**Sorting Algorithms Experiment**

Final Report

Authors: Justin Jemison, Jackson Kettel, and Maximiliano Cifuentes

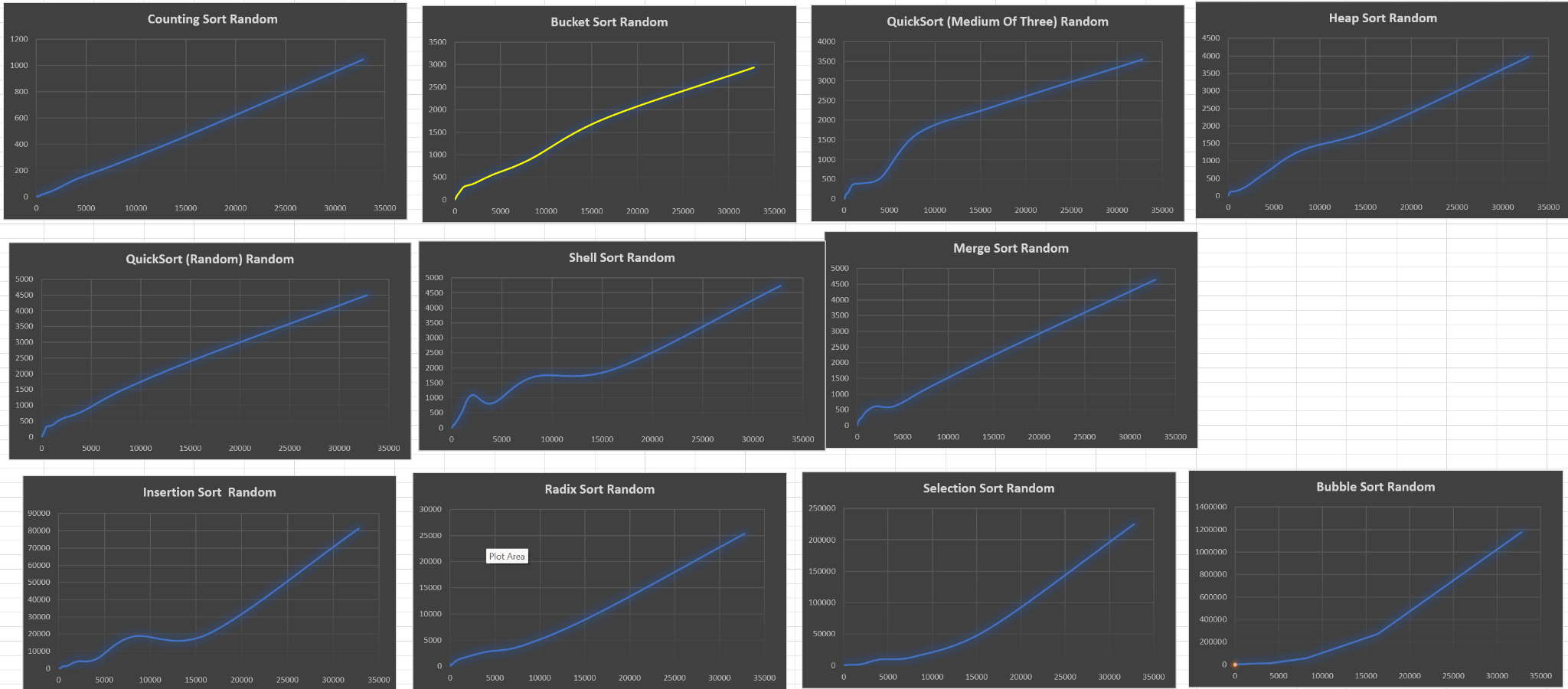
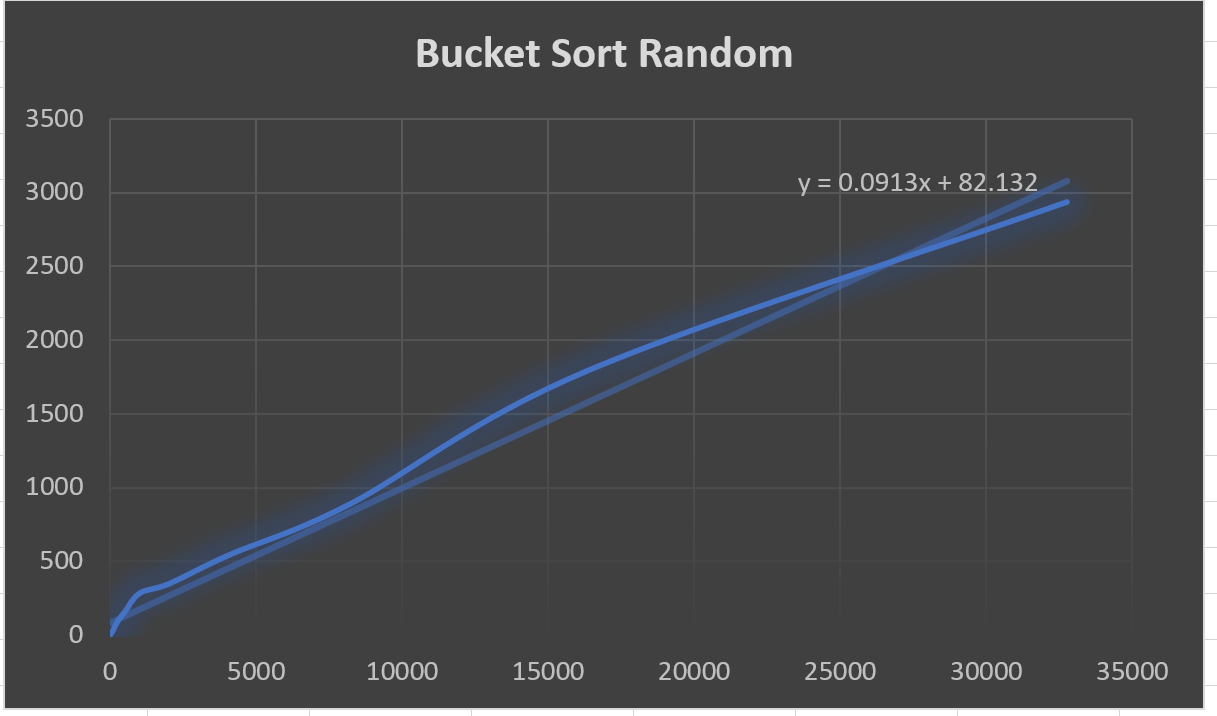
*Authors Note:* all graphs have array size on the x-axis, and runtime (Microseconds) on the y-axis

**Individual Analysis of Algorithms**

Analysis of Justin’ s Sorting Methods:

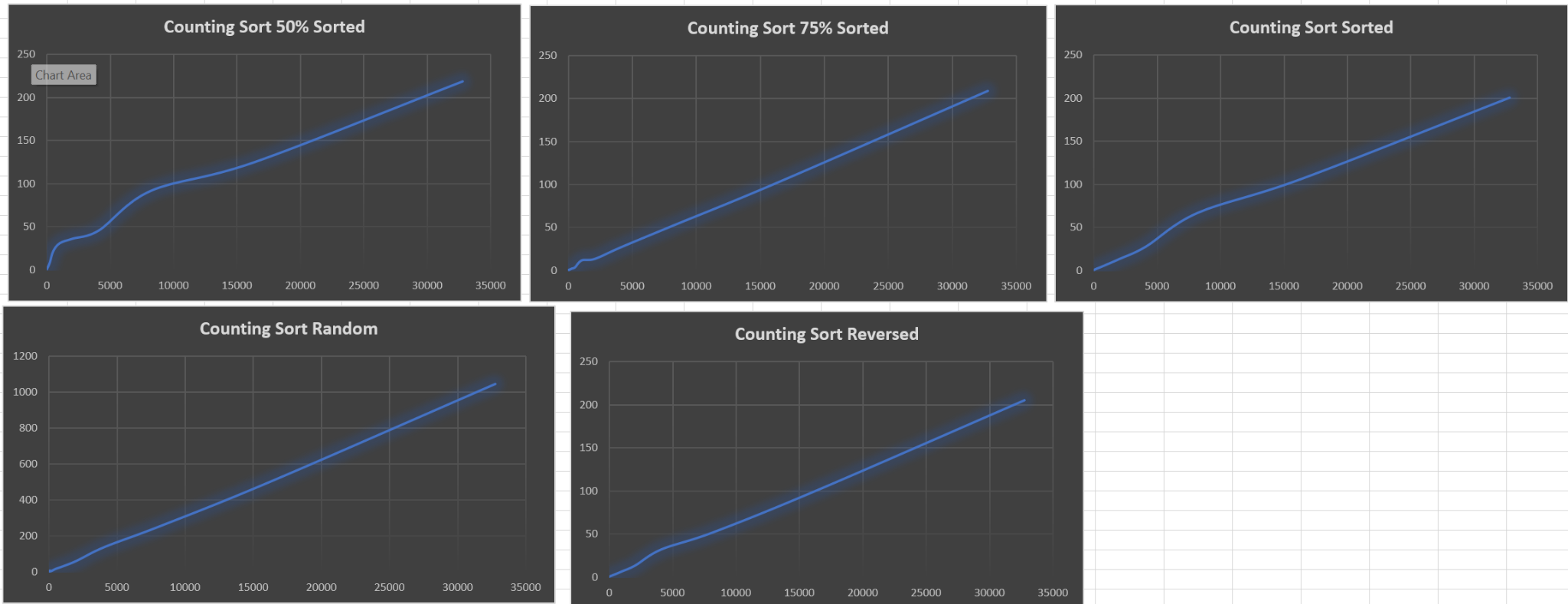
**Bucket Sort Algorithm:**

Bucket Sort showed a very linear progression through all its various sorting methods. As seen above, its weakest sorting method was when the array was 50% already sorted. Where it excels is when sorting a reverse array. Compared to other sorting methods Bucket Sort came in second for the random sort. This is consistent with the Big O(n) time that we see below on the first line. Memory used by bucket sort is O(n + m). I used single nodes to save each element costing O(n) + an array to hold all the elements so tha tagged size n equates to (n + m) size .

