**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:

**Test Objective:** This experiment will compare the differences in run times between separate sorting algorithms using the same input parameters. The requirements were to ensure that all test were comparing the same test arrays. There are 5 separate types of arrays ranging from random, 50% sorted, 75% sorted, 100% sorted and sorted in reverse. Array sizes are increased from 24 elements to 214 elements. The goal is to verify the difference of run times between different sorting algorithms and validate unique aspects of individual algorithms. In doing so, we can better understand the unique qualities that different sorting algorithms exhibit and increase our knowledge of such algorithms.

**Test Environment details:**  Due the differences in computer system and a multitude of other factors involved in computations. The test environment was more focused on ensuring the same parameters were given to each sorting algorithm instead of measuring the performance external of the currently running machine. The results shown for my portion of the analysis were generated on a MSI laptop with 32 GB of RAM. The processor is an Intel i9-9880H 9th generation CPU with 8 cores running at 2.3GHz. All programs are created and ran thought Jet Brains IntelliJ IDE with no other application running at that time. The Test and results are saved at the GitHub REPO [https:github.com:jjemi8884/SortingAlgorithmsExperiment.git](mailto:git@github.com:jjemi8884/SortingAlgorithmsExperiment.git).

**Summary**: Since different machines will compute the following algorithms in different times, we will more focus on the delta in run times vs the actual run times themselves. To ensure that everything was fair between all algorithms. We created first primed the classes that algorithms where stored in, by running a few “warm up” test. Then we started a timer and conducted the test in the exact same manner each time for each sort algorithm. Furthermore, to remove any error in randomness, we ensured the test arrays where exactly the same array for each sort algorithm.

**Results:**

**The Bucket Sort**

The graph to the left shows how bucket sort grew linearly as more items in the array were added.

This is consistent with the average and best-case time complexity of Big O(n). worst case would be if Big O (n \* n). This would be observed when the numbers to be sorted are very near to each other.

This bucket sort is given its “maximum number” that will identify how many buckets are needed, other algorithms will find the maximum number and then create that many buckets. This brings us to our first drawback of the bucket sort, for sorts that have high values items, this would not be a good option due to the number of buckets that would be generated and iterated through for no reason. To save a little time, the maximum number, if know beforehand, can be automatically assigned. This would save O(n) time.

This bucket sort also uses links that are generated to be stored in the bucket. If multiple items are in the bucket, then the linked list will be inserted into the correct spot within the bucket. This seems like the most efficient method instead of using arrays within the buckets and possibly having to dealing with array size adjustments. Since the sort uses the element in computing the bucket index, this type of sort is only limited to elements that are able to be represented at integers.

Results from the bucket sort show that it does better when the items are already in a sorted order. This is inconsistence with the expected results due to the indexing aspect of the sorting algorithm. Yet it clearly shows that sorted in reverse order is far quicker than the random sort.

As for space complexity, bucket sort uses O (n + m) due to creating the bucket array. This would not be a good algorithm to use in memory limited application like embedded systems.

Bucket Sort (yellow) compared to Heap Sort (Red), Counting Sort (Blue), Quick Sort Mo3 (green) (green), Quick Sort Random (white). You can see that Bucket Sort is slightly better than most of the divide and concur type of sorts that are recursion (stack) heavy. If memory is a concern, the bucket may not be the best choice. As for all the other algorithms, their numbers are greatly higher and would have skewed the chart.

With the sorted array the bucket sort (yellow), as expected, was not as quick. The Insertion Sort (red) was the fastest by far. Then the counting sort was next (green) followed by the Shell sort (light blue) and quick sort (Mo3) (orange). The next fastest sorting algorithm was the merge sort sorting 32k items in just under 1600 millisecond. Where the bucket sort was just under 900 milliseconds. All other algorithms where far slower and would have skewed the graph it added to the chart.

**Bucket Sort conclusion**: The sorting algorithm preformed very well over all. It was the second fastest sorting algorithm when the elements are random. With sorted arrays, it was not the quickest, but it did perform better than most other algorithms by a factor of 2 or more. This sort does use extra space for the bucket array and when compute is O( n + m) for space. Time complexity is O (n) as observed and expected.

**The Counting Sort:**

As seen in the graph to the left, the counting sort algorithm is linear in regards to the number of items needing to be sorted. This is consistent with the expected results for counting sort. As it has the big O (n) for complexity. As for space complexity it uses 3 arrays (third array due having to sort the original array so space complexity is O(2n + m).

