

# CS 150 – Project 1 Report

*Yazdan Basir & Jannat-ul-Ferdous*

*March 23rd, 2020*

---

## **Introduction:**

The goal of this project was to introduce us to the major, most commonly used sorting algorithms and their functionalities. The goal was to understand how and when these different sorting algorithms perform their best, worst, and average by testing them against different sized lists sorted to varying degrees – completely random and unsorted, partially sorted, completely sorted, and sorted in reverse order.

All the sorting algorithms that we tested and analyzed, along with a brief description of each, is as follows:

### **1) Bubble Sort:**

Compares two values at a time and swaps them if a larger value appears before a smaller value in the array. This ensures that at the end of each pass, the largest value is “bubbled” to the end and so, it does not need to be considered in the next pass. Bubble Sort takes  $O(n^2)$  and so it becomes more and more inefficient as the size of the array grows. Another downside of it is that even after everything in the array has been sorted, one more pass occurs at the end, which checks two elements at a time, one last time. This just adds on to the time taken by the number of elements.

### **2) Selection Sort:**

Looks for the smallest value in the array and places it at the start of the list. Once a value has been placed at the start of the array, it is not looked at again

in order to avoid inefficiencies. It essentially 'selects' the smallest value from the whole array. Like Bubble Sort, this algorithm also takes  $O(n^2)$  and becomes unusable as 'n' increases.. Again, like Bubble Sort, it has to go through the array once more at the end to ensure every value has been looked at. Selection Sort is also an unstable sorting algorithm which makes it less desirable to use.

### **3) Insertion Sort:**

Insertion sort starts looking at values in the start of the array and places each subsequent value in the correct place between the values it has already looked at. It essentially 'inserts' every new value into its correct place. Like both Bubble and Selection sort it has a runtime of  $O(n^2)$  and becomes extremely inefficient as the size of n increases, even though it is very efficient for smaller values of n.

### **4) Merge Sort:**

Divides the array into halves at the midway point. Sorts each half separately and uses recursion to do so. After both of the halves have been sorted, they are merged together and put into order to create the final sorted array. A downside of this is that a lot of extra memory is used when the new temporary array is created to hold half the values while the other values are being sorted.

### **5) QuickSort:**

Initially, QuickSort starts off like Merge Sort and aims to partition the whole array into 2 smaller parts. However, it doesn't do so based off of the middle value of the array and instead does some sorting before doing so. Using the pivot value, it places the smaller values into one half of the array and the larger values into the second half. After restoring the partition, the array is split according to it and the process repeats until the array has been partitioned into groups of 1 and 2 elements. From here, putting the smaller arrays back together in order starts.

For the purposes of this project, we used 3 variations of QuickSort – the first element is the pivot, the median of the first, middle, and last elements is the pivot, and a random value is the pivot. For the sake of simplicity, we will refer to these variations of QuickSort as QuickSort First, QuickSort Median, and QuickSort Random.

The runtime for all algorithms is in the table below:

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$

### Approach:

The test cases were designed to test the widest range of possibilities regarding the performance of each sorting algorithm. For this purpose, we decided to use 4 types of arrays to test our sorting algorithms:

1. Completely random and unsorted

2. Partially sorted
3. Completely sorted
4. Completely sorted in reverse order

The completely random and unsorted array was generated using a random number generator with the time of instantiation as the seed used. Values up to 5000 were allowed to avoid duplication of values which would have affected the test results to an extent.

The partially sorted array was generated using a range of values between a min and a max and by setting every other value to be a random value in that range to ensure some order of the elements.

The completely sorted array was generated for an 'n' number of elements by using a regular for loop and adding whatever value corresponded to 'i', starting from 0, to the array.

The same process was used for the array sorted in the reverse order but 'i' started from 'n' number of elements instead of 0 this time.

These different arrays brought out the advantages and disadvantages of each sorting algorithm.

Firstly due to the size of the small, medium, and large arrays used. The small arrays consisted of 100 to 5000 elements, the medium arrays consisted of 5000 – 100,000 elements, and the large arrays consisted of up to 100,000 – 10,000,000 elements. For QuickSort upto 10,000,000 elements were used whereas for Merge Sort only upto 1,000,000 elements were used to prevent the Stack Overflow error from showing up.

As the picture shows, Bubble Sort, Selection Sort, and Insertion Sort have  $O(n^2)$  runtimes. Due to this, these algorithms were not tested for large arrays, as it would be inefficient, near impossible to run, and hard to extract information at that point. So, these algorithms were just used on small and medium arrays. Merge Sort and all 3 versions of QuickSort were tested on small, medium, and large arrays.

The limitation of sorting only random, unsorted arrays is that there is too much fluctuation. Since every random array is completely different from each other, we don't really get anything other than a general sense of how fast each sorting algorithm can sort.

Although it is very important to test how fast these algorithms are in cases of random arrays, it is more important to find out – using the partially sorted array as well – which conditions are ideal for each algorithm. This is because in most practical situations in life where there is a need for sorting items using any of our algorithms, the data that needs to be sorted will not be completely random or unsorted. These scenarios involve sorting data that is already partially sorted or data that is in some existing order that needs some modification rather than complete rearrangement. This is where the need for partially sorted arrays can be seen.

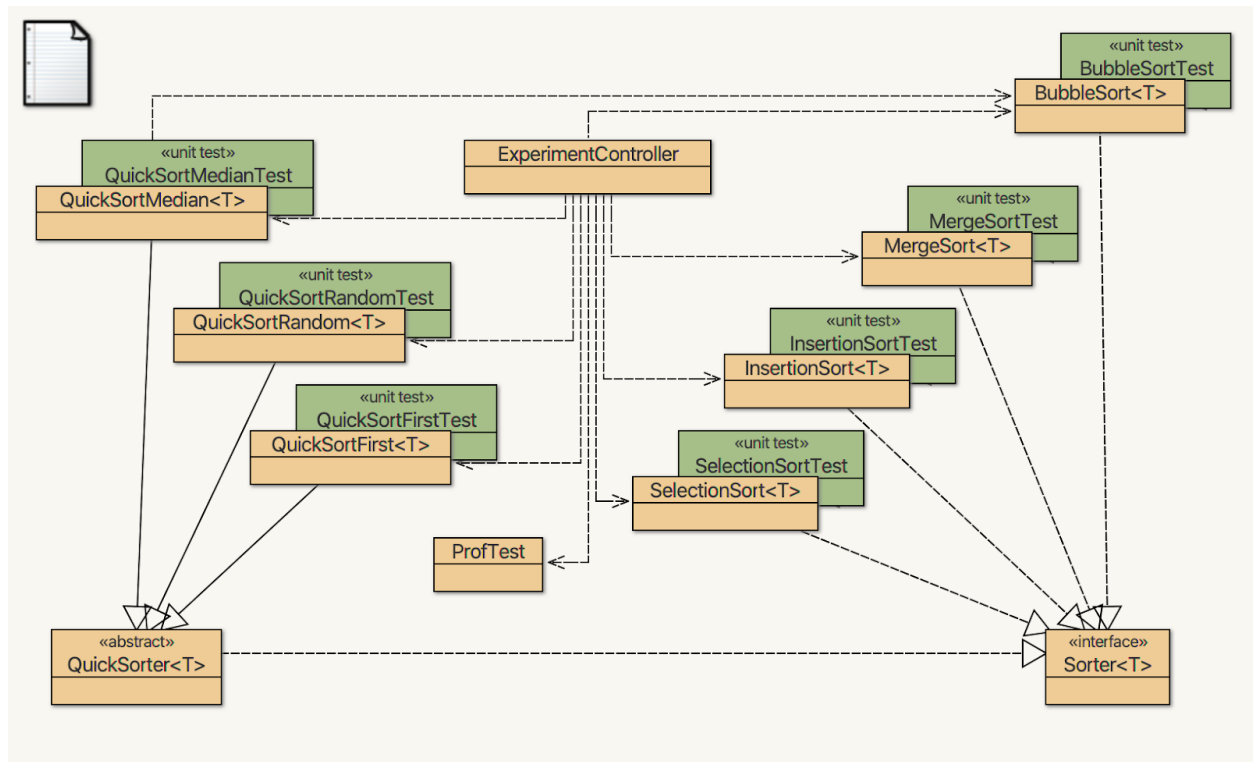
The completely sorted array was intended as a test to see how many extra/unnecessary passes each algorithm makes in such cases and how much time is added on to the total when this happens. For an extra dimension to the three aforementioned arrays, the reverse order array was presented as the worst possible case when it comes to sorting and it allowed us to test the runtime of each sorting algorithm in an absolute sense.