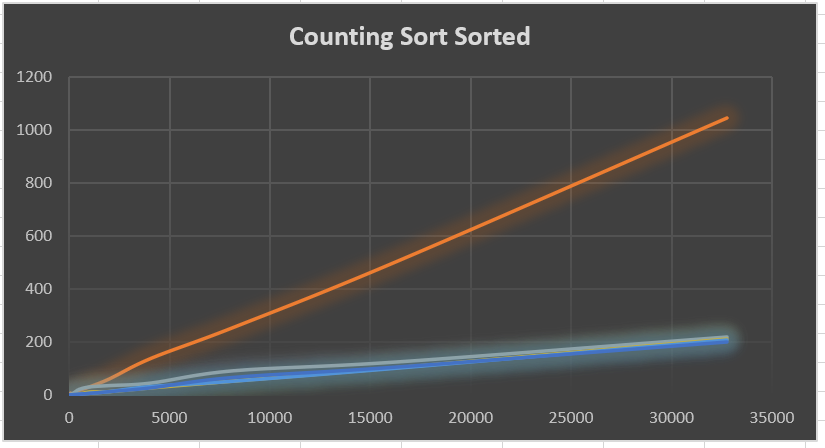
 

You can kind of see a curve compared to the linear prediction that excel provided. It makes me wonder if this type of sorting algorithm would end up being more of a (log n) time complexity if allowed to increase in array size.

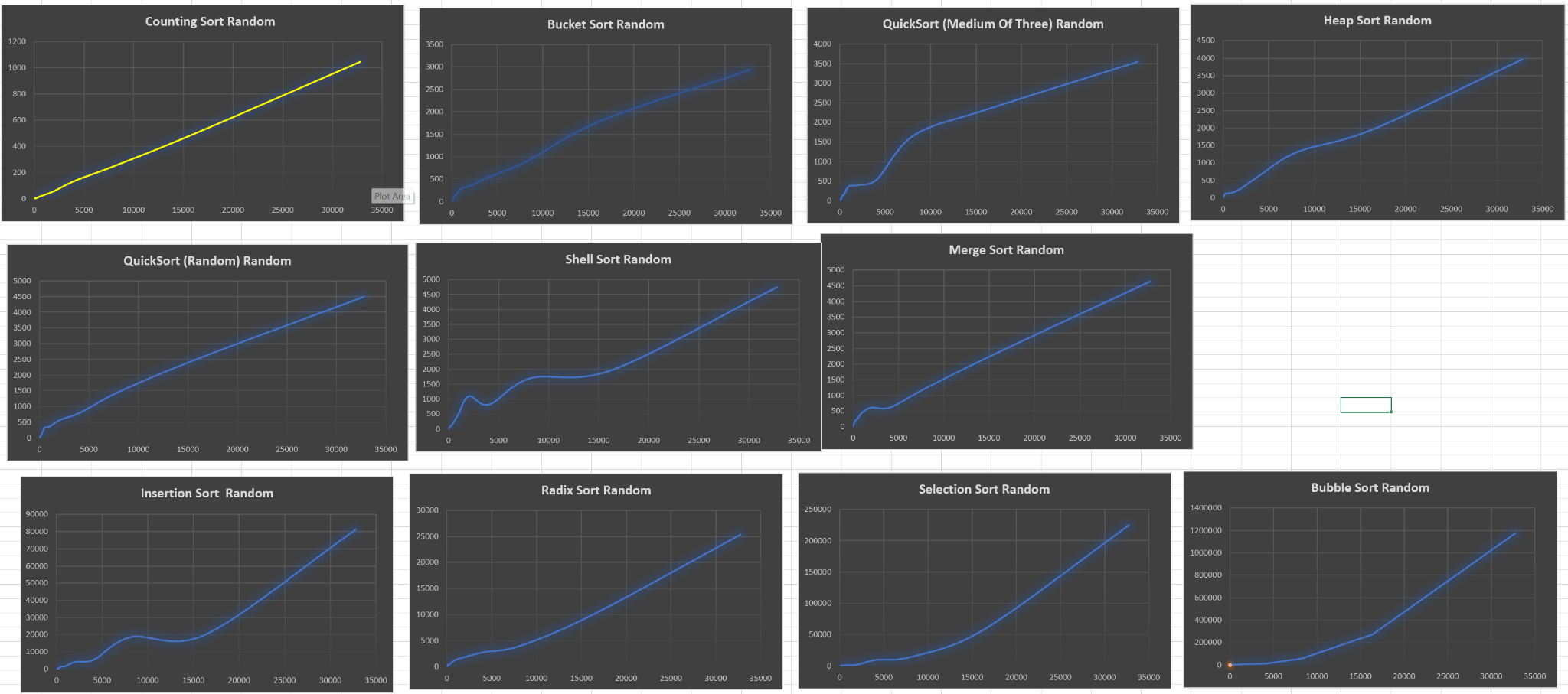
This type of sort is very good to use if you do not know if the array is sorted or if the array is unsorted, It does use a bit of memory so if you can spare it, it's well worth it.

**Counting Sort:**

Another O(n) time complexity algorithm is counting sort. As you can see, it is very fast for all types of arrays and the linear nature is evident in the fairly straight line. It was very consistent across the different types of arrays as seen above. Only time it took over 1000 millisecond was during the Random sort. I was very surprised by the success of this algorithm. According to our database structures book, this algorithm is really only used for clustering of numbers. Counting sort is excellent for high frequency of numbers. But in our standard test It performed above my expectations.



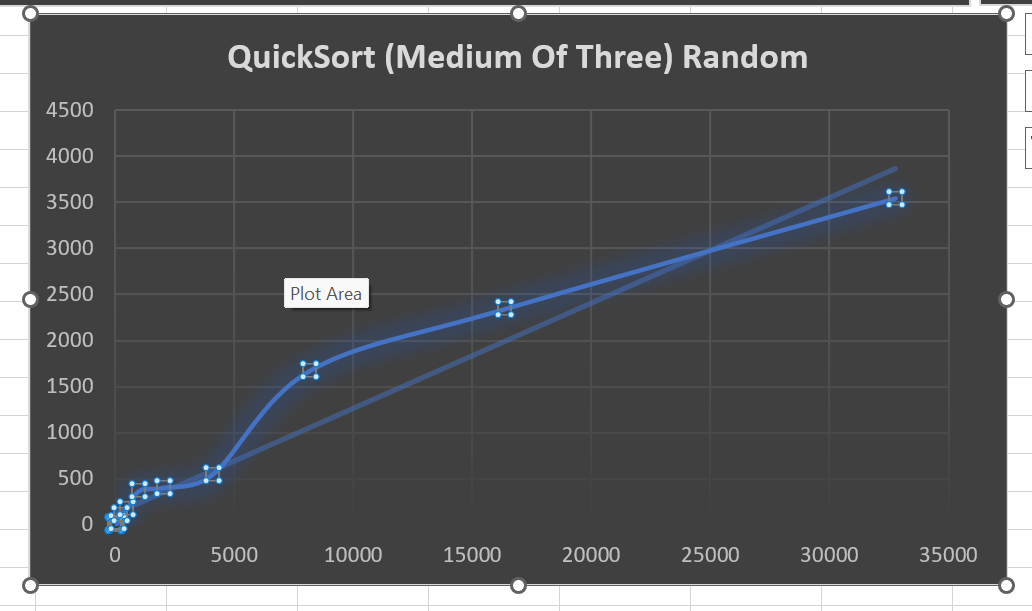
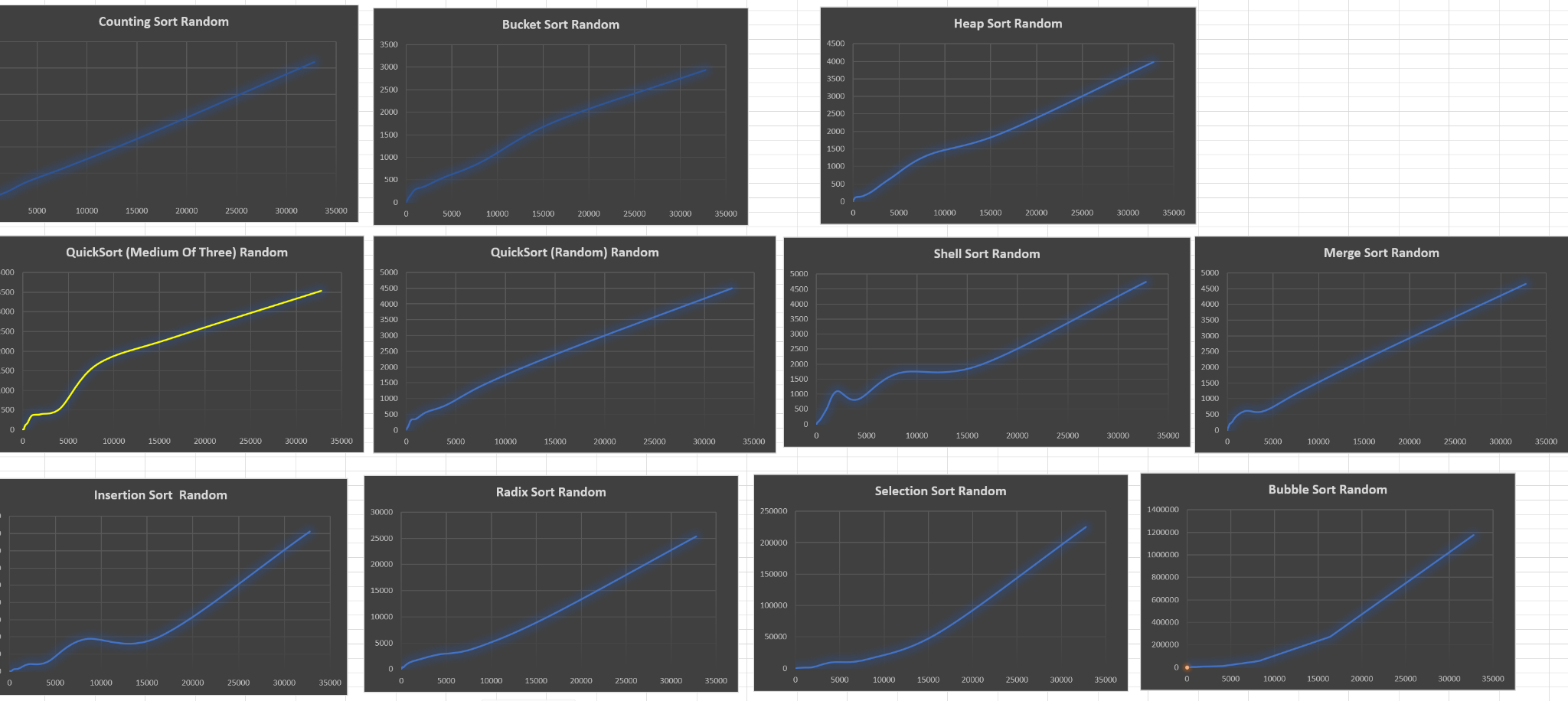
Superimposing the lines on top of each other you can see how consistent it is. This is due to the indexing nature of its algorithm.