Above you will notice that counting was linear across all the types of arrays that where passed to the algorithm. Again this is some what supprizing since this is more of a index type sorting algorithms that uses the element in determining its location in the sorted array. It was expected that counting sort would not have been as fast as it was in the sorted algorithms. But, it should be noted that when some of the arrays is sorted it was fairly consistant in its time to sort. As observed in the above chart. All “sorted” times are within 20 millisecond of each other.

The counting sort has some major drawback, first it only can sort elements that have some type of integer, and the integer has to be fairly close to the array size. The counting sort would lose its advantage if the elements within the array are large than the array size.

Compared to other sorting algorithms, counting sort was the fast algorithm as was to be expected. As noted in the chart to the left, counting sort (blue glowing) was over three times as fast as the bucket sort (yellow). This was unexpected since they both use the index scheme to sort.

When sorting a sorted array, counting sort (yellow) was the second fastest algorithm behind the insertion sort (red). Do note that counting sort is 2 times quicker then shell sort( orange) which was the next fastest algorithm. All other algorithms are far slower and not shown.

The counting sort does iterate through the array at the beginning looking for the maximum number to build the counting array. One could instead give the maximum number to the counting array to speed up this process by O(n) but there is another reason for iterating through the array at the beginning. The classes are designed to manipulate the original array, unfortunately, counting sort does not do this and instead create a new array to be returned. So instead of copying the array at the end, it uses the “Find Max” to also copy the orignal array and then using the copy to manipulate the original array.

**Conclusion for Counting Sort:**  As seen above, counting sort is a very fast sorting algorithm, and when the elements are integers that are index fairly close to the array size then counting sort is very efficient. Problems are when the elements are high in number, or are far apart from on another then the algorithm loses it efficiently.

**Quicksort (Medium of Three)**

Quicksort median of three is one of three quicksort that are tested. This uses a median of three approach to choose the pivot. Furthermore, this method will revert to an insertion sort when the array size is less 10 (blue). One can decrease this to an array size of 4 which is done later on. Performance improvements that are fairly drastic in regards to the random array. The other charts in this report are with the array size of 10 or less before the insertion sort will take over.

Looking at the chart below (4 arrays or less) we can see that quicksort Mo3 is fairly consistent when an array is only partially sorted with only 500 millisecond between the random and 75% sorted arrays with 32K elements. But when fully sorted or reversed we see drastic improvements in time. 

When the quicksort is run again with 10 arrays instead of 4 arrays being the limit for switching to insertion sort We can see that there is an increase in time when the arrays are larger and random but when they are sorted, it is slightly quicker when sorting sorted arrays.

From the results, is seems that when running quicksort Mo4, it is quicker for random numbers to only switch to insertion sort after we reach array lengths of 4 or less. Whereas if we know the elements are partially sorted, then increasing the array size has some advantage.

Quicksort Mo3 (green line) was a fast-sorting algorithm compared to the rest of the sorts and is again groups with the top performers about the middle of the pack. What is surprising is the random quicksort method(white) seems to perform better with larger array sizes. Bucket and counting sort also preformed faster as expected. But It was something of note as to how close quick sort was to Bucket sort for the random sorting.

Quicksort (orange) did better than expected when sorting an array that is already sorted. Quicksort was again grouped with the top preforming algorithms for sorting the sorted array. As expected the counting and insertion sort preformed twice as fast. What is surprising is the shell sort, an augmented form of insertion sort should have done better

**Conclusion:** Overall Quicksort Medium of Three is a fast sorting algorithm and was with the top performers across all types of array sorting. The time complexity for this algorithm is (n log n), thought it does look liner in the graph that is above. Switching back to max of 4 elements before insertion method takes over shows a better logarithm equation and for any sort that requires a large array, as seen in the chart, a max of 4 will achieve higher performance.

As for space, this is a recursive call and could does use stack space for the recursion calls.

**Shell Sort:**

Shell sort is my final sort to analyses. As seen below it is linear which was expected being O(n2). The advantage of sell sort is the space complexity of O(1), where it uses the original array for sorting and does not create any new arrays.



The numbers for shell sort are consistent for a sort of this type. Except the full sorted and reverse sorted are higher then expected. As shared previously, shell sorted I would have imagined would have been closer to insertion sort for a sorted algorithm where as it was actually closer to the merge sort as we will see.