---

## **Classes:**

The Project has a *Sorter* Interface with a *sorting* method. All the sorting algorithms are written in a different class, with each class extending the *Sorter* interface and overriding the *sorting* method based on how the specific algorithm works. For *QuickSort* algorithm, we have an abstract *QuickSorter* Class and the different variations of the *QuickSort* inherits this abstract Class. Three versions of *QuickSort* algorithm, based on the partition position, are written in three *QuickSort* Classes (*QuickSortFirst*, *QuickSortMedian*, *QuickSortRandom*).

Finally, we have an *ExperimentController* Class that generates the different types of arrays for testing purposes, calls all the different sorting algorithms and calculates their run-time to be collected for comparison data. Each Class also has a Test Class associated with it for unit testing purposes.



## Methods:

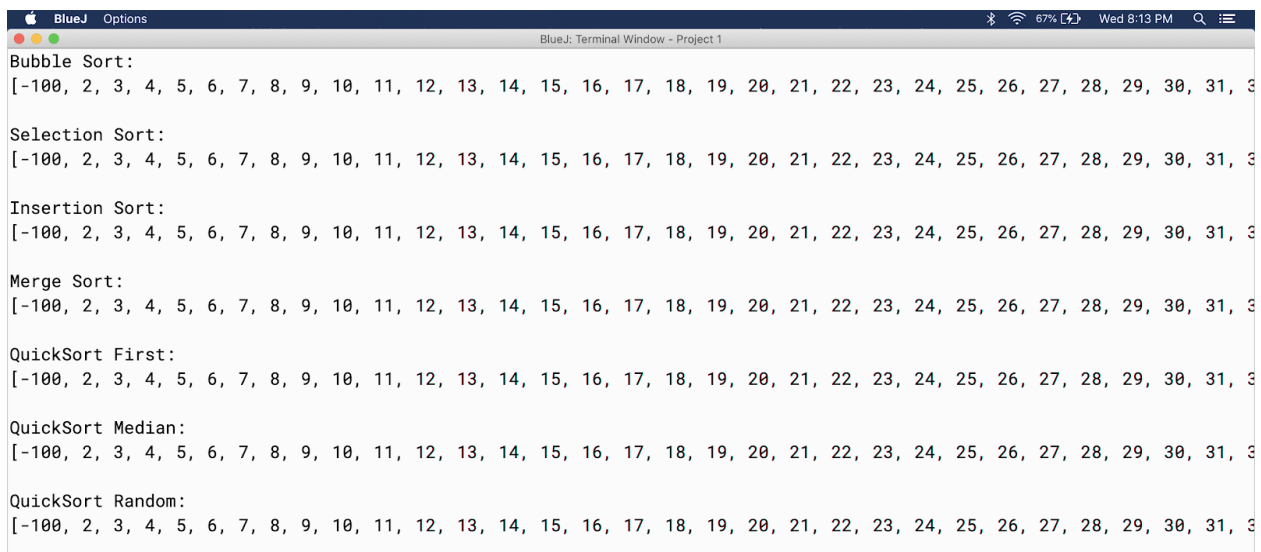
All of the data collection and testing took place across 2 MacBook Pros. The algorithms were tested and the data was collected only after all other programs had been closed on the machines and a 10–15 minute rest was given to the processor to avoid using unnecessary memory that would've been available to BlueJ. The fluctuation and inconsistencies in the data was perceived to be a problem with the program just starting up and not performing efficiently. Several tests were run to see if the fluctuations could be ironed out, but they were an inherent part of the small and medium arrays.

After a few trial runs and after checking whether each algorithm was working, the final test runs were made and the data was saved into .txt files straight from the BlueJ output window.

---

## Data Analysis:

We were provided a test array over which we had to run our algorithms and showcase the preliminary results during class. As we were unable to do so because one of our members wasn't present, we have provided the output of all our sorting algorithms on the test array (as much of the sorted array that could fit into one screenshot). A new instance of the test array was created before every run to ensure the algorithms were not sorting an already sorted array. It is as follows:



```
BlueJ Options
BlueJ: Terminal Window - Project 1

Bubble Sort:
[-100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

Selection Sort:
[-100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

Insertion Sort:
[-100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

Merge Sort:
[-100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

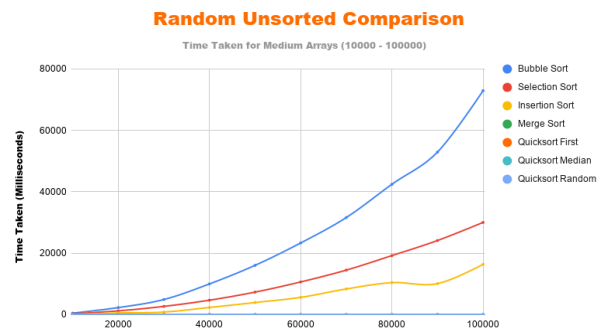
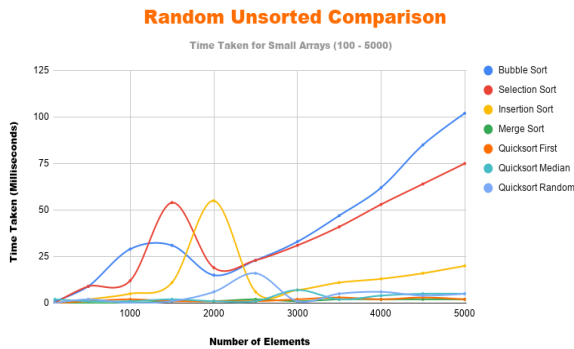
QuickSort First:
[-100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

QuickSort Median:
[-100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

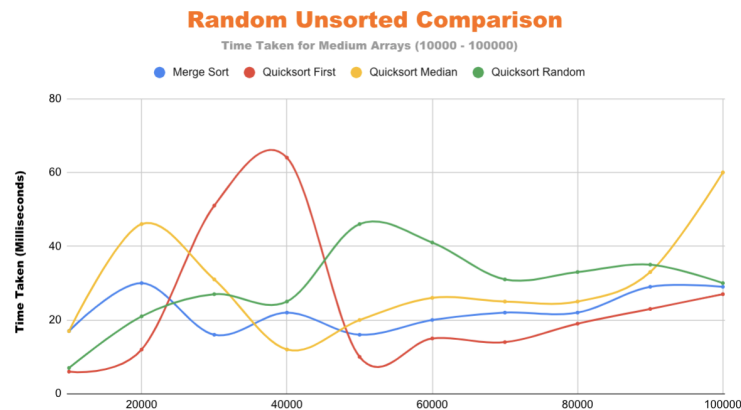
QuickSort Random:
[-100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
```

Before general data analysis of each algorithm, we should take a look at how all the algorithms perform when it comes to a certain type of array.