The bucket sorting algorithm is the next closest algorithm but it was almost twice as fast as seen in the graphs below. This is kind of surprising due to the similarities of the counting sort to bucket sort. I wonder if it has to do with using linked lists instead of array indexing. 

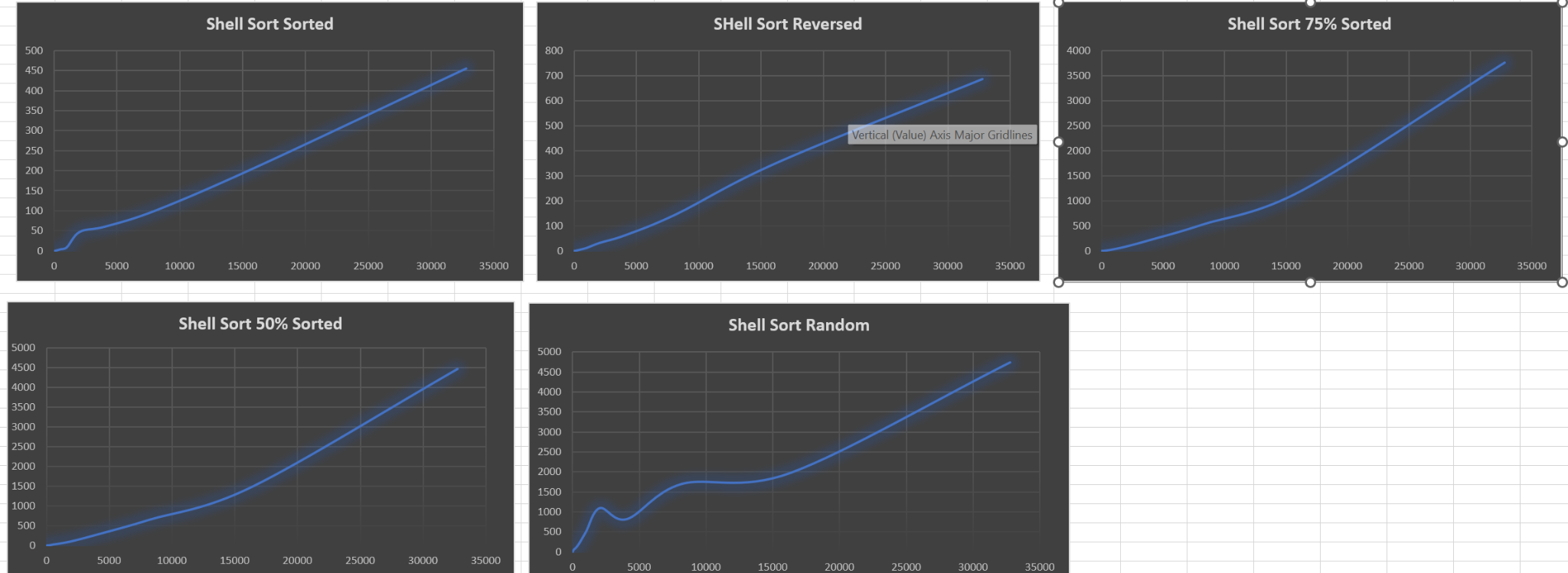
Compared to all other tests, counting sort was the fastest algorithm by far. There is one drawback with this algorithm which is the space it requires for running. It makes 3 copies of the arrays, using one for indexing, and a second for manipulation during the sorting phase and a third array to save the results( actually it copies the original array and then that array is used in altering the original array). In conclusion, if you have the space A counting sort will work well for all kinds of arrays.

**QuickSort (Medium of Three)**

The QuickSort Median of Three, was the faster of the QuickSort so far (Still waiting on the (First)) in any case

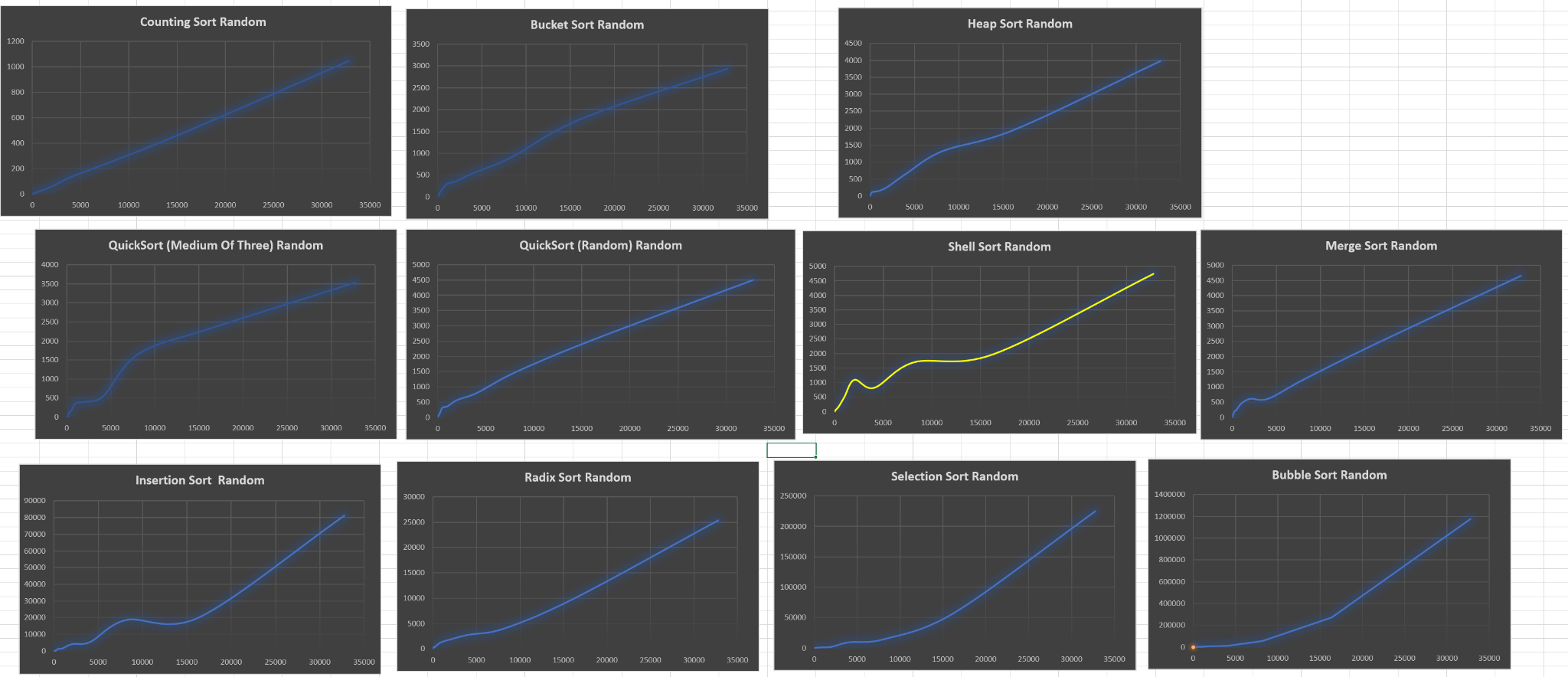
In any case you can start to see the (n log n) curve compared to a liner line as shown to the left. This quick sort will not sort anything below 10 elements in an array, instead it will use an insertion sort to sort the array. Compared to the rest of the sorting methods, it was in the middle of the pack but it was leading the recursive called algorithms, consistently beating merge sort and the other quick sort methods. It goes to show that using a valid pivot method will allow the quicksort to most likely avoid the worst time complexity of O(n^2). It is worth noting that our sort did not really have the shape of a (n log n) type algorithm except for when the array was random. Suspect that it is more linear in nature due to most of the work being completed in linear time compared to multiple levels of recursion. On top of that, each iteration, it would linearly sort the first, last and middle number. Then when getting to at or below 10 indexes, it would just conduct an insertion sort. In any case, 

Overall this is a faster sorting method than merge sort with the same space complexity. But if space is an issue, then I would turn to shell sort which does better with mostly sorted arrays and does not take up extra space.

