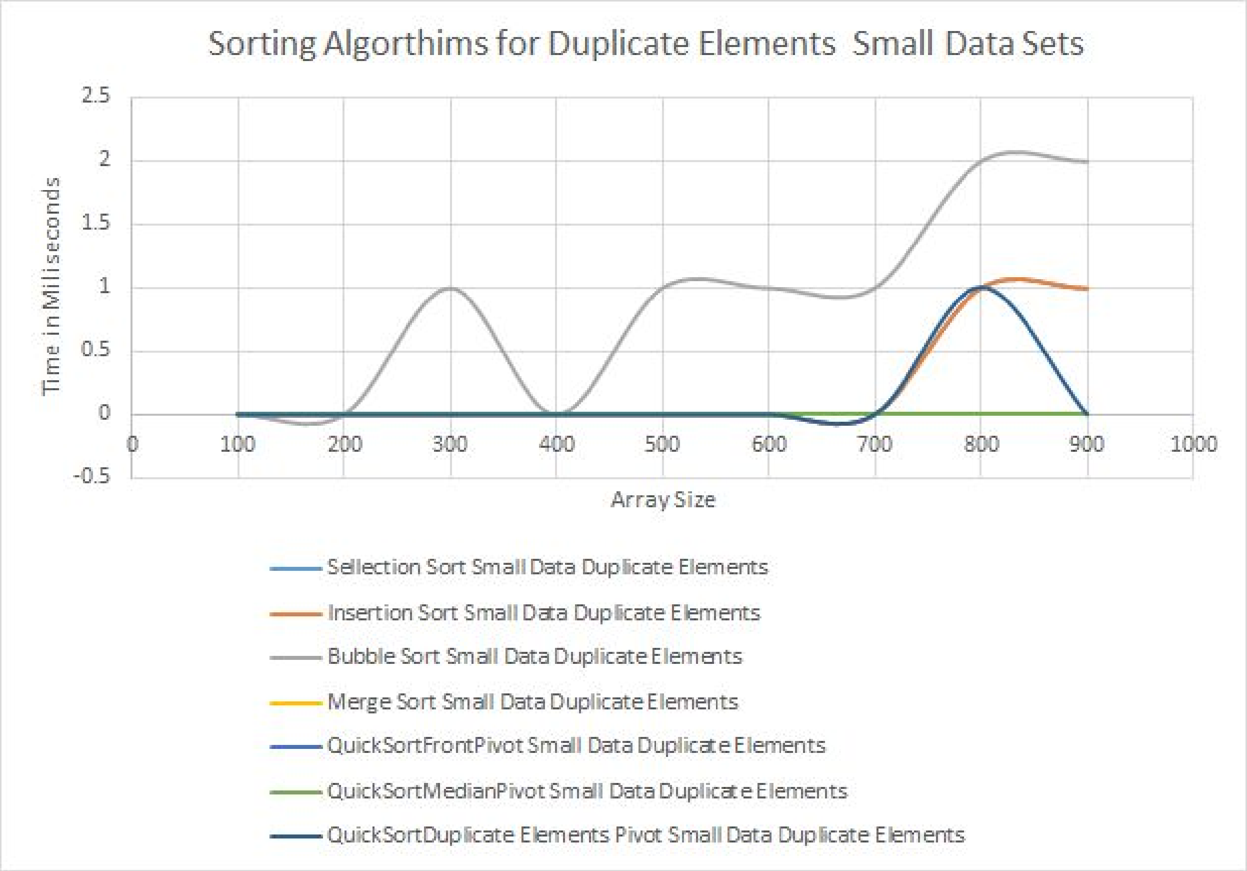


Graph 12

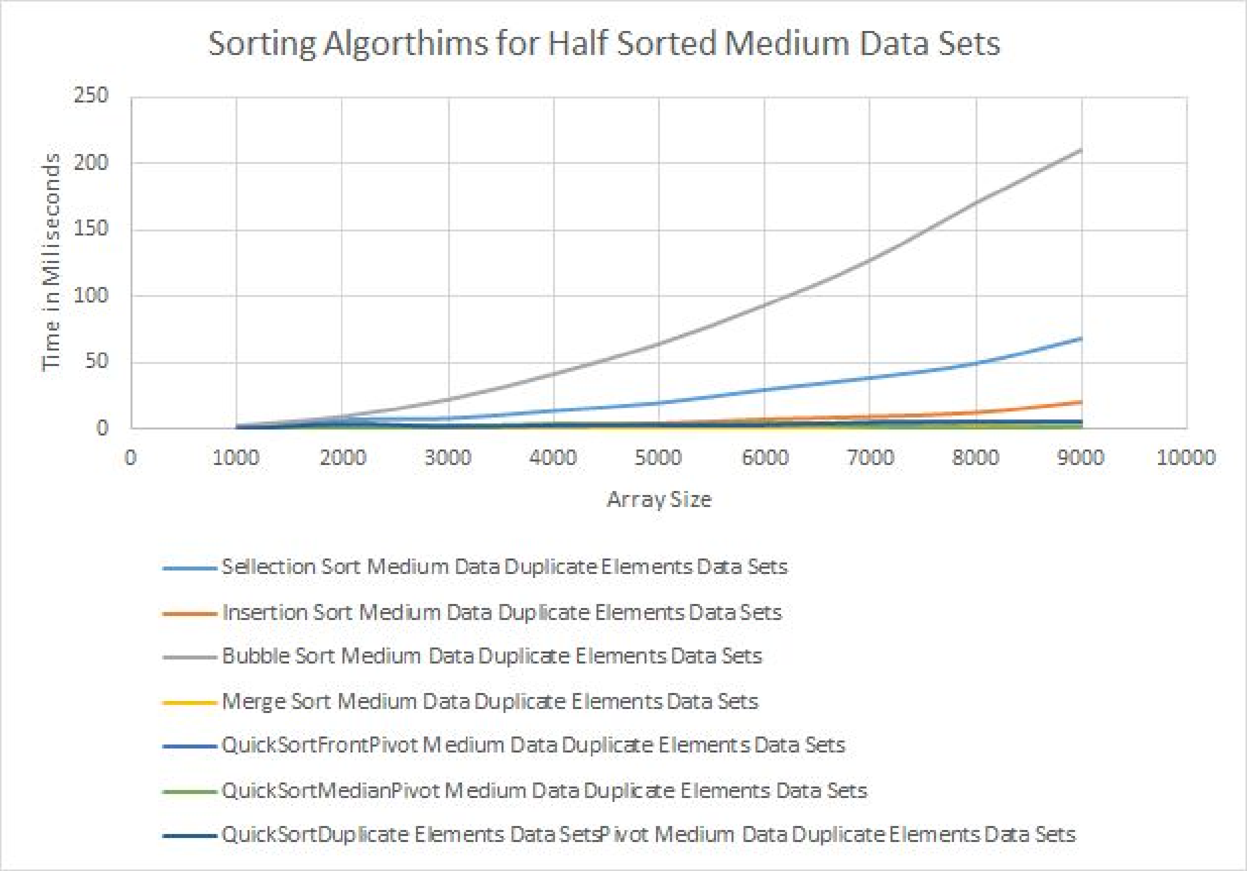
The three graphs above show the running time of the sorting algorithms when given an array that is half sorted and half reversely sorted. Graph 10 is for small datasets, graph 11 is medium data sets, and graph 12 is for large datasets. Because graph 10 displays information for small data sets, the graph does not show an accurate representation of the algorithm performance. When dealing with small data, the readings are mixed up with other operations undertaken by the computer. All the curves are showing sine-like trends, which do not represent their theoretical growth. Having the first half of the elements sorted and the other half to be not sorted triggers the average performance for all sorting algorithms. For graphs 11 and 12, the curves are reflecting the performance of the algorithms more accurately. Graph 11, Bubble Sort, Selection Sort, and Insertion Sort are showing n^2 trends that follow the expected results. Moreover, all the