### 1) Random Unsorted Arrays:



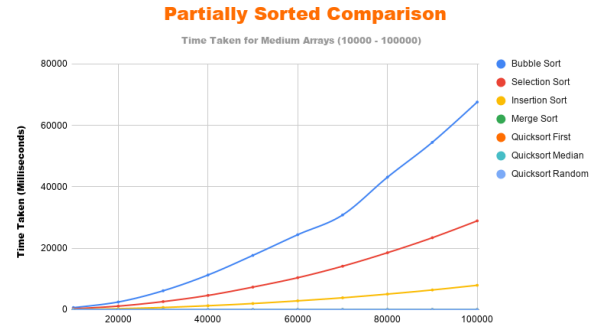
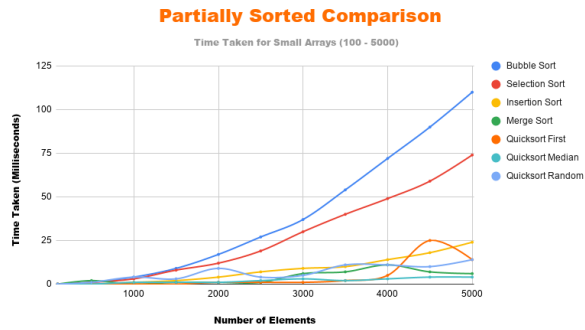
The data lets us conclude that, in the case of both small and medium arrays, Bubble Sort generally tends to perform the worst. The graph for the small arrays shows us that Merge Sort performed the best out of all the algorithms, whereas this conclusion isn't very clear from the graph of medium arrays.



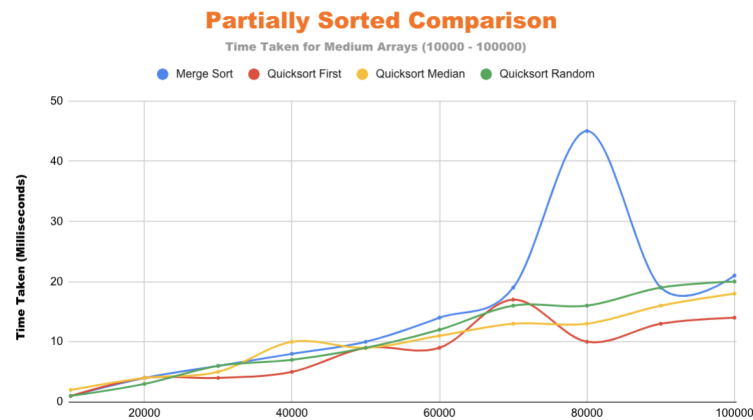
Removing Bubble, Selection, and Insertion Sort from the graph shows us that, despite no clear best performer, QuickSort First (with the first element as the pivot) generally tends to perform the best for medium-sized random unsorted arrays even though it has some fluctuations at the start.

## 2) Partially Sorted Arrays:



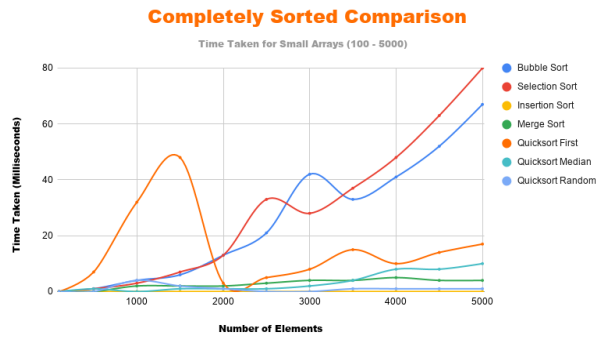


The data here shows again that for both small and medium partially sorted arrays, Bubble Sort tends to perform the worst. For the small arrays, we can notice how it is QuickSort with the first element as the pivot that performs the best generally with the median pivot QuickSort performing even better as 'n' gets larger.



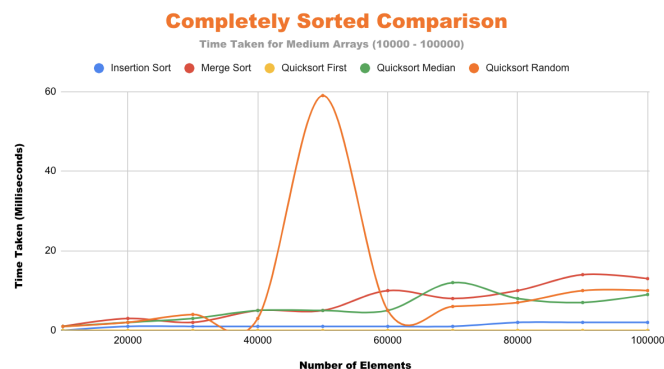
For the medium-sized partially sorted arrays, concluding the best algorithm is difficult unless Bubble, Selection, and Insertion Sort are removed from the graph again. From the new graph above, we can now see that it is indeed Quicksort First which generally performs the best once again.

### 3) Completely Sorted Arrays:



For completely sorted small and medium arrays, it is extremely close between Bubble Sort and Selection Sort for the worst performance. For small arrays, there is a tie at the start between the two, where QuickSort First seems to be performing the worst. However, as 'n' gets larger, it is Selection Sort that generally performs the worst here. The best performer in this case is Insertion Sort, with near-zero numbers all around.

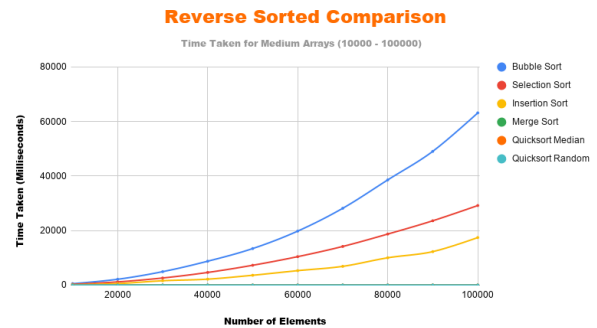
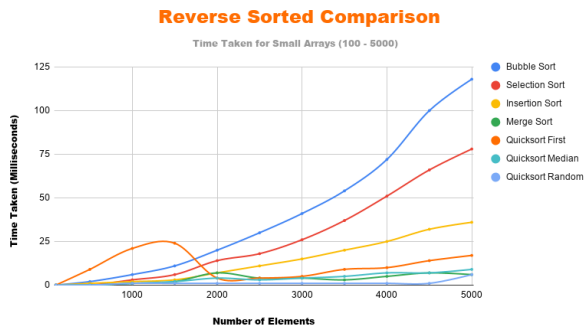
For medium-sized arrays, again, at the start, it is close between Bubble and Selection Sort, but the difference gets clearer as 'n' gets larger and Bubble Sort emerges as the worst performer. The graph of QuickSort First could not be included as a Stack Overflow error kept occurring. On the other hand, to get an idea of the best performer in this case, we need to remove Bubble and Selection Sort.



This lets us see that, once again, Insertion Sort is the outright best performing algorithm with near-zero times once again. This shows that Insertion Sort spends the least amount of time making unnecessary comparisons and passes through the array if it is already sorted, making it every efficient for

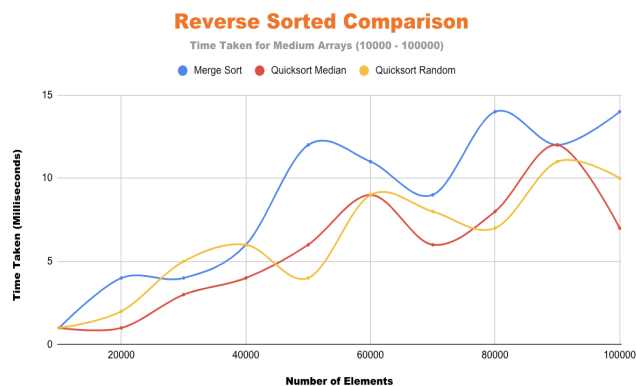
cases where the order of data needs to be checked and ensured rather than it being necessarily sorted.

#### 4) Reverse Sorted Arrays:



In the case of small and medium-sized reverse sorted arrays, we can see that Bubble Sort is the clear worst choice and takes more time to sort than any other sorting algorithm. The reverse sorted algorithm was designed to be used as the worst-case test and it was expected that Bubble Sort would take the most time in this case because it compares every two elements to each other in order to make a swap.