**Shell Sort:**

Shell sort is the better cousin of insertion sort by iterating through the arrays at intervals. It was created by Donald Shell to improve the insertion sort method[1]. As observed above, the random is a vast improvement over the insertion sort and it only has O(n) space complexity, conducting the sorting within the array. This sort is O(n^2) at its worst, which we did not see. It is interesting that the High scoring numbers for the shell sort were at the beginning of the random sort. Maybe that was a couple of bad cases that caused our graph to be skewed and started us down the path of n^2 to n^1.5 time complexity. In any case it flattened out.

One should note the amount of improvement from 75% sorted (blue) to reversed sorted (gray) to sorted (orange), this algorithm is very good when an array is mostly sorted, only getting better the more sorted it gets. As suspected by Donald Shell. 

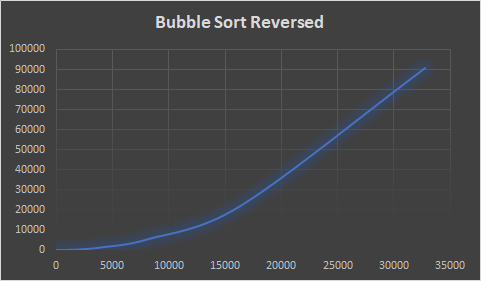
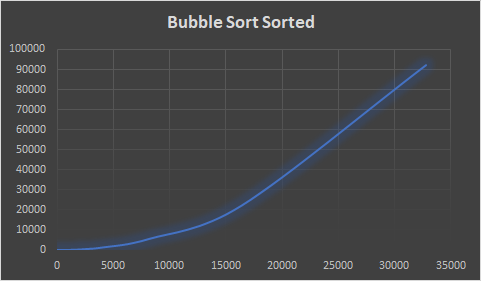
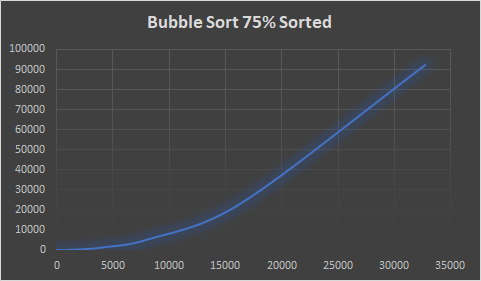
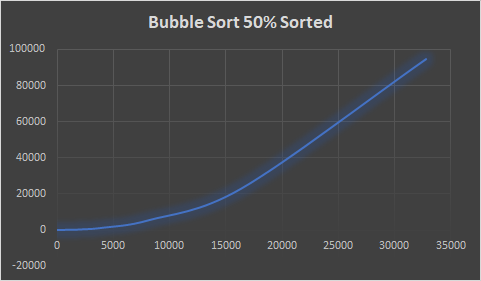
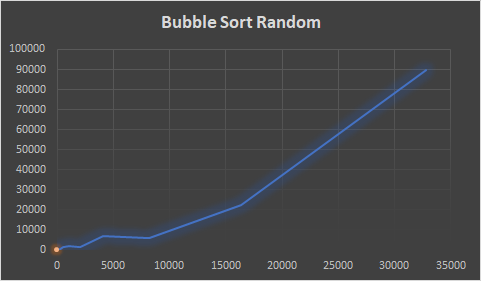
Shell Sort was middle of the pack when compared to the other algorithms which is very impressive since it does not use any extra memory to hold multiple arrays. I would highly recommend this type of sorting algorithm for when space is an issue. Or when you have a mostly sorted array, the time difference between Counting sort (500 millisecond) and QuickSort first index (90 milliseconds) was fairly close. Again the others are faster, but add the cost of memory or risking a stack overload. 

**In conclusion,** There is a big difference between using more memory and using more time to sort an algorithm. But the faster methods are turning out to be the more clever methods that rely on indexing. The results with linked list and the bucket sort kind of show that when able to avoid the creation of classes to store simple primitive arrays.

Analysis of Jackson’s Sorting Algorithms:

**Bubble Sort:**

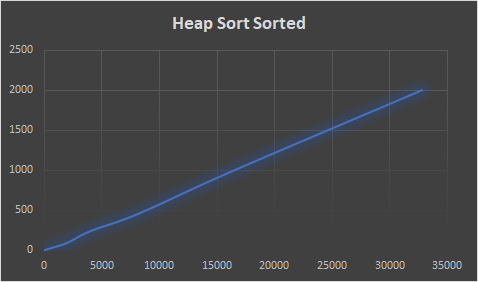
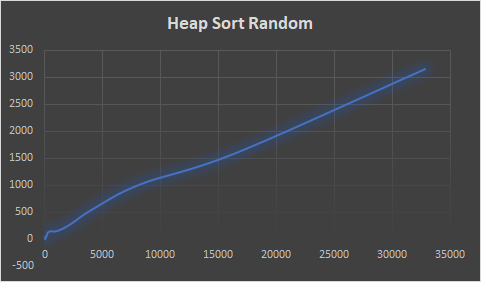
The bubble sort implementation I used was similar to other versions of bubble sort. Very simply, it loops through the array, checking if the current two adjacent elements are in order, and swapping them if they are not. With each pass of the array, the next largest element should “bubble” to the top, hence the name of the sort. The algorithm uses two for loops, an outer for loop that keeps track of the part of the array that has been sorted, and the inside array that iterates through the elements that have not been sorted yet. Inside this inner for loop is where the comparison happens between the two elements, and where the swap is made.

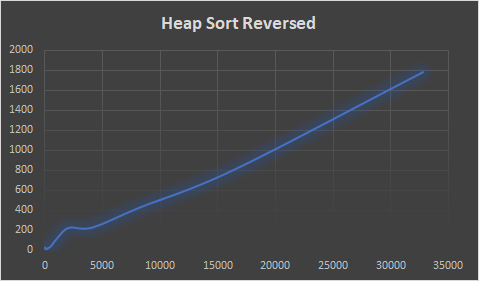
Because you must loop through the array once for every element minus one each time, this algorithm becomes O(n ^ 2), and in its best case scenario (already sorted) it still must loop for each element. The empirical evidence from the graphs matches this theoretical hypothesis, as when the size of the array increases, the amount of time it takes to execute the algorithm increases exponentially, and the difference between each type of test does not vary much. This makes bubble sort a very slow algorithm, as it does not scale well with the size of the input. ****

**Heap Sort:**

For my implementation of heap sort, I used a recursive method that constructs a max heap from the given array, removes elements from the max heap in sorted order, then uses heapify to turn the heap back into a max heap. I achieved this using three methods: a main sort method, a method to build the max heap, and a method to perform heapify. The method that builds the max heap works by calling heapify on each non-leaf node in the given heap. The heapify method works by checking if any of the current node’s children are larger than it, swapping those nodes if it is, then recursively calling heapify on the child node. Doing so will ensure that the given node is a max heap.

For theoretical time complexity, heap sort should be O(n log n), as you must perform heapify on each element at least once, and heapify is a log n operation. Thus, we would expect to see the graphs not grow as rapidly when the input grows in size. And with the results that we got, it shows that the results do in fact match this theory. The time it takes to perform the algorithm does grow exponentially with the size of the array. The performance of the algorithm seems to run the best when the array is sorted in reverse order, which makes sense because that is the closest configuration that resembles a max heap, requiring the least amount of operations to make it a max heap. This time complexity makes heap sort a very good option for sorting large data sets.

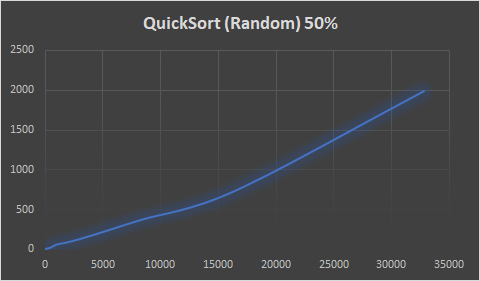
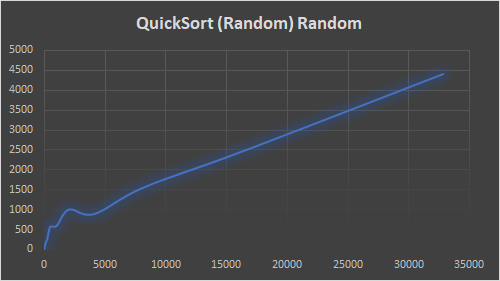
****

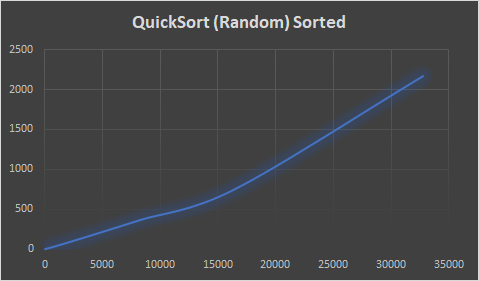
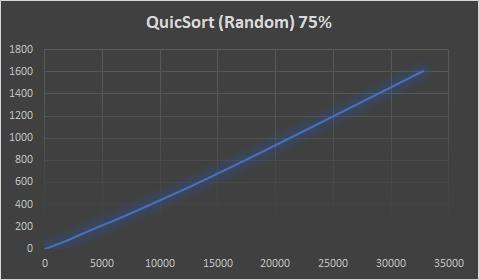
****

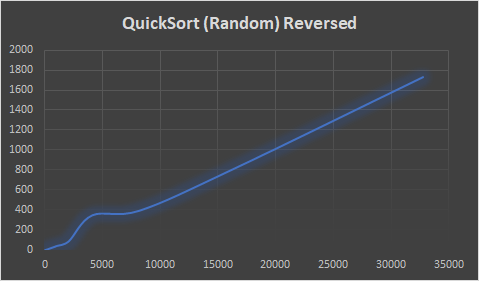
**Quick Sort , Random:**

My implementation of this quick sort algorithm was pretty standard, with the addition of making the pivot randomized. This algorithm works by randomly selecting a pivot, arranging all the elements so they are either less or greater, re-inserting the pivot into its correct spot, then recursively performing the last three steps for the two subarrays to the left and right of the pivot. When you select the pivot, you swap the pivot value with the last value in the array. Then you perform a loop where you check if each element is less than or greater than the pivot, and either keep it in its position, or move it to the end. After doing so and putting the pivot in its correct position, you then repeat for each subarray until you cannot anymore.