Quicksort methods and Merge Sort are displaying their excepted o(nlog(n)). Graph 12 is a logarithmic graph which shows the same characteristics discussed for graph 8, but with larger values.

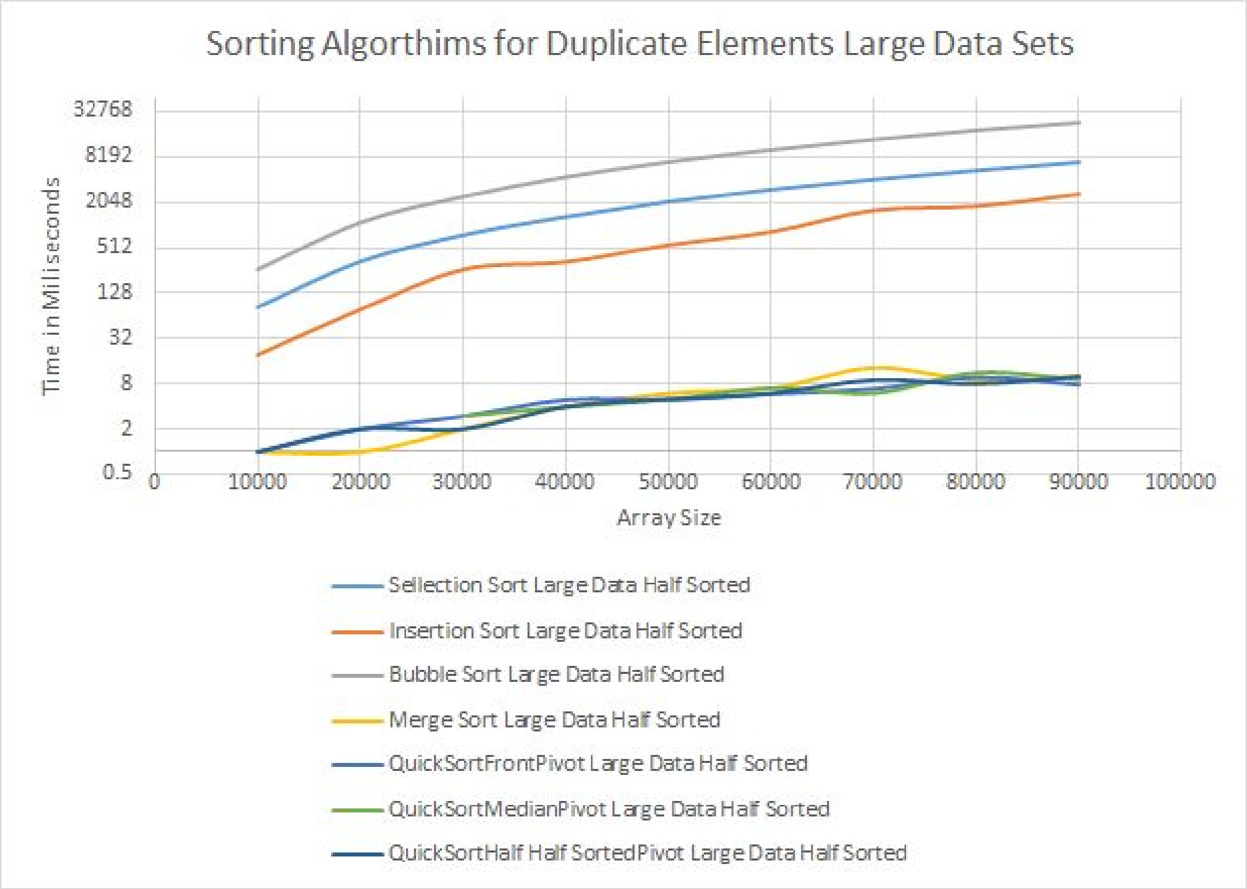
4.1.5 Arrays with Duplicate Elements



Graph 13


Sorting Algorthims for Half Sorted Medium Data Sets
Time in Miliseconds
0
50
100
150
200
250
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Array Size
Sellection Sort Medium Data Duplicate Elements Data Sets
Insertion Sort Medium Data Duplicate Elements Data Sets
Bubble Sort Medium Data Duplicate Elements Data Sets
Merge Sort Medium Data Duplicate Elements Data Sets
QuickSortFrontPivot Medium Data Duplicate Elements Data Sets
QuickSortMedianPivot Medium Data Duplicate Elements Data Sets
QuickSortDuplicate Elements Data SetsPivot Medium Data Duplicate Elements Data Sets