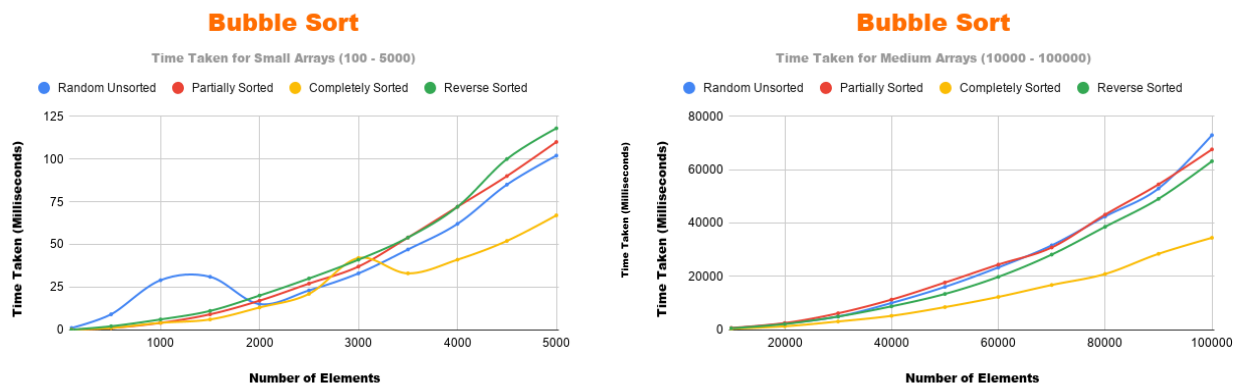
The best performer over the small reverse sorted array seems to be QuickSort Random for the most part. For medium-sized arrays, Bubble, Selection, and Insertion Sort have to be removed again to see the best performer.



From this new graph, there is no clear best performer and there seems to be a tie between QuickSort Median and QuickSort Random. QuickSort First could not be graphed because of a Stack Overflow error here.

Now, we can look at the individual performance of each algorithm.

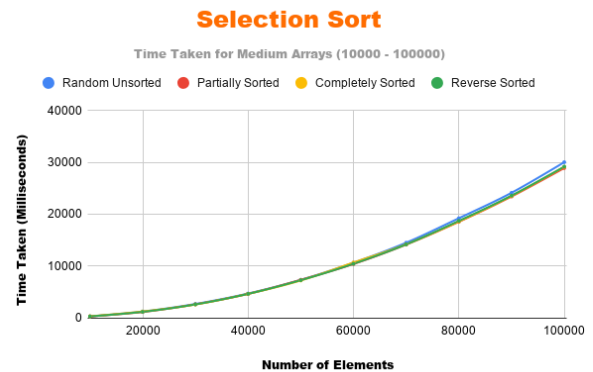
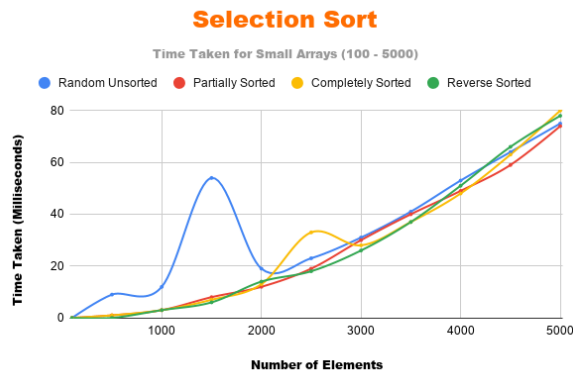
## 1) Bubble Sort:



Bubble Sort was tested on small and medium arrays. While we can see that there was some fluctuation in the time taken for smaller arrays, that generally flattens out with the medium arrays. In both cases, it is clear, as was expected, that Bubble Sort spent the least amount of time on the completely sorted arrays. At the same time, however, the line for completely sorted arrays is quite close to and similar to the rest of the arrays, which is a worrying sign for Bubble Sort and highlights its general inefficiency across the board.

From the rest of the data, we can conclude that, despite the initial fluctuation, Bubble Sort performed its best on a completely random and unsorted small array and on the reverse order sorted medium array. Between both sizes of arrays and all 4 types of arrays, Bubble Sort generally performs the same.

## 2) Selection Sort:

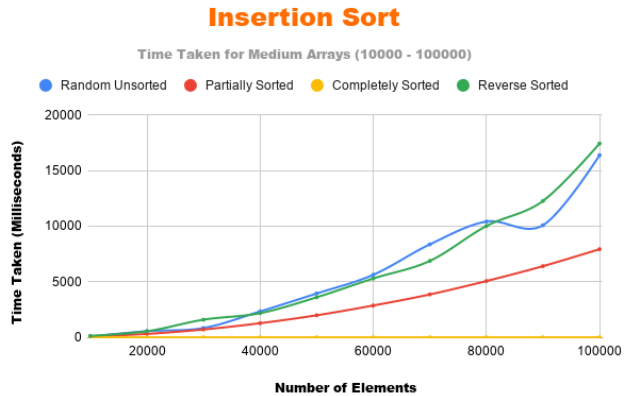
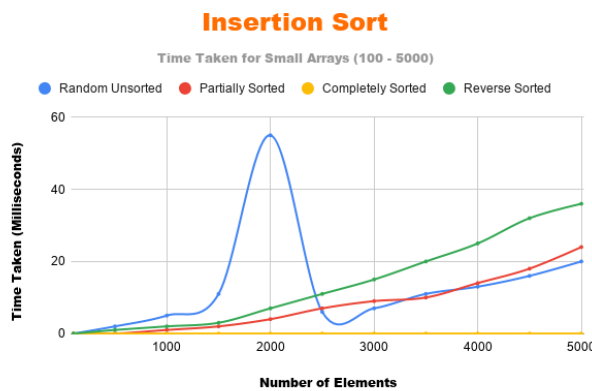


Selection sort was also tested on small and medium sized arrays. Similar fluctuations for the initial values of the small arrays can be observed here as well. There is no clear array for which Selection Sort performs its best as the data overlaps to a large degree. However, for the sake of choosing one type of array which has a better time performance over small arrays, we can select the reverse order array.

The medium sized arrays paint a similar and inconclusive, but more telling, picture. While there is no fluctuation this time, we can see that Selection Sort performs almost exactly the same over the 4 types of medium sized arrays.

The performance is so similar that the graph of the partially sorted array and completed sorted array cannot even be seen from the picture above. And just like Bubble Sort, it is a worrying sign for the performance of Selection Sort in that it takes the same time to sort completely sorted arrays as it does partially sorted or reverse sorted arrays.

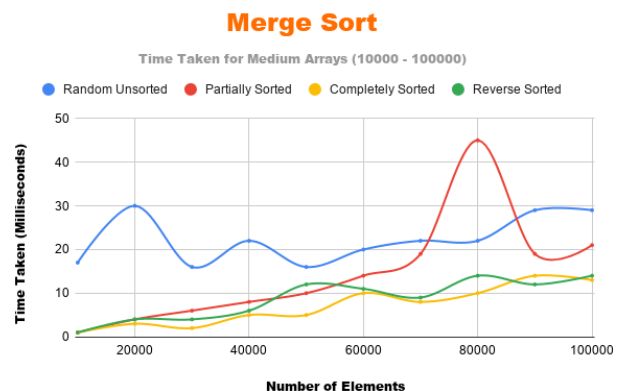
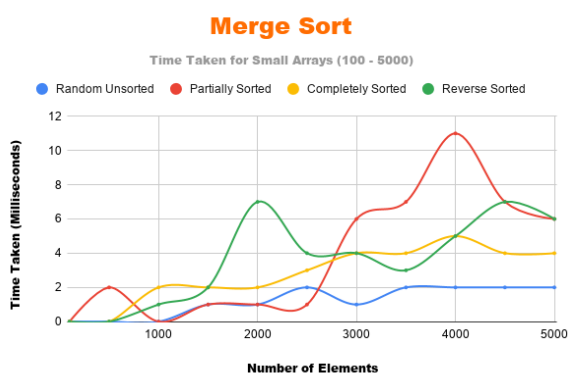
### 3) Insertion Sort:



Tested over the same sized arrays as Bubble and Selection Sort, Insertion Sort seems to provide better performance and more refreshing results. Even though it has the same fluctuations at the start, it takes significantly less time than Bubble or Selection Sort for all the arrays. Unlike those two algorithms, Insertion Sort is also extremely efficient when dealing with already sorted arrays, taking near-zero time to deal with it. This is a huge change and advantage to consider when using Insertion Sort.