This algorithm at its worst will be an O(n ^ 2), but at average and best will be O(n log n). As we can see with the randomly sorted array, we get much worse times than the more sorted tests. This is because when selecting pivots in sorted arrays, they are more likely to already be in the correct spot, requiring less swaps. When comparing this quick sort to the other quick sorts in this experiment, it performs quite a bit worse. This is due to two main factors: Generating a random number takes a decent amount of time, and randomly choosing a pivot is less consistent at getting an efficient pivot point than the other two methods. Because of these two factors, we see the runtime to be longer.

****

****

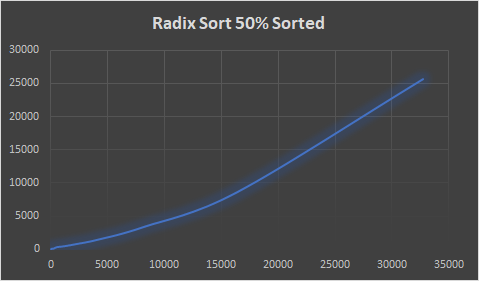
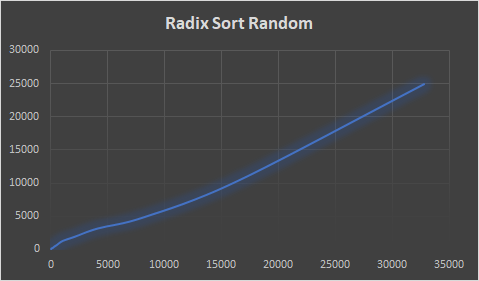
****

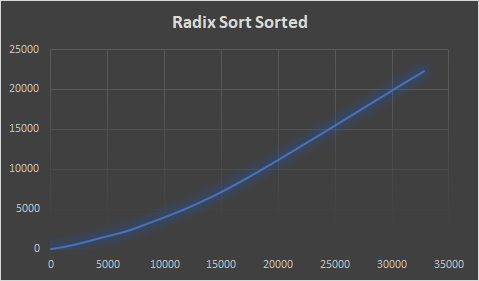
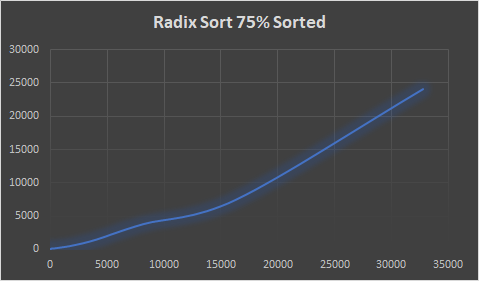
**Radix Sort:**

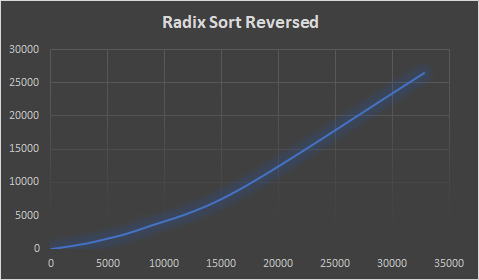
Disclaimer: These graphs are not updated with the real values, because in the implementation of my sorting algorithm there is a Math.pow() calculation I did not correctly place, which increases the time complexity of the algorithm by quite a lot. With the updated code in the repo, it had much better performance than what is shown.

For radix sort, I did a standard implementation. First, I identified the maximum amount of digits in any given number in the array. Then, I created buckets to store numbers with the given digits. Looping through each element, I put them into the bucket matching their last digit, and then feed them back into an array in this order. I repeat this process for each digit until all of the first digits of each number have been sorted appropriately. I create the buckets in a separate method than the sort method, as this seemed like an appropriate separation of functions.

Therefore, this algorithm is O(n \* k), where n is the size of the array being sorted, and k is the maximum number of digits found in any of the array elements. Because of this, the algorithm can be simplified to O(n), as our k value is not very large. The below graphs are not representative of the time efficiency of the fixed algorithm (I performed a Math.pow function more times than needed), but in the fixed algorithm I was able to get much faster results, which reflect this theoretical performance. Compared to many of the other sorting algorithms, this one performs very well.







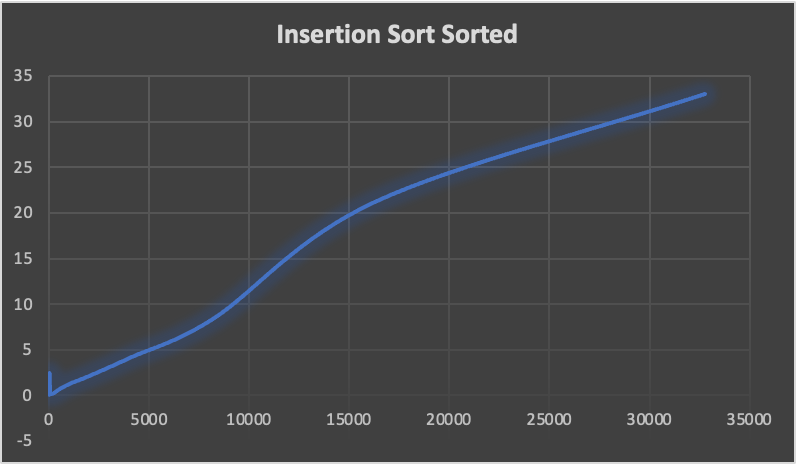
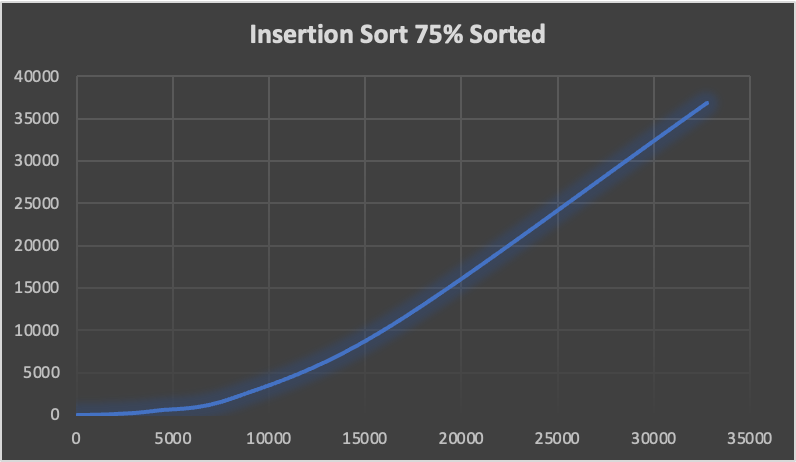
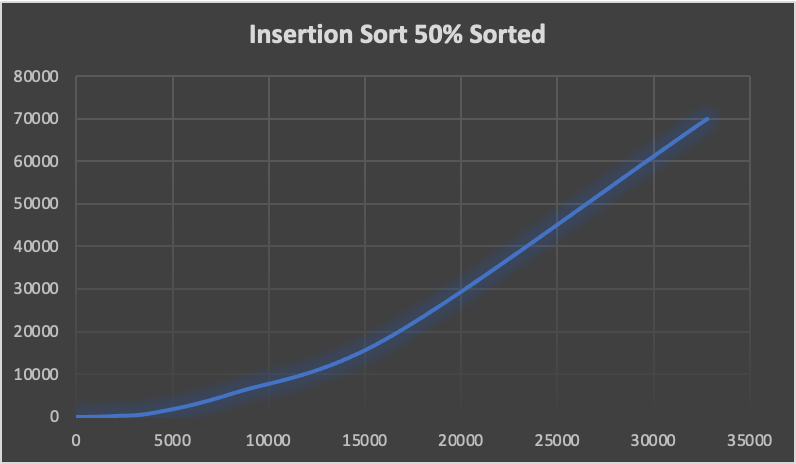
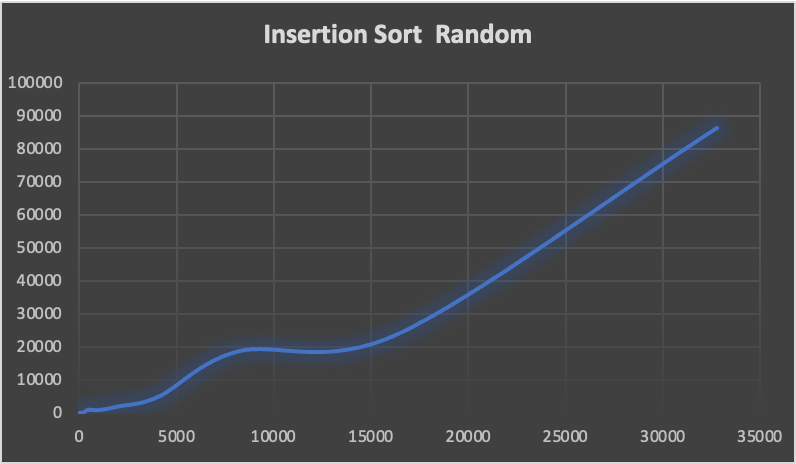
Analysis of Max’s Algorithms:

**Insertion Sort:**

The insertion sort implementation I ended up doing was fairly standard. Insertion sort is considered one of the barest, simplest forms of sorting that can be done, and there are not too many variations of it that I saw when researching it online. The use of a while loop and for loop meant that the array being passed in could be more easily leveraged as we can work with the indices directly (rather than say a LinkedList). This allowed for a rather simple comparison method that is better suited for smaller data sets. Additionally, since insertion sort does its work in-place, it can be used when working in a limited-memory space since it won’t require much more memory to do its work.