Graph 14

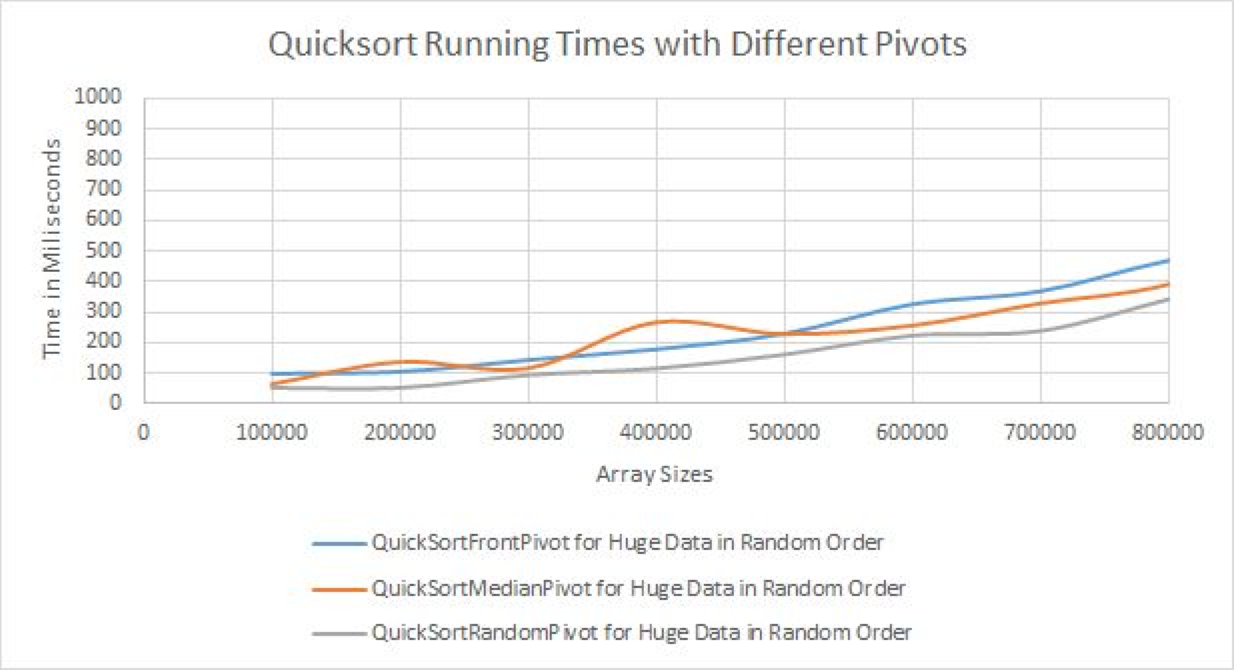


Graph 15

The three graphs above show the running time of the sorting algorithms when given an array with some duplicate values. Graph 13 is for small datasets, graph 14 is medium data sets, and graph 15 is for large datasets. Because graph 13 displays information for small data sets, the graph does not show an accurate representation of the algorithm performance. When dealing with small data, the readings are mixed up with other operations undertaken by the computer. All the curves are showing sine-like trends, which do not represent their theoretical growth. Having some elements as duplicates and the other some as random triggers the average case performance of each algorithm. For graphs 13 and 13, the curves are reflecting the performance of the algorithms more accurately. Graph 13, Bubble Sort, Selection Sort, and Insertion Sort are showing n^2 trends that follow the expected results. Moreover, all the Quicksort methods and

Merge Sort are displaying their excepted o(nlog(n)). Graph 13 is a logarithmic graph which shows the same characteristics discussed for graph 8, but with larger values.

4.2 Data and Analysis for Quicksort with different pivots



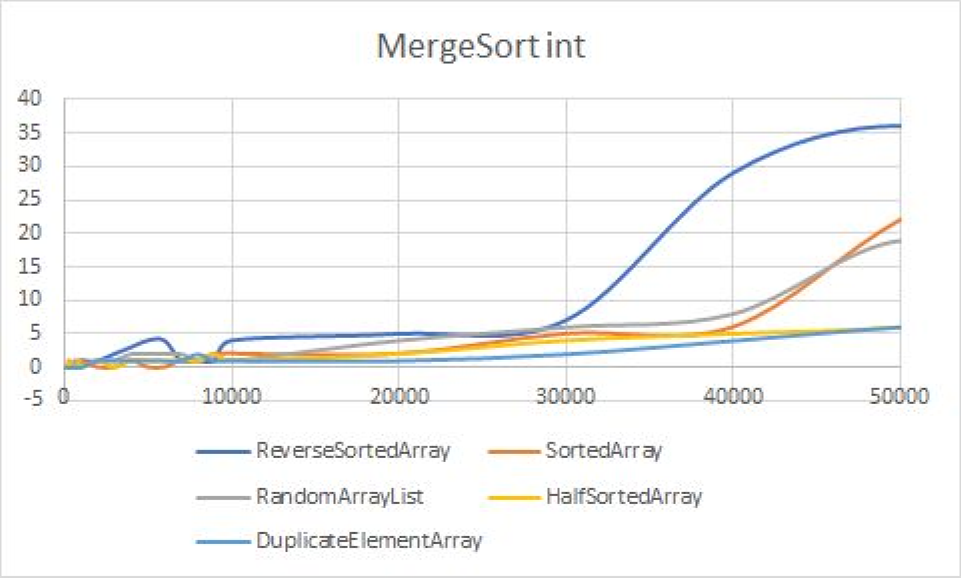
Graph 16

The graph above shows the running time for Quicksort with a front pivot, a median pivot, and a random pivot. Because those methods have low running times, the sizes of the arrays used to compare the methods range from 100000 to 800000. Furthermore, the arrays used were in random order. This characteristic was chosen to trigger the average performance of each algorithm. As expected, the graph shows all three methods trending in the same fashion. The curves only differ by constants. Following the theoretical expectations, having a front pivot takes the longest time compared with random and median pivots.

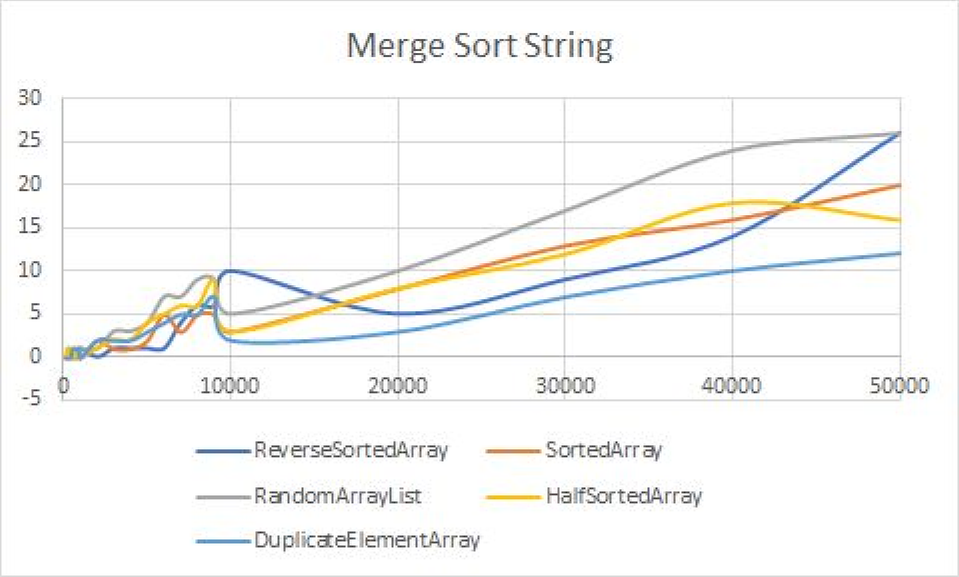
4.3 Data and Analysis for Quick Sort and Merge Sort performances with primitive types and reference types

The following shows how Quicksort(with random pivot, median pivot, and front pivot) and Merge Sort perform with permeative types (int) and reference types (Strings).