The same conclusions can be seen for medium-sized arrays as Insertion Sort seems to perform the worst on reverse sorted arrays, as expected, and has near-zero runtime for already sorted arrays.

#### 4) Merge Sort:

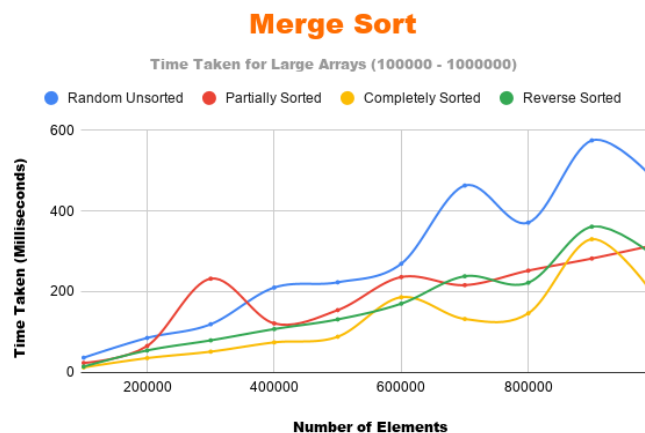


The runtime for Merge Sort over small and medium-sized arrays is full of fluctuations and inconsistencies as the two graphs above show. Although it is

very unclear to draw a winner as the best performer over small arrays, we can clearly see that Merge Sort performs its worst on random, unsorted arrays.

For medium-sized arrays, it's slightly more intuitive to draw such conclusions. Merge Sort seems to worst on random, unsorted arrays whereas it performs its best, unsurprisingly, on completely sorted arrays.

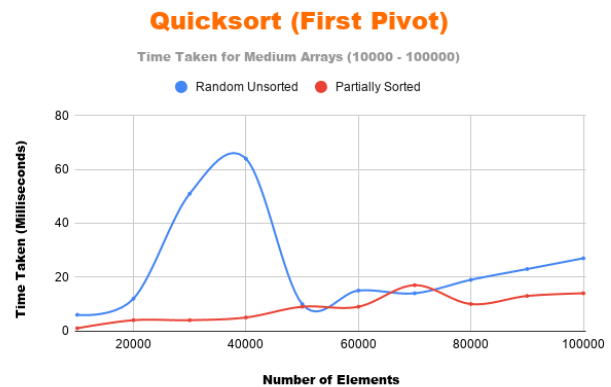
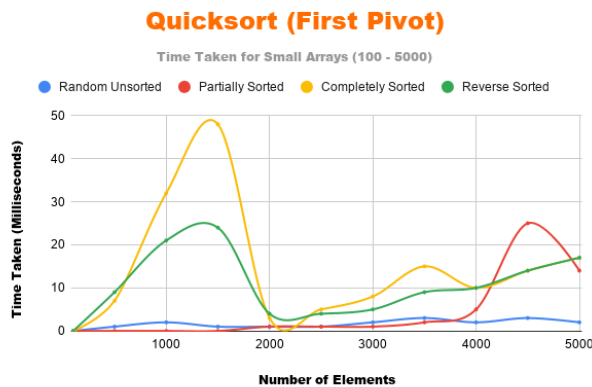
Now that it is possible to extend the data set and include large arrays, we can test the algorithm over much larger values of 'n' and draw more definitive conclusions. The graph below shows exactly that.



From here, we can observe consistent patterns in the time taken for Merge Sort over the different types of arrays. The data above shows us that Merge Sort performs its worst for random unsorted arrays, like it did in the data above for medium-sized arrays, and performs its best on completely sorted arrays, again just like the medium-sized arrays.

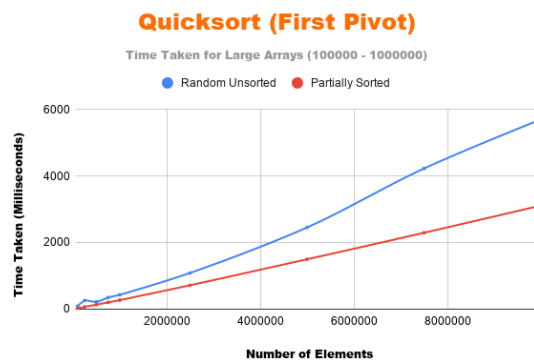
#### 4) QuickSort:

QuickSort was tested in 3 different variations – one where the first element was the pivot, one where the median of the first, middle, and last elements was the pivot, and one where a random element in the array was the pivot. All 3 variations were tested over small, medium, and large arrays.



The graphs for QuickSort First over small and medium-sized arrays show that this algorithm, surprisingly, performs its worst on completely sorted arrays and generally tends to perform its best on random, unsorted arrays and partially sorted arrays to a slightly lesser extent.

The data for medium arrays is a bit limited in this case because the algorithm could not be tested for completely sorted and reverse sorted arrays due to a Stack Overflow error. However, from the limited dataset, we can see that QuickSort performs better on partially sorted arrays and has consistent runtimes as compared to random, unsorted arrays.

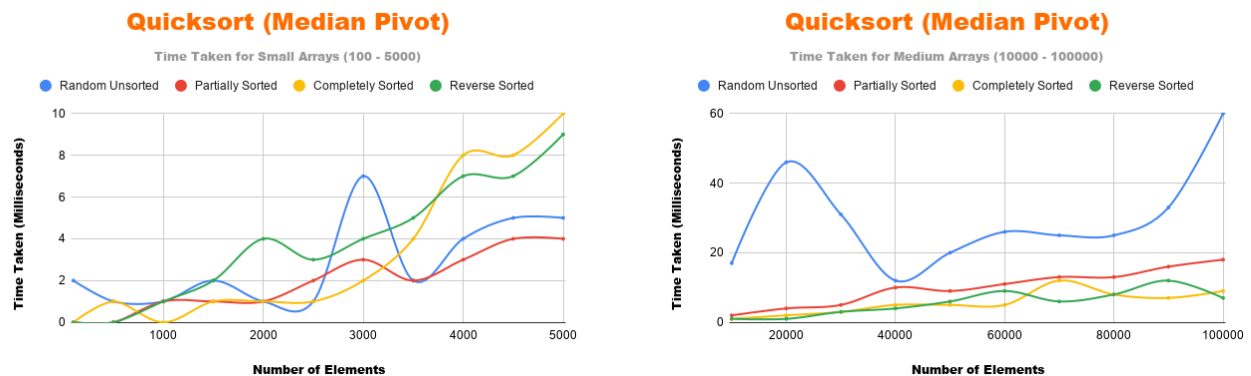


Testing QuickSort First over large arrays brought out the most consistent results. Although the data is still limited because the algorithm could not be tested over completely sorted and reverse sorted arrays due to the same error, there are still some conclusions that can be drawn.

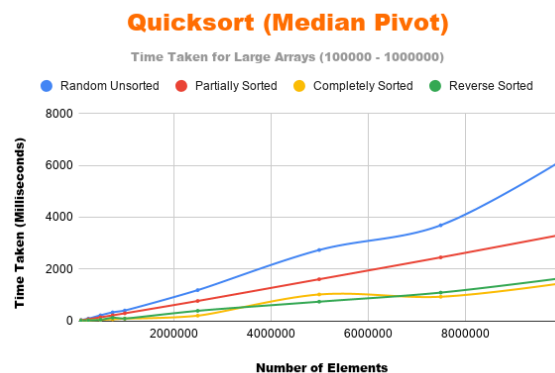


We can now observe that there is a significant difference between the performance over random unsorted arrays and partially sorted, with the latter being the best performance case.

Next, we can take a look at QuickSort Median.