Insertion sort is one of the slower algorithms, as it requires a comparison with every element present within the array. This became especially prevalent across all of the different organizations of arrays, as they became larger, the longer amount of time that it took for the insertion sort to take to complete. Important to note is that insertion sort was able to do better when it was presented with the different forms of already-sorted arrays rather than having to do a new sort from scratch- that is when it took a much longer amount of time and was closer to its average of O(n^2). Without having to make as many swaps, the insertion sort was able to dedicate less time to that leading to a faster runtime.

*Graphical Depictions:*



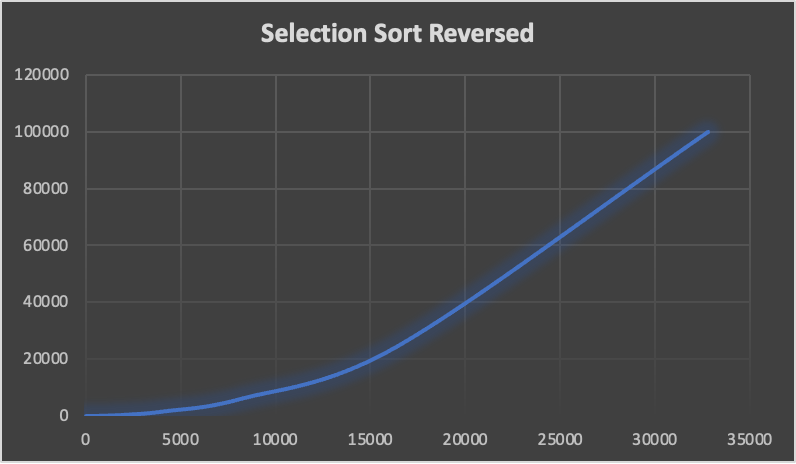
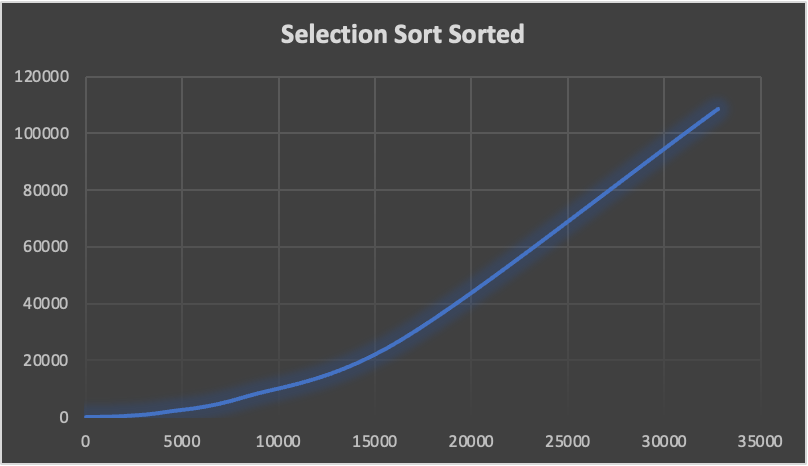
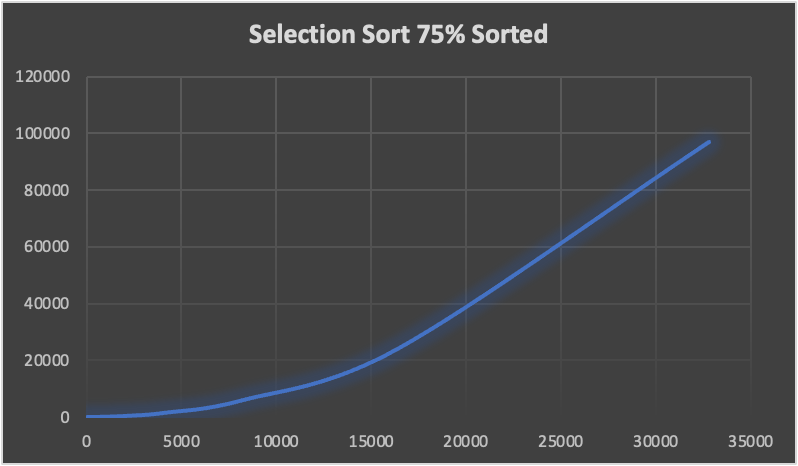
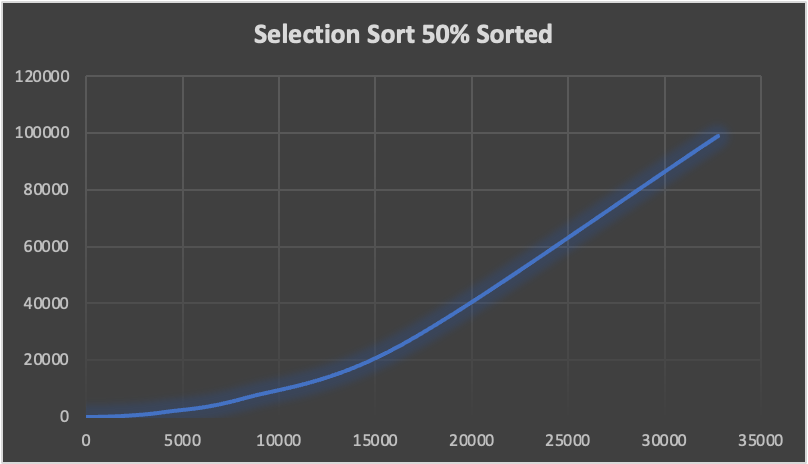
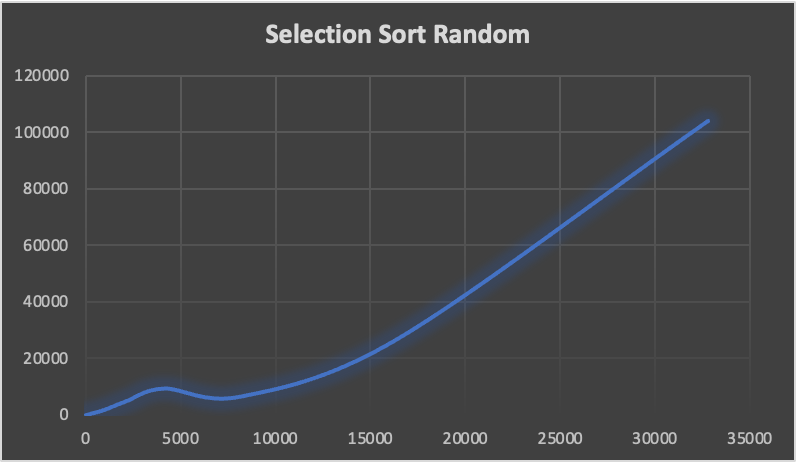
**Selection Sort:**

Selection sort was implemented in a standard method, it is another simplistic method. In my implementation, I had chosen to do the one that I had learned throughout my classes which involves using two different for loops to keep track of the different swaps. In a selection sort, we look at an unsorted versus sorted area, and expand on our sorted area until it encompasses (mostly) the entire array. By using two for loops, it was easier to keep track of the indices for both the unsorted portion and do the swapping as needed to expand our unsorted portion. Since the swapping is kept within the array itself, it is in-place and can be better utilized when dealing with memory constraints

Across all three categories of best, average, and worst in terms of time complexity, selection sort does amongst the worst with an O(n^2) across all three different categories because of the two for loops. Even when dealing with any sort of sorted array, it would still perform roughly the same amount of operations as its counterpart in an unsorted array, it has no choice but to keep expanding that unsorted portion until it encompasses the entire array. This lack of sorting means it would perform pretty badly regardless of what kind of pre-sorted array you throw at it.

As a result of this, we would expect the selection sort to be one of our worst performers, and even worse than its close relative insertion sort. Indeed, when looking at the graphs between both for the different sorting algorithms, most of the insertion sort did slightly a bit better. This likely owes to the fact that insertion sort has a better best case runtime of O(n) which is much better than O(n^2) that is the best possible with selection, and can be easily seen when looking at the different graphs and recognizing that most of them taking roughly the same amount of time.