Merge Sort:



Graph 17



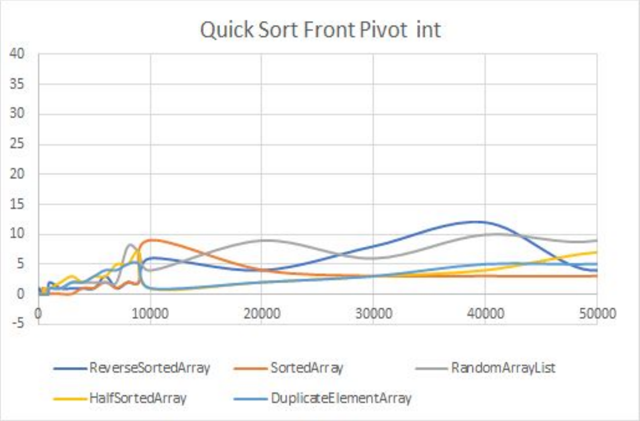
Graph 18

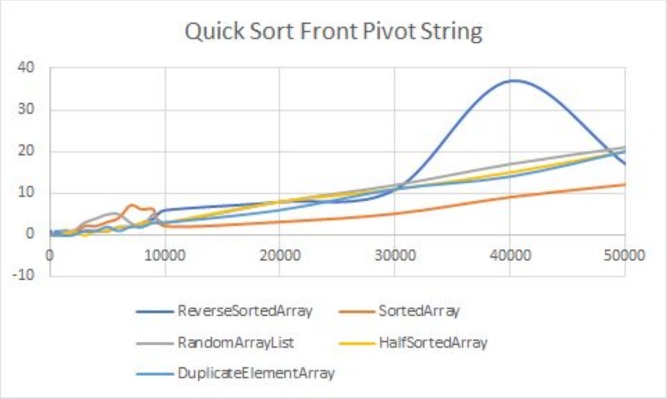
For both Strings and Ints, arrays with elements fewer than 10000 show unexpected behavior. However, as the number of arrays increases, the curves get smoother and clearer. Although Reverse Ordered Arrays do not follow, it can be concluded from the graphs above that sorting reference types (Strings) takes much longer than primitive types (ints). The team assumes that the reason for this difference is that reference types are stored in the heap as opposed to the primitive types which are stored in the stack. Accessing data from the stack is faster than accessing it from the heap.

Quicksort:

Graph 21

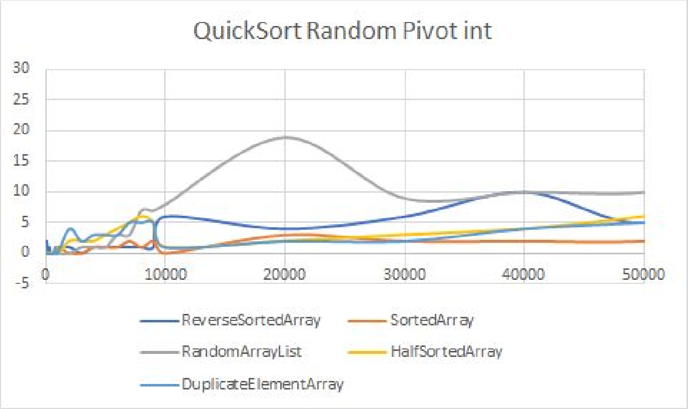
Graph 22

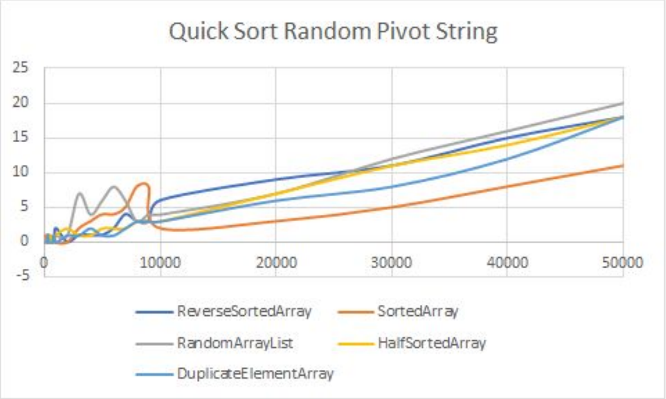




Graph 21

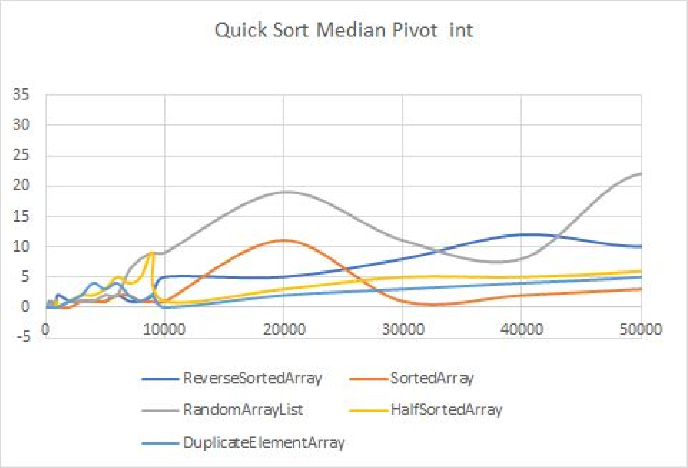
Graph 22

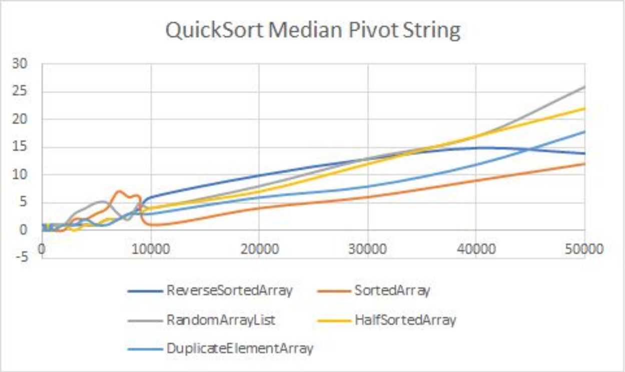




Graph 20

Graph 20



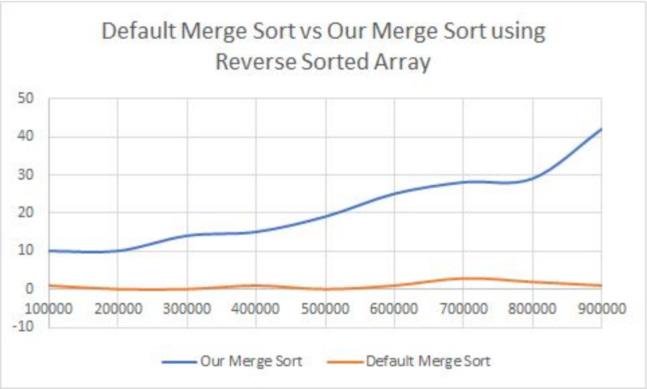


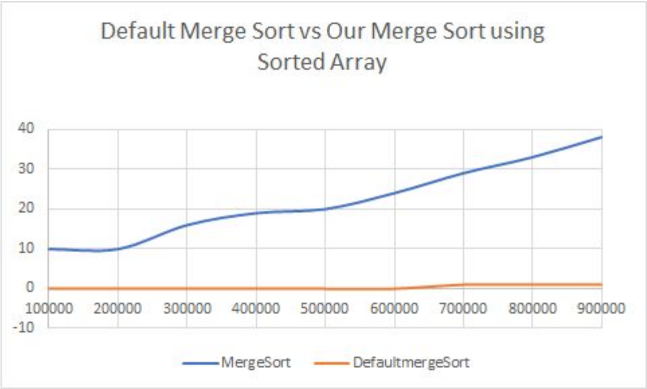
The above graphs compares the running times for Quicksorts with reference types(Strings) and primitive types (int). Throughout all the pivots, it can be seen that the overall running time for Quicksorts with ints is faster than the Quicksorts with Strings. Again, the team assumes that the reason for this difference is that reference types are stored in the heap as opposed to the primitive types which are stored in the stack. Accessing data from the stack is faster than accessing it from the heap. Moreover, when data sizes are less than 10000 all graphs display data that is very unexpected. After that benchmark graphs 20 and 22 have all curves following the expected outcome. Although most of the graphs are displaying curves with the expected trends, some of them are showing unexpected sin-like trends.

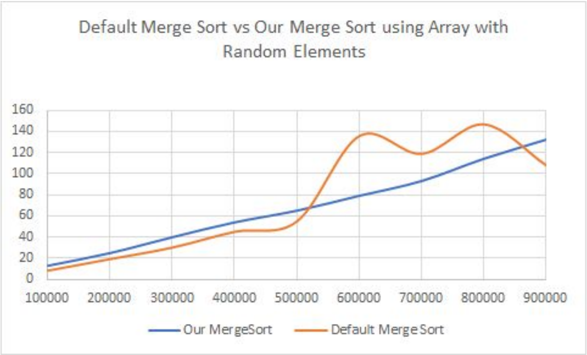
4.4 Data and Analysis of the Performance of Our Sorting Methods and Java's Default Sorting Method

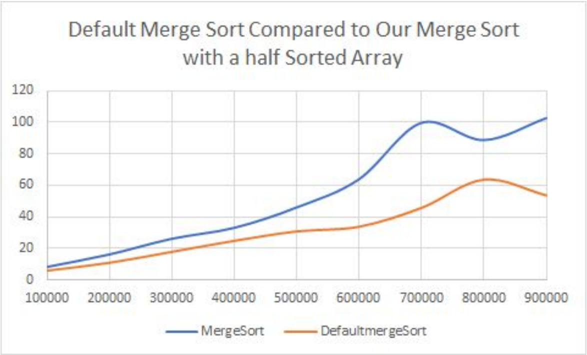
MergeSort

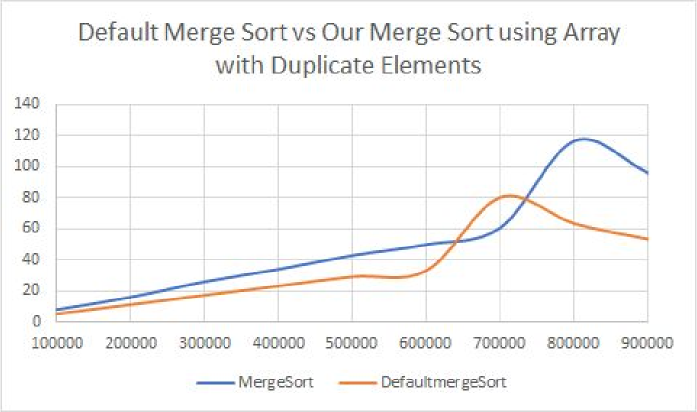
Java possesses a method that uses merge sort directly. We implemented that in our program and compared it with our own program. The following are the result based on the array generated