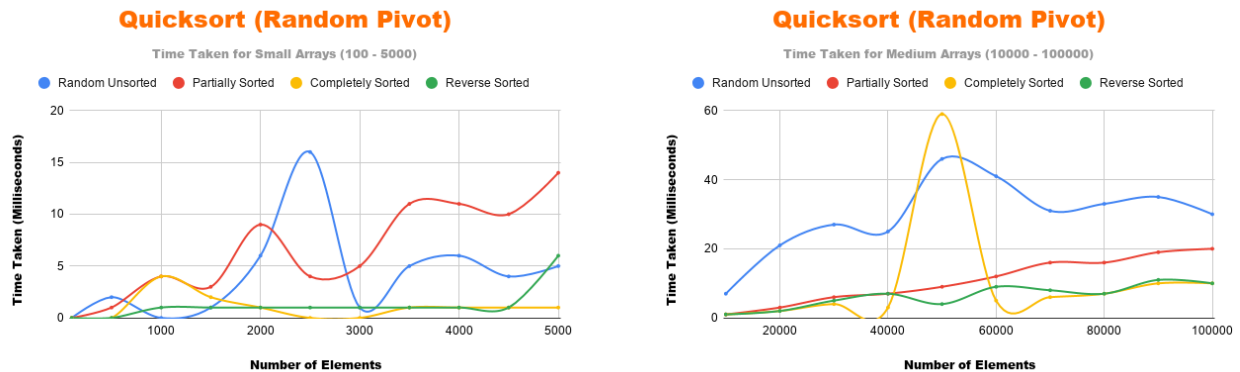
The data shows us that, over small arrays, QuickSort Median's runtime is all over the place and not consistent at all. There is no clear best performer or worst performer that can be concluded from this dataset. The runtimes become much more consistent as we move into medium arrays. Here, we can see that QuickSort Median performs its worst on random unsorted arrays. However, unsurprisingly and surprisingly respectively, QuickSort Median performs its best on completely sorted and reverse sorted arrays.



The most consistent data consists of the runtimes over large arrays. From here, we can see that, like for medium arrays, QuickSort Median performs its

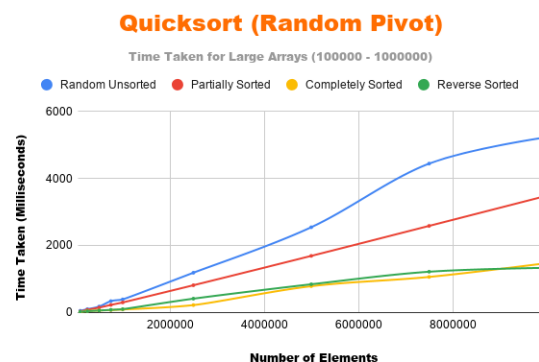
worst on random unsorted arrays. On the flipside, the algorithm performs its best evenly across completely sorted and reverse sorted arrays.

Finally, we can analyze the last variant of QuickSort – QuickSort Random.



Despite the massive fluctuations going on for the algorithm over small arrays, it is consistently the worst for partially sorted arrays and performs the best over completely sorted arrays. The runtimes for completely sorted medium arrays is also pretty consistent except for one big fluctuation at 50000 elements. In this case, QuickSort Random performs the worst for random unsorted arrays and performs its best over completely sorted and reverse sorted arrays.

The data becomes more clear and much more consistent over large arrays. From here, it can be concluded that QuickSort Random performs its worst on random unsorted arrays again and performs its best on completely sorted and reverse sorted arrays like before.



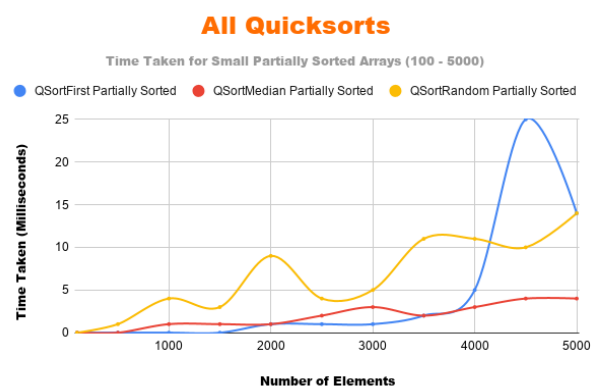
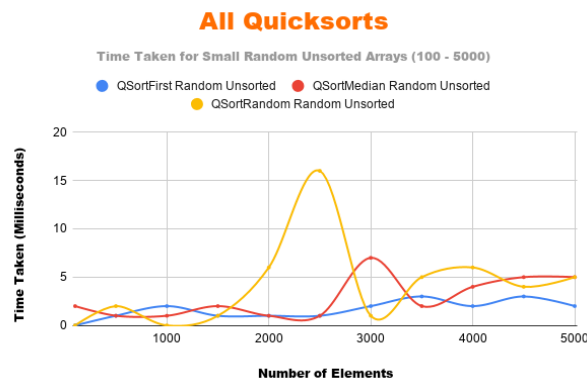
Despite inconsistencies in the data at the start, we can see that as the value of 'n' gets bigger and the number of elements increases, the algorithms' performances become more consistent and more defined. This allows us to more easily find out which algorithms are consistently performing better than others and for which type and size of array they are doing so. This will be continued further in the Conclusion section of the report.

## QuickSort Analysis:

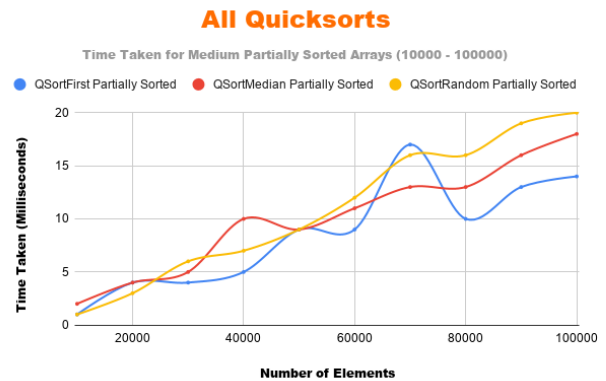
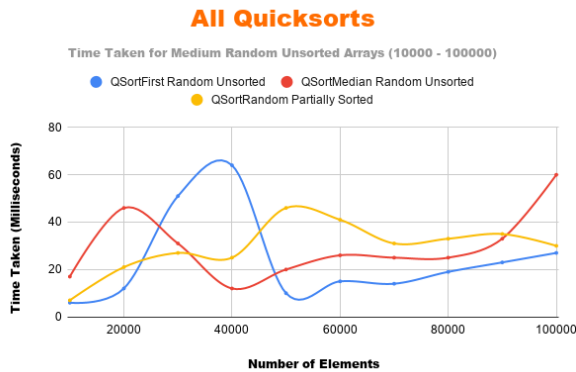
As 3 different types of QuickSorts were used, it is important to look at their performance differences and attempt to find out where each of them performs their best and worst.

For the purposes of comparing these 3 QuickSorts, all 3 sizes of arrays were used – small, medium, and large – but only as random unsorted and partially sorted arrays as it seems they will let us analyze the performance of each algorithm definitively.

The results are as follows:

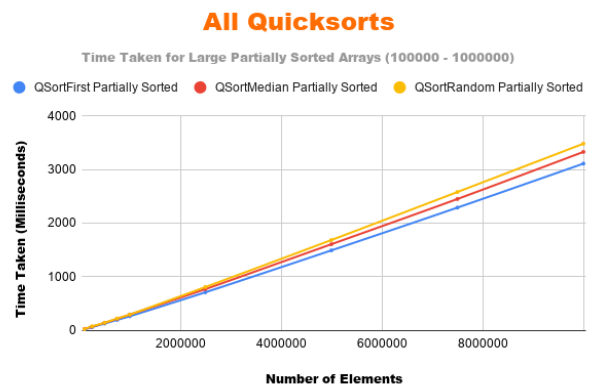
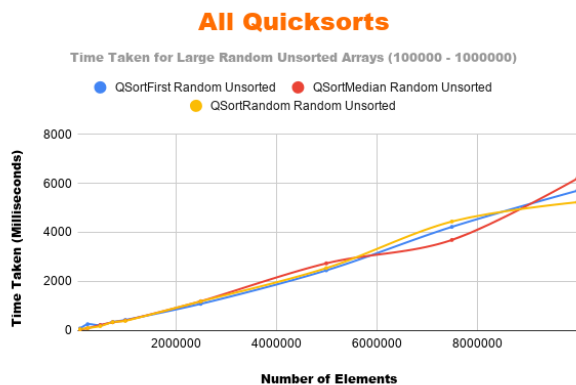


The results for the small random unsorted arrays are inconclusive as there is too much fluctuation and inconsistency to be able to draw a fair result. However, it does seem as if QuickSort First is performing the best in this case. For the small partially sorted array, we can see that the best performer is again QuickSort First until it hits a huge fluctuation after 4000 elements. QuickSort Median became the best performer in that period then.