*Graphical Depictions:*

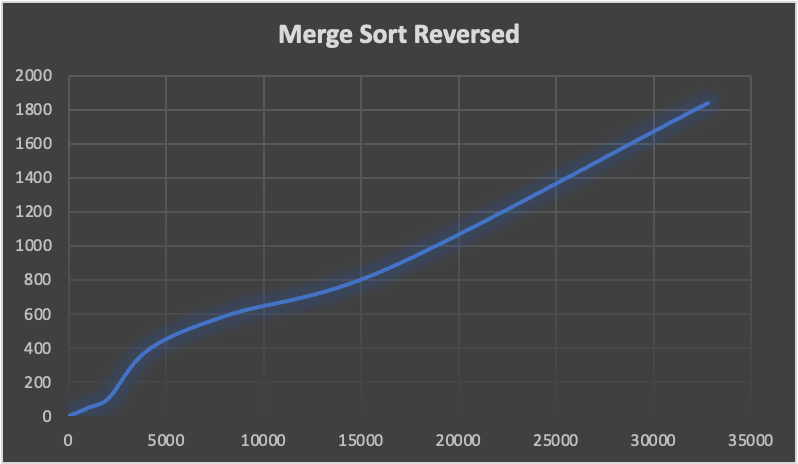
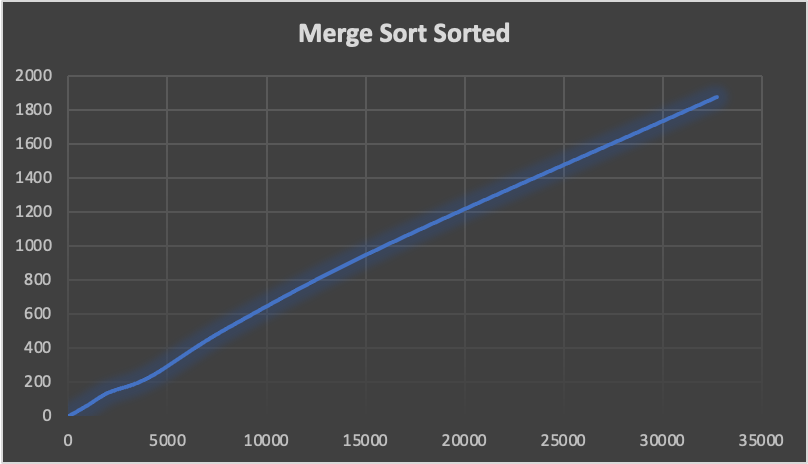
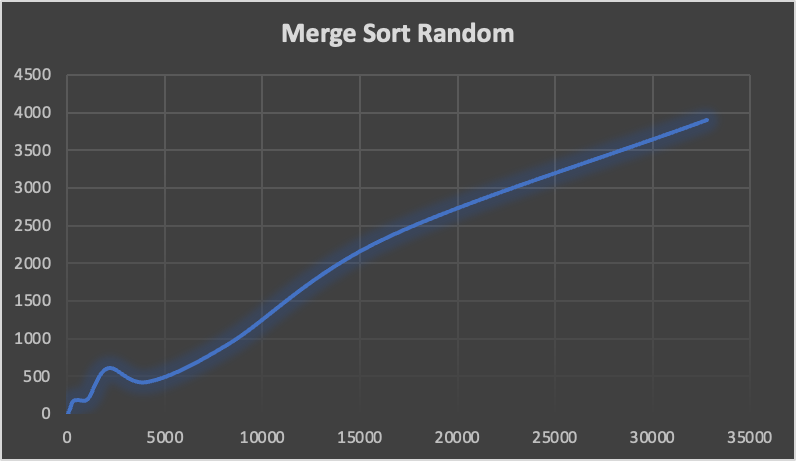


**Merge Sort:**

Merge sort is a much better algorithm (in most cases) that can handle larger datasets with ease. It involves using a divde/conquer strategy that splits the array in half continuously, until it reaches a base case where it can begin comparing and building itself back up. When researching different implementations, there were some that did the division using iterative approaches. I chose not to take this approach as I felt it would muddle the divide and conquer concept that merge sort is usually associated with. Additionally, I didn’t want to implement code that I wasn’t as familiar with, as I have always seen merge sort as a recursive algorithm. Lastly, merge sort is not typically in-place, so it would require more space to perform the work of saving the different halves of the arrays then putting htem back together, so caution should be exercised if being down in a memory-short space.

Merge sort is also a stable algorithm, meaning that if keeping the relative order of equal elements is important, merge sort is something that can be looked at for both its efficiency and stability. When comparing Merge Sort, which has an average Big O(n log n) against a simpler sorting algorithm such as selection sort- it becomes especially evident of which should be chosen for larger data sets. Not unlike selection sort, its best, average, and worst are all the same O(n log n) and while the graph of selection sort reached into the six digits, merge sort never did. When looking at the graphs of this algorithm, and the relatively fewer runtimes, it shows that the more sorted that the array becomes, it (typically) is able to finish the sorting process in fewer amounts of time. This is likely due to the amount of swaps that would presumably be needed when putting it back together would be lessened, and so it could just put it together instead.

*Graphical Depictions:*

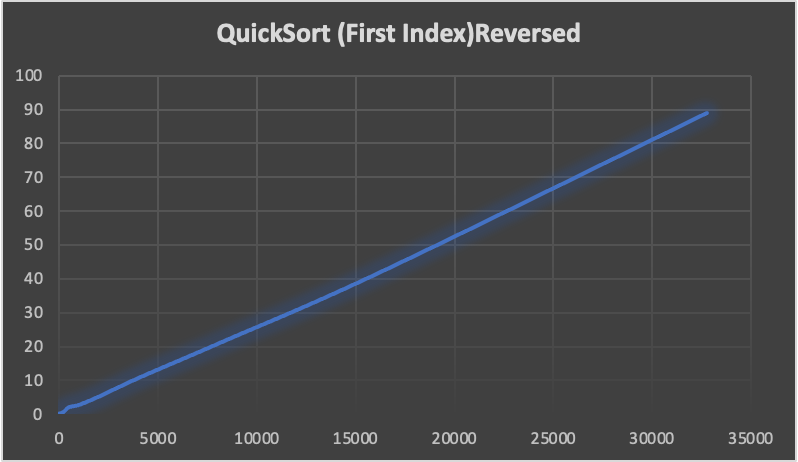
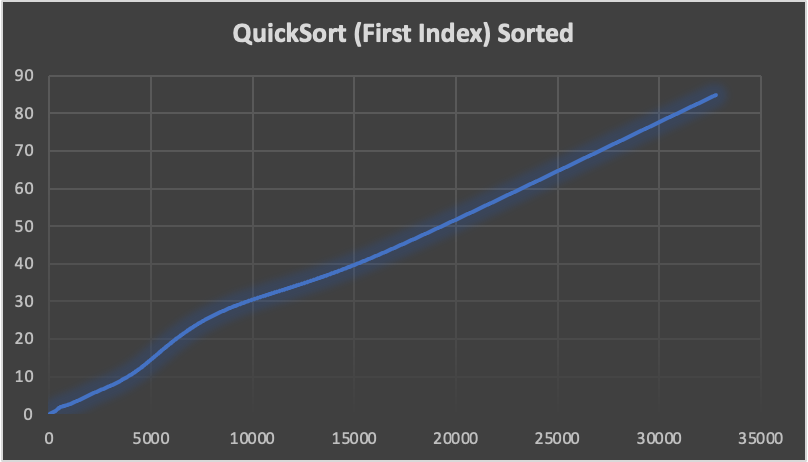
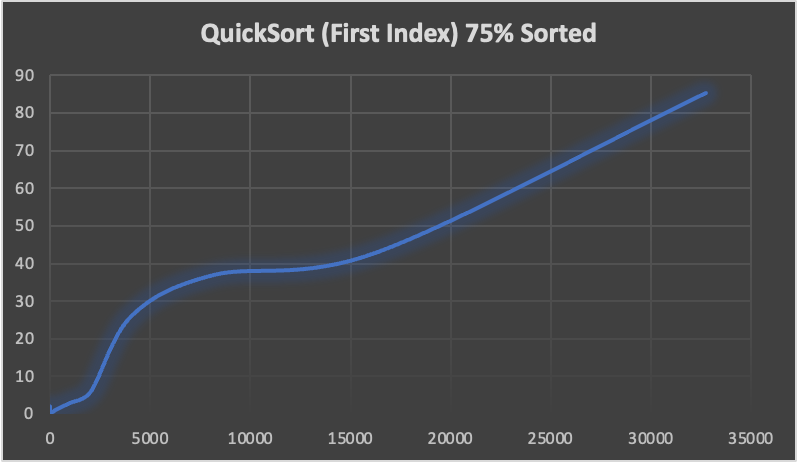
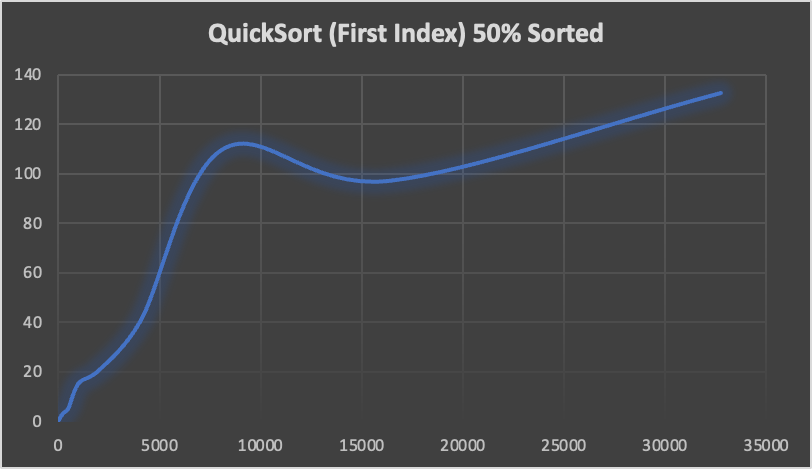
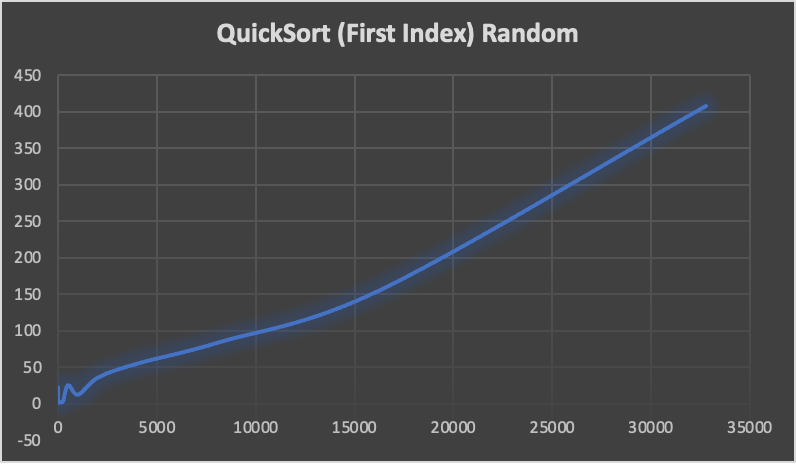


**Quick Sort, First Index:**

Quick Sort using the first index is a possible variation of the quick sorting algorithm. Quick sorts use a pivot element which is used to make every element that is less than the pivot element go to the left end, and everything that is greater to the right. This is done through a (typically) recursive method that does a partition to create this separation and sort. The implementation I chose was nearly identical to what a standard implementation woud look like, but I did have to make a modification to the base case that is used to go into the recursion. Since there remains a possibility that the pivot we chose is “bad” and one half of the array goes into too-deep of a recursive method that creates a stack overflow. To account for this, my modifications make it so you choose to essentially do the one that has the least first, allowing you to circumvent that restriction.