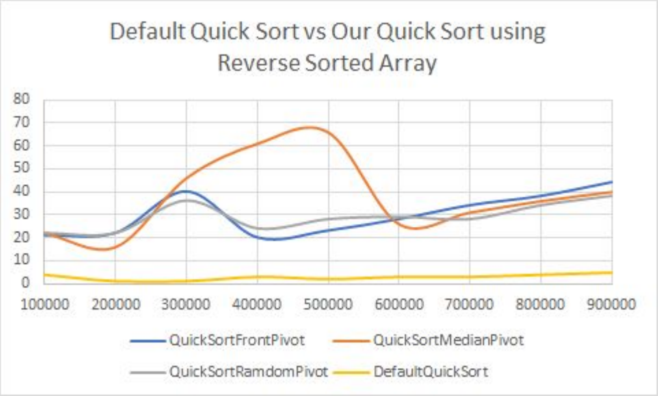


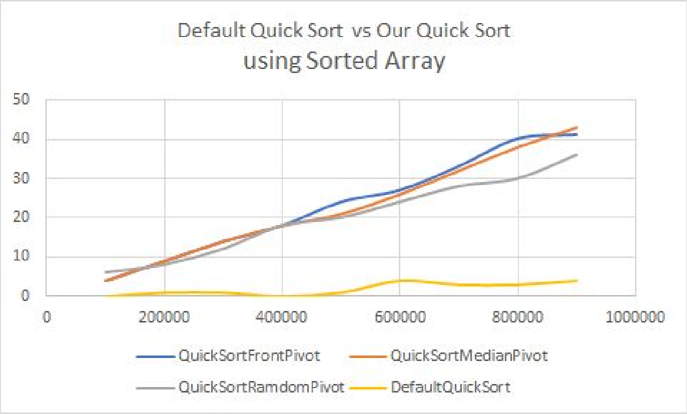


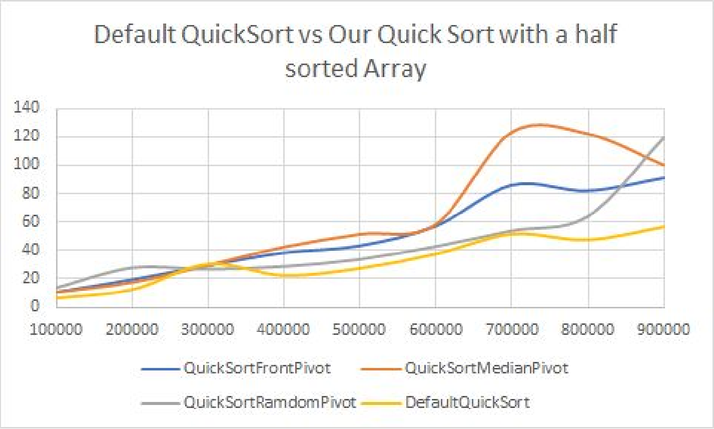


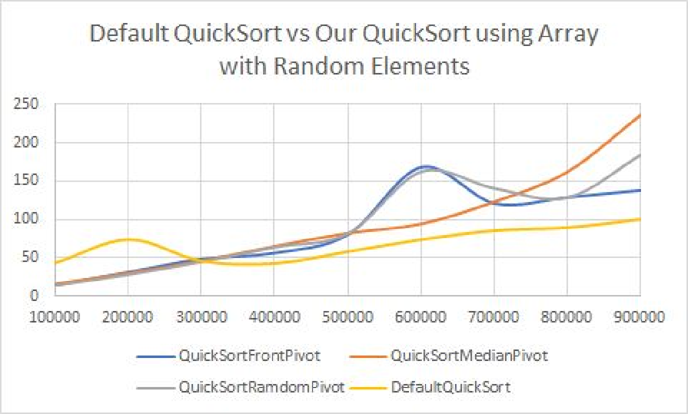
Java's merge sort is much more efficient to sort a sorted array and a reverse sorted array compared to our implementation. However, the two algorithms start to perform the same time when it comes to unsorted arrays. Despite that, the Java Native Merge sort is still quite performant. Merge sort is also known as for its use of space but unfortunately our measurement does not allow us to do such analysis.

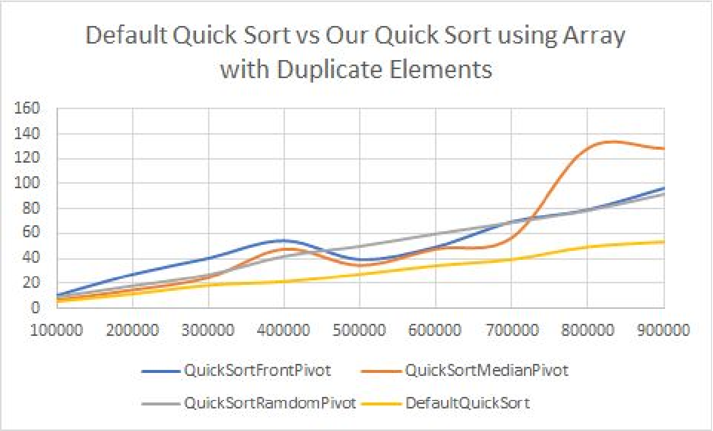
QuickSort







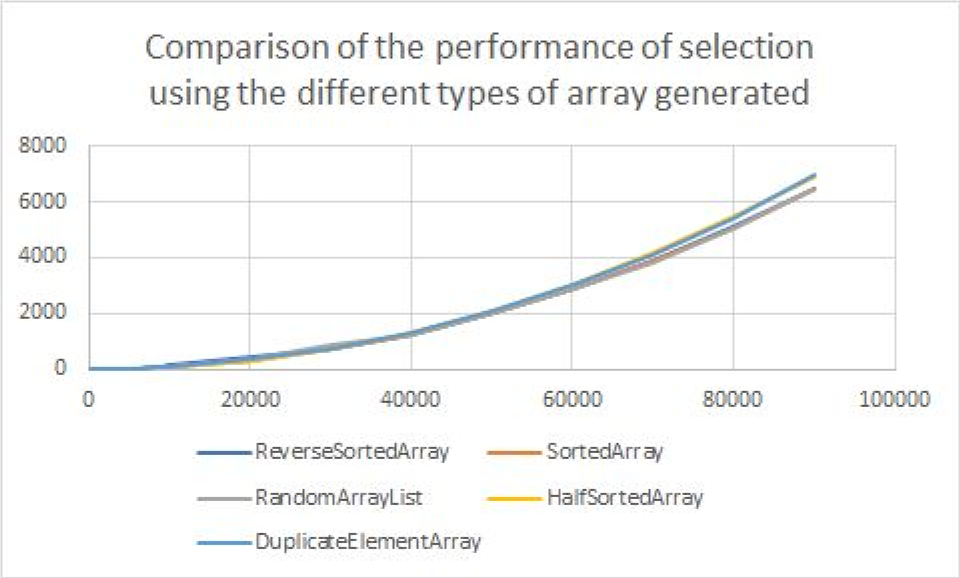




Similar to Java's native merge sort, this algorithm is quite efficient when it comes to sorting a sorted array (in ascending or descending order) . However, it relies on the size it sorts an unsorted data. We implemented free different types of quick sort. However, it is quite hard to predict which one behaves similar to the native algorithm.

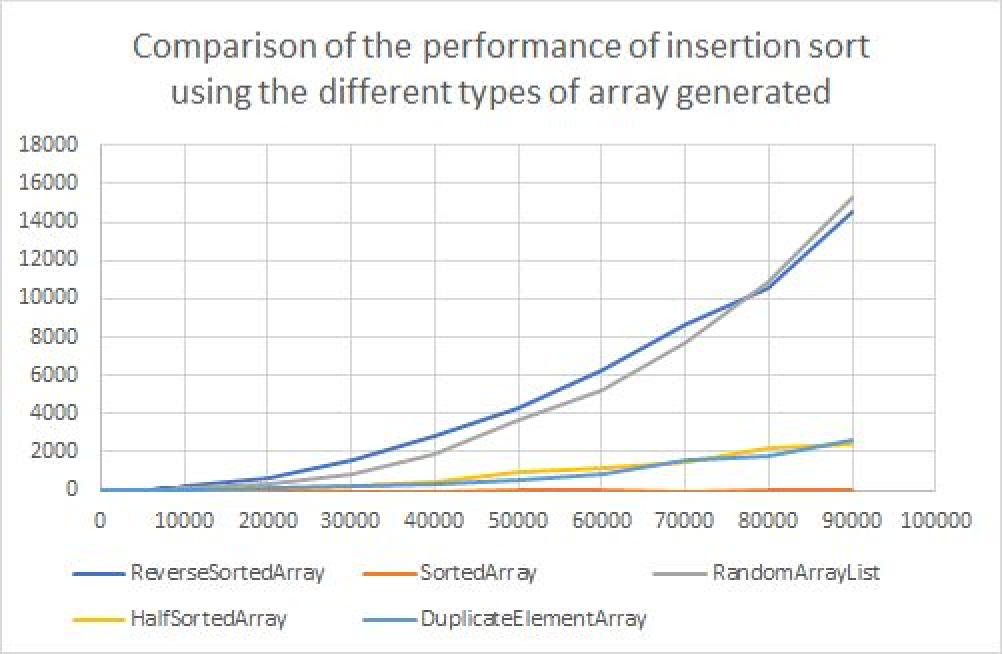
4.5 Data and Analysis for each individual sorting Algorithm

4.5.1Selection Sort.