The results for the medium sized arrays are slightly more consistent. For the random unsorted arrays, QuickSort Random seems to be the worst performer for the most part and after the initial inconsistencies, QuickSort First seems to be the best performer.

For the partially sorted arrays, QuickSort Random is again the worst performer and QuickSort First is the best performer despite some fluctuations clustered in one place.



The results for the large arrays are extremely consistent but they muddle the conclusivity to an extent as the runtimes are more similar than ever now. From the random unsorted arrays, there is no clear winner or loser because the graphs overlap many times and are very close to each other overall.

From the partially sorted arrays, despite the runtimes being similar, we can conclude that QuickSort First is once again the best performer and QuickSort Random is the worst performer.

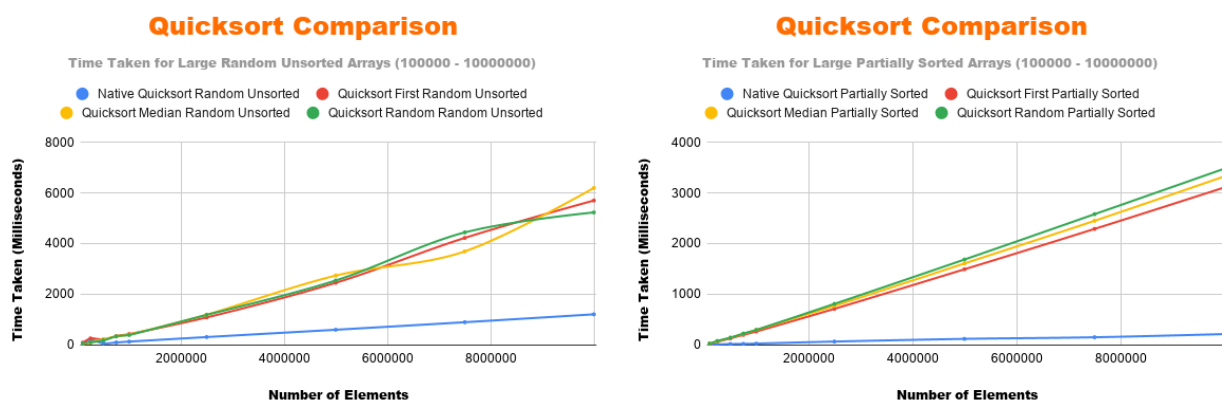
The 6 graphs show us that out of the 3 variants of QuickSort, it is QuickSort First that generally performs the best – quite fitting of its name. QuickSort Median, again like its name, just lies in the middle of First and Random, not outshining them outright in any department. The most consistent worst performer in each of these results is QuickSort Random.

### QuickSort vs Java's Native QuickSort:

Java has a built-in QuickSort algorithm that can be called upon primitive data types. Simple research allowed us to find out that for an array of type 'int', Java will use its version of QuickSort First when the .sort() method is called on that array.

Using the same randomly generated test data as before, but after casting the test array to type 'int', allowed us to test Java's native QuickSort and compare it to the versions we had implemented in our code.

The results are as follows:



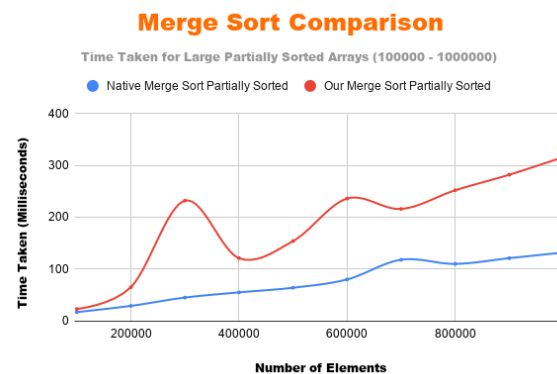
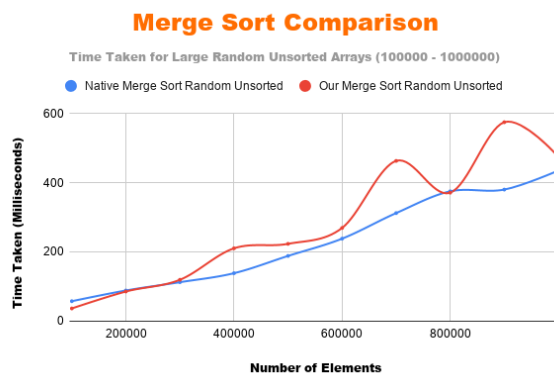
These tests were conducted over large random unsorted and partially sorted arrays because this is where the data is the most consistent and most clear.

The runtime data is quite shocking in this case. It can be seen that Java's native implementation of QuickSort is vastly superior to any implementation of our QuickSort. Where our variants of QuickSort take thousands of milliseconds, Java's QuickSort only takes a few hundred milliseconds and less to do so. This means that our implementation of QuickSort, while it does work perfectly and is pretty fast as compared to the rest of our algorithms, is nowhere near its maximum possible performance and efficiency.

## Merge Sort vs Java's Native Merge Sort:

A similar test was conducted to compare our implementation of Merge Sort to Java's native implementation of the algorithm. Our research allowed us to find out that Java automatically calls Merge Sort on reference data types and by casting our test arrays to type 'Integer', we were able to call upon Java's Merge Sort whenever we used `.sort()` now.

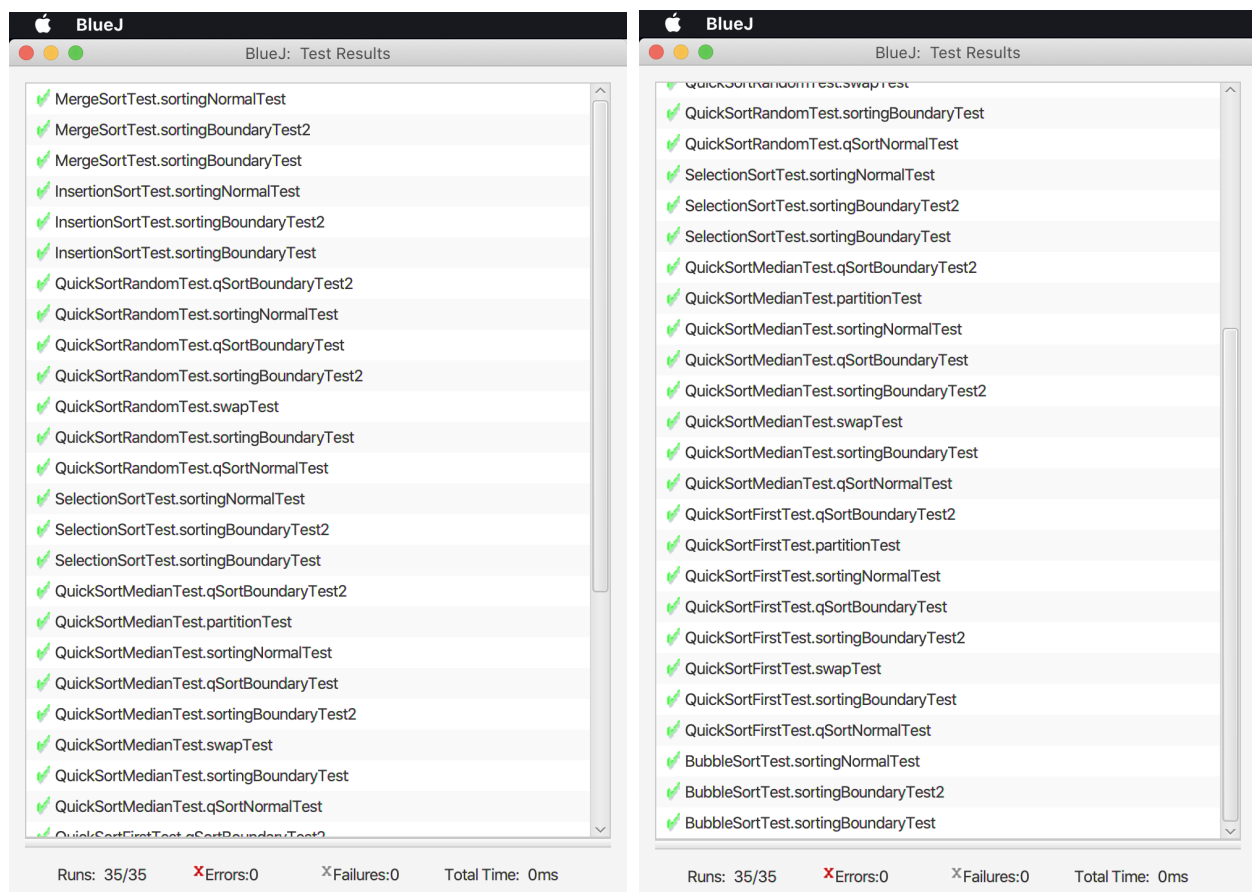
Similar to the tests for QuickSort, only large arrays were used to ensure the maximum possible consistency and clarity when it came to the data.