Similar to merge sort, it also requires additional memory for the partition of arrays, and then the act of putting them back together. Unlike merge sort, it is unstable meaning it is not guaranteed equal elements would be in the same relative order.In terms of how it performed amongst the other algorithms, quicksort has a worst case of O(n^2) but a best and average of O(n log n). Additionally, this implementation of quick sort was easily able to handle the arrays that were already sorted, and was able to do better than the other implementations of quick sort finishing, most often, in faster time than the other cases. Interestingly, despite radix sort being most of the time closer to linear (really is O(nk)) it was still able to beat it when looking at runtime and the differently sorted arrays. This is perhaps different that we would expect since O(n log n) grows at a faster rate than O(n).

*Graphical Depictions:*



**Design Implementation Details**

When designing our classes, we thought about different implementations that both made intuitive sense to all of us, but also allowed for the proper organization of classes. In the end, that led to us dividing our classes (with the exception of main) into three different packages: Sort, ArrayCreators, and SortTester. All of these packages contain within them classes that are important for their own purposes. Ultimately, all three classes work with each other in order to create a final working product that can be run and produce the results that are required for the project. We ultimately chose runtime as our second measurement because we felt that it made easier sense to understand, rather than having to keep track of the basic operations for the different sorting algorithms (and the array that was tested).

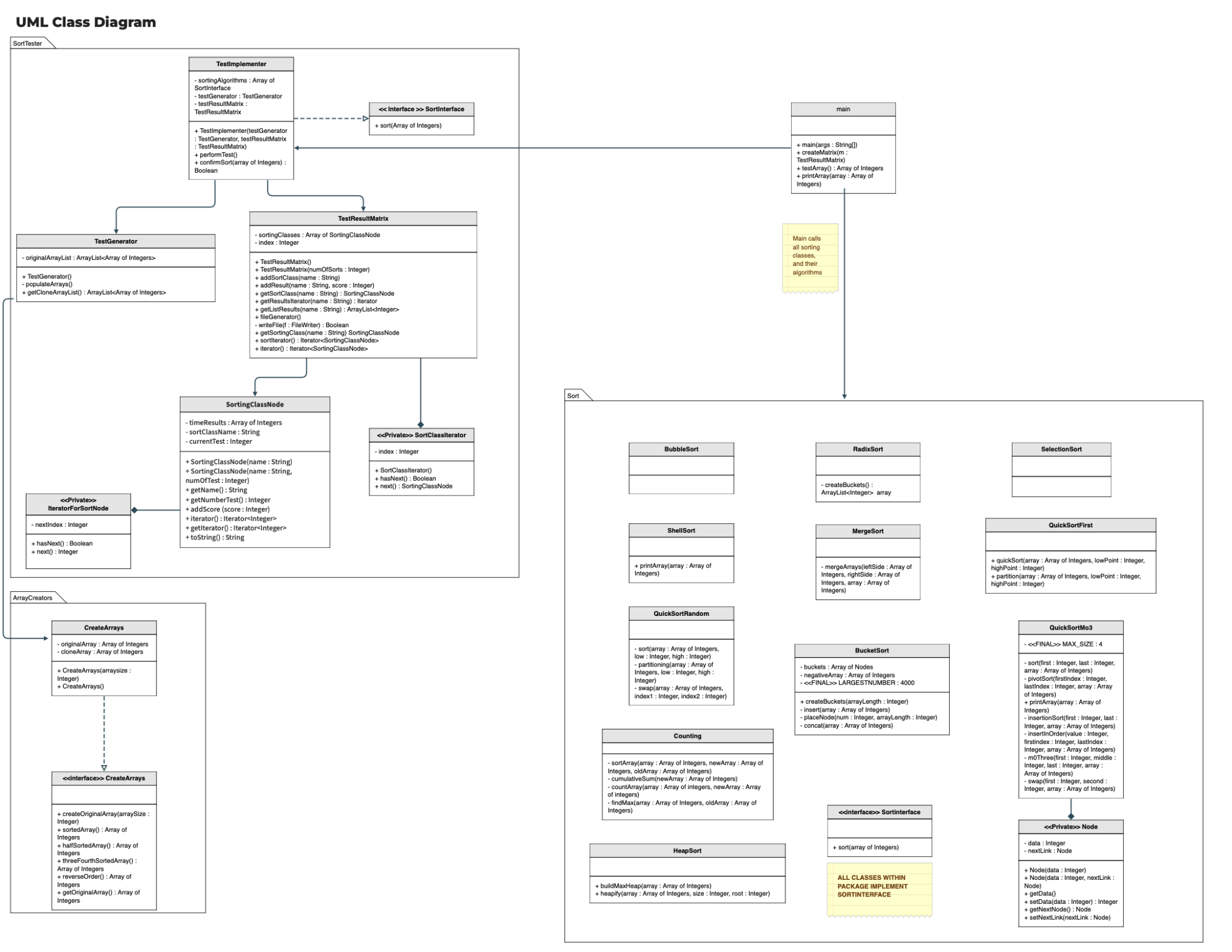
Sort is our package that contains all the different sorting algorithms that will be tested. All of the different classes within were required to implement the SortInterface, that is it required them to have a method called sort(). Initially, our idea was hopefully to make it easier to call the different methods via this method, but as it evolved we divulged a little from that path and instead used it as a way to identify our different sorting classes as identifying them of a specific object type (all implement SortInterface).

ArrayCreators is the smallest package, and contains the class (and its interface) that can create the different arrays that are required to test on a single algorithm. By making it its own class, we were able to ensure that the array that will be tested is kept in a single class that can remain untouched. Additionally, it had the added benefit of ensuring that we can call the appropriate size of the array, and make sure that the integers within remain unchanged only are changed in terms of order. Lastly, the creation of it as its own package meant that we could more easily access it in other classes when needed, while keeping our code organized.

SortTester is responsible for the bulk of our logic for conducting the various tests and organizing it on the different sorting algorithms. It contains four classes: TestGenerator, TestImplementer, TestResultMatrix and SortingClassNode. TestGenerator is responsible for the creation of an ArrayList that can hold the differently sized and arranged arrays (created by ArrayCreator) that can be passed to TestImplementer for use. TestImplementer is the one who arranges for the testing of the different sorting algorithms using the acquired arrays, and proceeds to send the results to TestResultMatrix. TestResultMatrix holds that data (via SortingClassNode objects) and writes it into a file that can be used to create graphical depictions.

Our implementation allows for proper encapsulation, and allows for easier debugging since we can identify more easily what class, and what method an error occurred. Additionally, since our project was able to be compartmentalized this way, it meant we could code individually on our time, without having to worry about indirectly impacting the code of another person. Ultimately, our design implementation worked very well for our group project, and allowed for us to create a functional project able to output the appropriate results

*UML:*



**Overall Algorithm Analysis**

When looking at the various algorithms, many of them have their own benefits and drawbacks. If you look at the algorithms that are often considered the simplest such as insertion, selection, bubble, and shell they are able to do most (if not all) of their sorting in place, which makes them especially valuable if you are looking at a system that is limited in memory, such as mobile devices, or embedded systems like those found in medical equipment that may not have their luxury of larger storage, so a method that won’t take much more space is vital to ensure the machinery is able to operate properly. Unfortunately, this same simplicity makes them rather slow- and when working larger datasets, they can still be used but not to the same degree of efficiency.

On the otherhand, many of the other sorting algorithms we looked at made use of extra space that was available to them, which meant taking extra memory. In some of the cases, such as quick sort (first) utilized extra memory to perform its recursion, and its methods had to be modified to a degree that can be considered unconventional in order to account for taking so much memory that it caused a stack overflow. While a modification was made to it to avoid this- it goes to show that these algorithms can take a lot of memory if not careful. At the same time, many of these algorithms were shown to do much better at sorting the algorithms at a reasonable amount of time, with counting sort, which doesn’t utilize a conventional sorting comparison, being able to take the cake as the fastest algorithm.

These algorithms that are much quicker are better suited when space is not as big of a concern, and when you know you have to sift through much data. For example, a bucket sort could be used effectively when presented with something like a database with ample space. Although its space complexity is Big O(n) which isn’t great, it’s able to perform its work much quicker than say an insertion sort that has a space complexity of O(1). Overall, when looking at the various algorithms, it becomes obvious that they each have their own purposes, and that the use of a specific algorithm should be based on what constraints are put on to you, and what is more important to you whether it be achieving the fastest sort, easiest implementation, or something else entirely.

| **1** | **Frank M Carrano and Timothy M Henry, “Data Structures and Abstraction with Java”Fifth edition,pp 452 - 453, Pearson ISBN 10: 0-13-483169-1** |
| --- | --- |
|  |  |
|  |  |
|  |  |