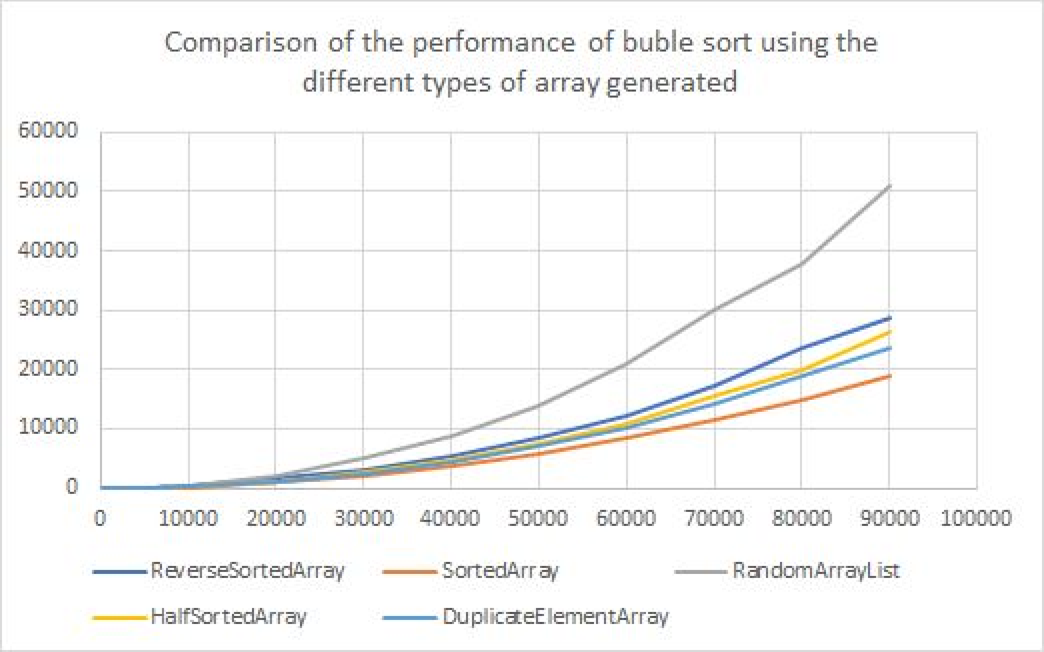
The graph below shows our result. The method was analyzed using the different combinations of arrays. Our hypothesis was that the graph should overlap because there is no best case and worst case in selection sort. This hypothesis was proven to be true based on our measurements.

4.5.2Insertion Sort



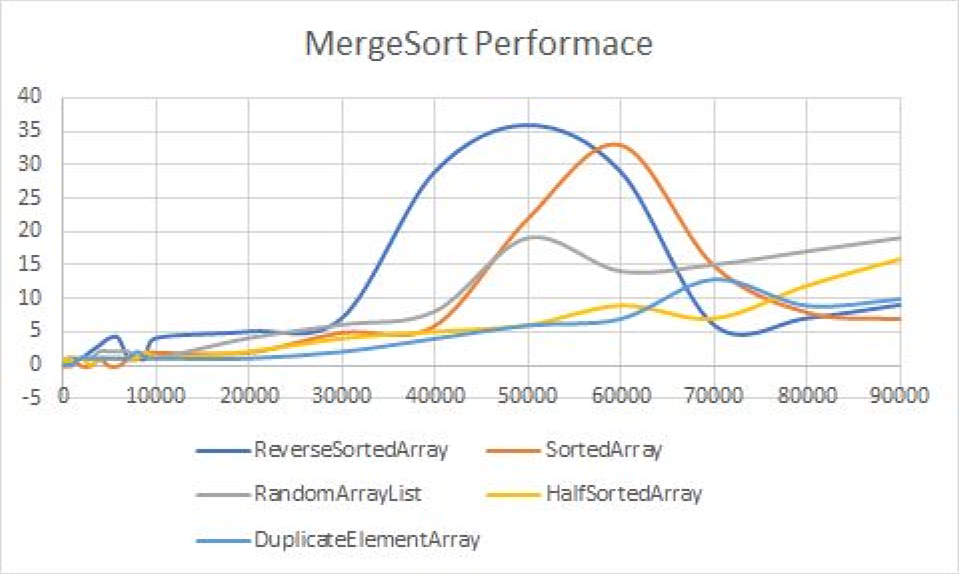
The comparison of each case of the insertion sort was realized. Above is the result that we discovered. Based on the result , we realized that insertion sort is extremely efficient in terms of sorting a sorted array. However, it is extremely slow in term sorting a reverse sorted array. Sorting a half sorted array and array with duplicates takes almost the same time. They can be considered as an average case in this situation.

4.5.3BubbleSort



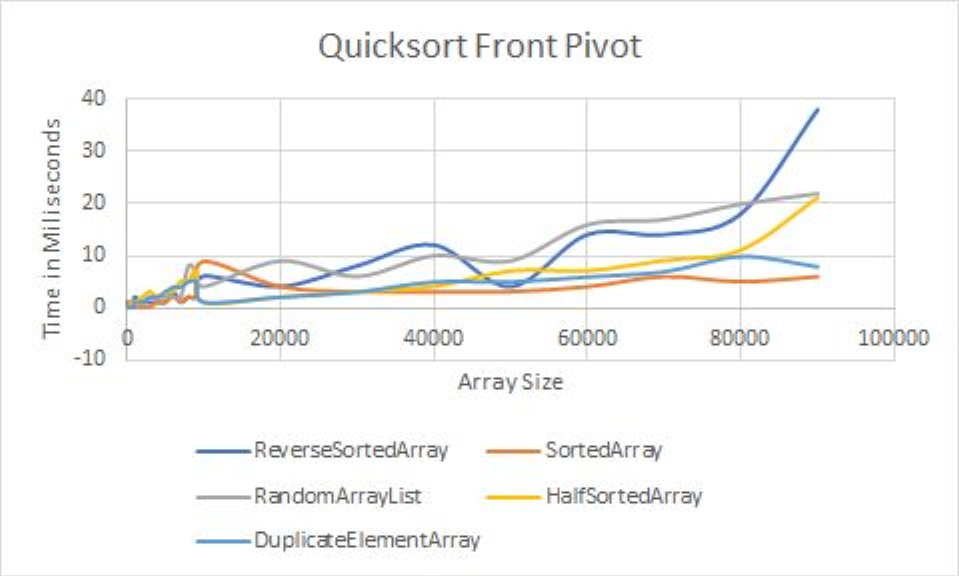
Based on the result above, the bubble sort algorithm performs slowly when it sorts a random list. The best case is when the list is sorted. All the other arrays are average cases.