While the difference between the performance of our algorithm and Java's native implementation isn't as drastic here as it was for QuickSort, once again Java's algorithm trumps our own. We can see, from the two graphs above, that for both large random unsorted arrays and large partially sorted arrays, Java's Merge Sort constantly performs better than our implementation. A bigger difference can be seen in the partially sorted algorithms, where Java's Merge Sort performs much, much better than our Merge Sort.

Like the QuickSort analysis, this clearly shows that Java's implementation of Merge Sort is a much more efficient and faster algorithm that consistently performs in any given scenario over the types of arrays we're testing for.

## Unit Tests:



The Unit Tests were conducted to check every aspect of each sorting algorithm in order to definitively conclude that they work properly not only as a whole, but as a sum of their parts too. For this reason, each method in each class was unit tested as such.

The algorithms were tested by passing unsorted arrays through them and seeing whether they were producing the expected sorted output by the end. These algorithms were also tested by passing already sorted arrays through them to check if they could detect whether any changes needed to be made or not and whether or not they messed up anything in an already sorted array.

Helper methods such as partition, swap, and merge were also tested individually to ensure they were running properly.

As explained in the code already, through the comments and Java Docs, QuickSort Random could not be unit tested for its partition value because that value changed each time as the pivot was randomly selected from the elements of the array being sorted. And, the methods in QuickSort, such as partition and swap, have exact duplicates of the methods with 'Return' in their name at the end for the purposes of unit testing.

This was done to avoid making the original methods public and risking interference from other classes or methods.

---

## **Troubleshooting:**

A Stack Overflow error was encountered when testing QuickSort First over completely sorted and reverse sorted medium and large arrays. While it is understandable that this affects the test data to an extent and influences the conclusions we are able to draw from the data, nothing could essentially be done in this case. These particular tests were run on a few different laptops and computers just to see if anything was different but to no avail.

The Do While loop was changed to a more simple While loop which allowed us to test QuickSort First for a few more elements but that wasn't an increase that would've made the data more complete or inconclusive.



Hence, it has not been included in the graphs above and the respective analysis has been done keeping this gap in the data in mind.

---

## **Conclusion:**

The data collected and visualized in the graphs above show us that different algorithms shine through in different settings and for different purposes. That general conclusion validates the introduction of this report, where we laid out that one of the goals of this Project was to figure out where and when should we use a specific algorithm and how it performs when put through different tests and sizes of arrays.

After having done significant analysis of the data and reaching conclusions, wherever possible, for each type and size of array, we have derived the best algorithms that can be used for different types of arrays. Since the concept of sorting revolves around which algorithm is the fastest and most efficient, we have put special emphasis on the best performers. The results are as follows:

### **Best performers:**

- 1) Random Unsorted:** QuickSort (First)
- 2) Partially Sorted:** QuickSort (First)
- 3) Completely Sorted:** Insertion Sort
- 4) Reverse Sorted:** QuickSort (Median & Random)

From the results above we can see that, aside from Insertion Sort being the best for completely sorted arrays, it is a variant of QuickSort that performs the best in each case. In particular, it is QuickSort First that seems to be the standout performer in each case. While QuickSort First couldn't be analyzed for the Reverse Sorted arrays because of a Stack Overflow error, it is not outside the realm of reason to believe that QuickSort First would've been the best in this case as well.

All in all, as per our collected data and analysis, QuickSort First is the best particular sorting algorithm to use in a general, overarching sense for overall sorting and it is QuickSort, broadly speaking, that is the best algorithm overall.

The worst performer doesn't necessarily need to be picked out for each type or size of array because it was Bubble Sort, generally speaking, that performed the worst in each case and was the most unreliable and inefficient.

Alongside all this, one surprising result that was concluded was how much faster Java's native implementation of both Merge Sort and QuickSort was compared to our implementation. It showed that while our implementations were doing their job properly and were quite fast compared to the quadratic runtime algorithms, they were not at their maximum possible speed and efficiency level. Understanding how Java implements its algorithms will be key to understanding how to improve our own algorithms and get better results.

---

## **References:**

Xia, Ge. "Bubble Sort." *Lecture Notes 2 – Sorting*, 2020, [moodle.lafayette.edu/pluginfile.php/551806/mod\\_resource/content/1/sorting.pdf](https://moodle.lafayette.edu/pluginfile.php/551806/mod_resource/content/1/sorting.pdf).

Xia, Ge. "Selection Sort." *Lecture Notes 2 – Sorting*, 2020, [moodle.lafayette.edu/pluginfile.php/551806/mod\\_resource/content/1/sorting.pdf](https://moodle.lafayette.edu/pluginfile.php/551806/mod_resource/content/1/sorting.pdf).

Xia, Ge. "Insertion Sort." *Lecture Notes 2 – Sorting*, 2020, [moodle.lafayette.edu/pluginfile.php/551806/mod\\_resource/content/1/sorting.pdf](https://moodle.lafayette.edu/pluginfile.php/551806/mod_resource/content/1/sorting.pdf).

Xia, Ge. "Merge Sort." *Lecture Notes 2 – Sorting*, 2020, [moodle.lafayette.edu/pluginfile.php/551806/mod\\_resource/content/1/sorting.pdf](https://moodle.lafayette.edu/pluginfile.php/551806/mod_resource/content/1/sorting.pdf).

Xia, Ge. "Quick Sort." *Lecture Notes 2 – Sorting*, 2020, [moodle.lafayette.edu/pluginfile.php/551806/mod\\_resource/content/1/sorting.pdf](https://moodle.lafayette.edu/pluginfile.php/551806/mod_resource/content/1/sorting.pdf).

Xia, Ge. "Test Array." *Moodle*, 2020,  
[moodle.lafayette.edu/pluginfile.php/555860/mod\\_resource/content/1](https://moodle.lafayette.edu/pluginfile.php/555860/mod_resource/content/1).

Gupta, Anchit. "Arrays.sort() in Java." *GeeksforGeeks*, 7 Dec. 2018,  
[www.geeksforgeeks.org/arrays-sort-in-java-with-examples/](https://www.geeksforgeeks.org/arrays-sort-in-java-with-examples/).

"Sorting Algorithm Time Complexities." *GeeksforGeeks*, 8 May 2017,  
[www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/](https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/).

---