4.5.4MergeSort



The algorithm looks a little unstable at first sight. It may have been caused by the fact that our computer was heating up when realizing the test. However, this data can tell us that merge sort takes a small amount of time to sort. The maximum time is about 35 ms for sorting 90000 elements. It can be still understood that reverse ordered arrays are the worst case,

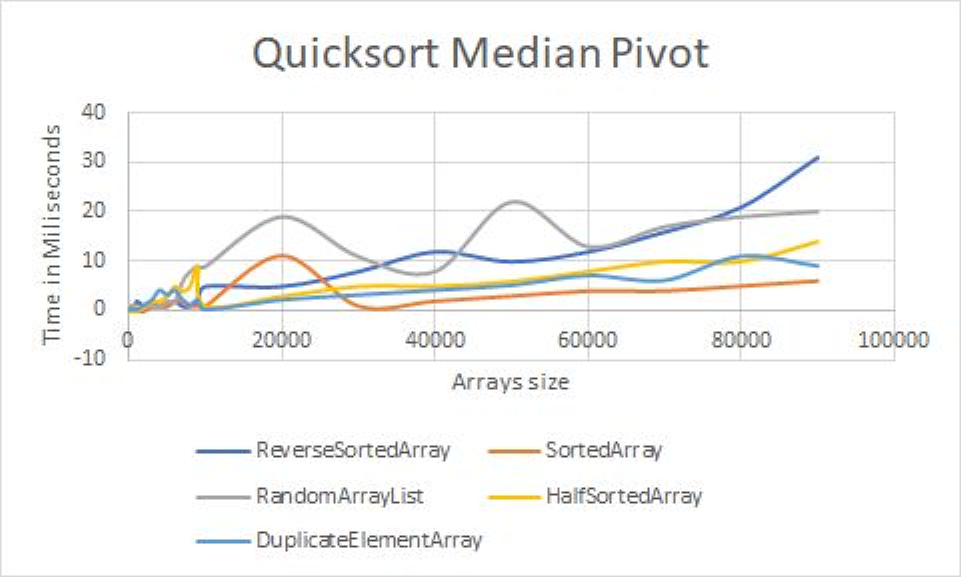
4.5.5Quicksort Front Pivot



The graph above shows that Quicksort with a front pivot has the best case with a sorted array, worst case with a reverse sorted array and average case with half sorted array, arrays with duplicate elements and random ordered arrays. These outcomes follow the theoretical expectation, except with the sorted array curve. Theoretically, having the pivot as the first element and sorting an array that is already sorted should cause a worst case, which is not

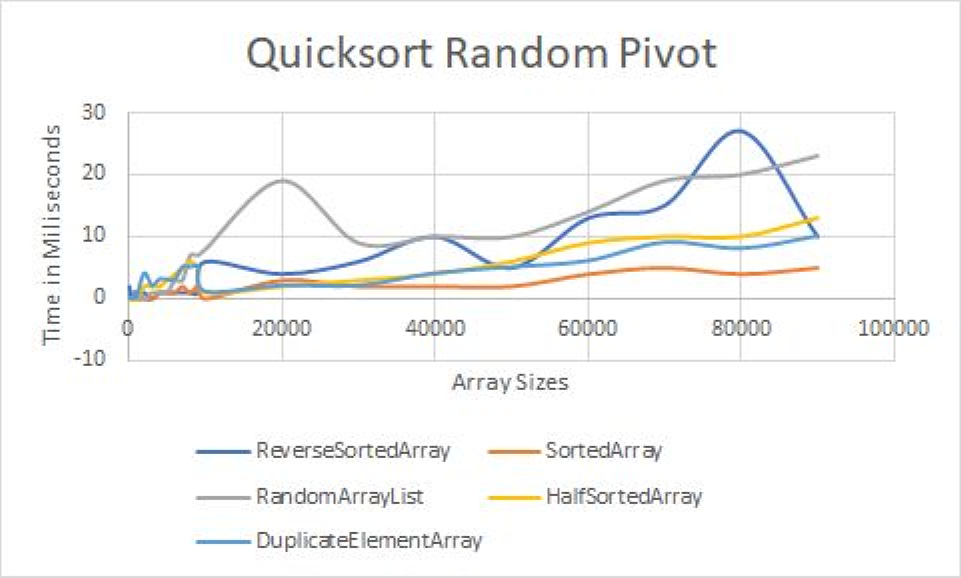
shown in the graph above.

4.5.6Quicksort Median Pivot



The graph above shows that Quicksort with a median pivot has the best case with a sorted array, worst case with a reverse sorted array and average case with half sorted array, arrays with duplicate elements and random ordered arrays. Neglecting some unfamiliar behavior from the random arrays curve and the sorted array curve, these outcomes follow the theoretical expectations.

4.5.7Quicksort Random Pivot



The graph above shows that Quicksort with a random pivot has the best case with a sorted array, worst case with a reverse sorted array and randomly ordered array and average case with half sorted array and arrays with duplicate elements. Theoretically, randomly ordered arrays should be an average case for the Quicksort with a random pivot. However, disregarding this and some of the unexpected behavior found in the reverse ordered array curve and the random ordered array curve, the other curves are drawn as expected.

5 Conclusion

Our different analysis in this project taught us that each sorting algorithm has different performance based on the total number of samples to be sorted and the order in which those samples are arranged. We discovered that some sorting alogrithms are better than others in some cases and the others are better in other cases. In theory merge sort and quicksort are the most efficient algorithms since both have O(nlogn). However, our measurements showed that merge sort can be even better than quick sort. The issue with quick sort is the choice of the pivot. Depending on the pivot, the program can be faster or slower. However, in practice the memory plays an important role because computers do not have infinite memory size. Taking that into account, we will exclude merge sort into our list of final best performer sorting algorthims. . After all the analysis, here are the best performers that we discovered during the analysis:

- Sorted Data: **Insertion Sort**
- ReverseSort : **Quicksort Median Pivot or Front Pivot**
- Random Array: **Quicksort Median Pivot**
- HalfSorted Array :**Quicksort Median Pivot**
- Duplicate Array :**Quicksort Median Pivot**

In our analysis as well, we attempted to compare our implementation with Java’s native sorting algorithm. The goal was to compare them in order to understand if they are related or similar to each other. We discovered that Java’s built in quick sort and merge sort is faster than our implementation. We suspect that these algorithms are enhanced versions of the current implementation and not exactly the same exact code.

Works Cited

Ge, Xial. "Lecture Notes 3: Sorting" Computer Science 150, 10 Oct. 2020, Lafayette College. Microsoft PowerPoint presentation.

Util. (July 14, 2020) August 29, 2020 from

https://docs.oracle.com/javase/8/docs/api/java/util/package-sumIllmary.html

Junit Assert. (July 14, 2020)) August 30, 2020 from

https://junit.org/junit4/javadoc/latest/org/junit/Assert.html

Junit Test. (July 14, 2020)) August 30, 2020 from

https://docs.oracle.com/javame/test-tools/javatest-441/html/junit.htm

io. (July 14, 2020) August 30, 2020 from

https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html