When comparing shell sort (red) to the other algorithms we can see it was in the middle performance group. The quicksort Mo3 (green) and quicksort (random)( blue) are displaying the logarithmic tendency and starting to preform better the more elements that need to be sorted.

With quick sort preforming in the middle performance group, yes not using any extra memory. It would make this sorting algorithm excellent for low memory operations that do not require high number of elements to be sorted.

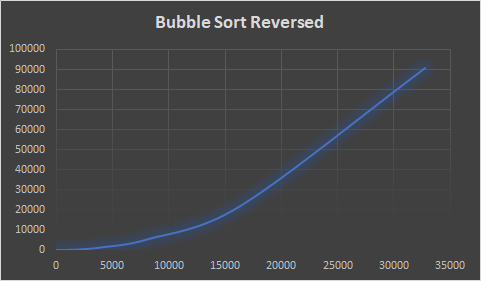
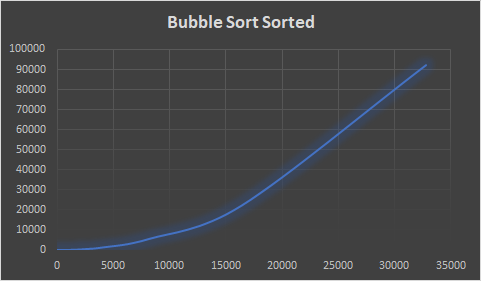
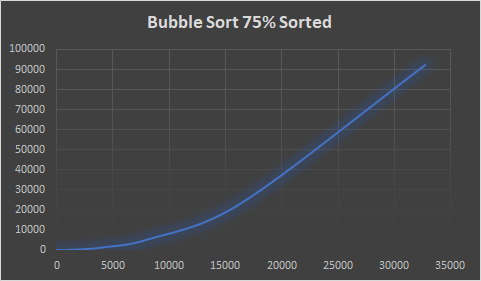
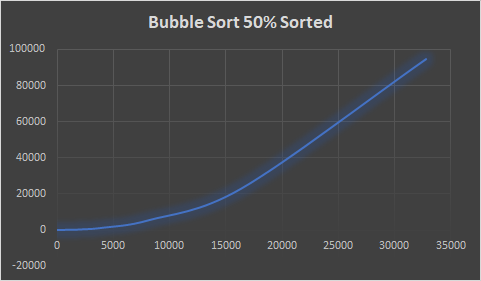
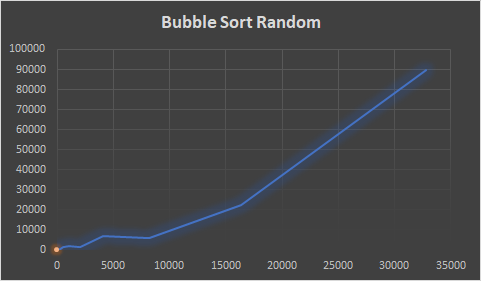
Comparing the shell sort (yellow) against the other algorithm in when sorting a sorted array, we can see that it is with the high performers being able to compute 32k in 700 millisecond. As stated previously It was expected for this sort to do better due to the close nature of the programming to the insertion sort(red). But that is not the case, instead preforming poorer than counting sort (bight red). As noted on other graphs, all other algorithm are many time higher in time and would skew the chart if added.

**Conclusion for the shell sort:** Shell sort performance with random sort was close to the top performers and again its was close to the top performers for when the array was sorted. Shells sort average time complexity is just under O(n2) meaning it will lose this its performance when the arrays are large. The strength of this algorithm is when space is of concern, the shell sort will not take up any extra space for processing an array. The other algorithm that has the space complexity if O (1) is Insertion sort. Which performed better when arrays are over 50% sorted. When arrays are random, shell sort would be the faster performer.

Analysis of Jackson’s Sorting Algorithms:

**Bubble Sort:**

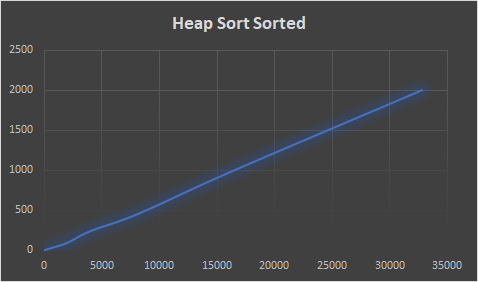
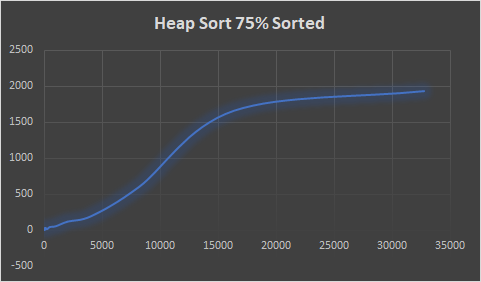
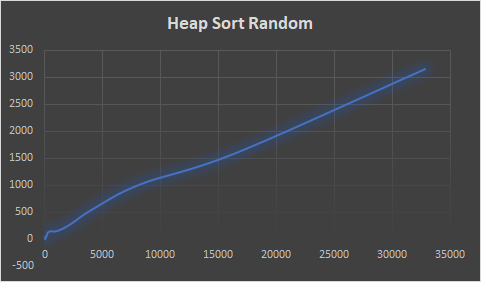
The bubble sort implementation I used was similar to other versions of bubble sort. Very simply, it loops through the array, checking if the current two adjacent elements are in order, and swapping them if they are not. With each pass of the array, the next largest element should “bubble” to the top, 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.

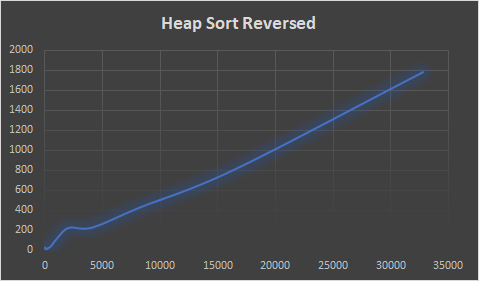
Because you must loop through the array once for every element minus one each time, this algorithm becomes O(n ^ 2), and in its best case scenario (already sorted) it still must loop for each element. The empirical evidence from the graphs matches this theoretical hypothesis, as when the size of the 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.

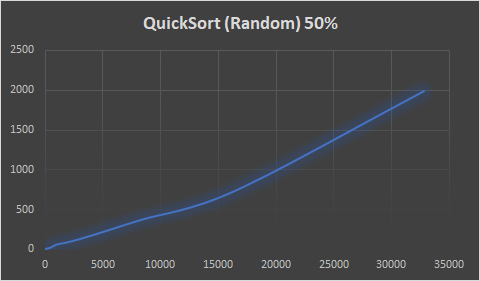
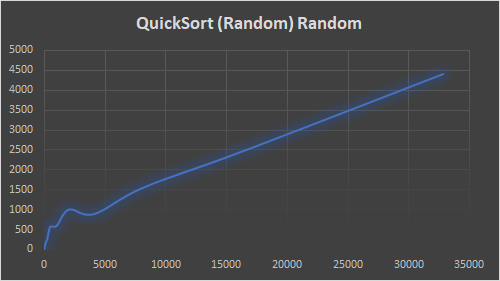
****

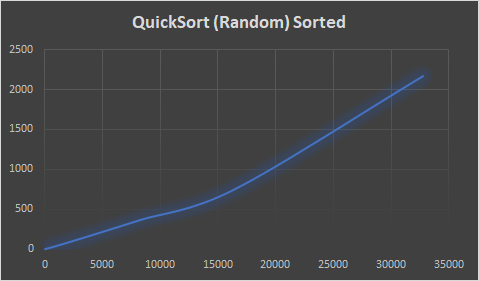
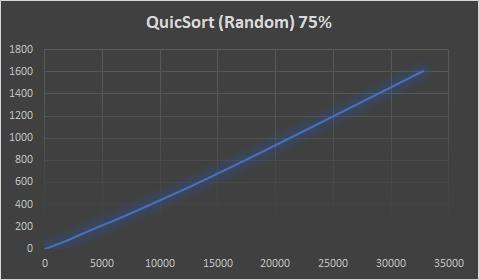
****

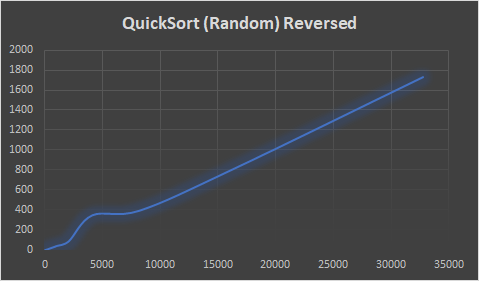
**Quick Sort , Random:**

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.

****

****

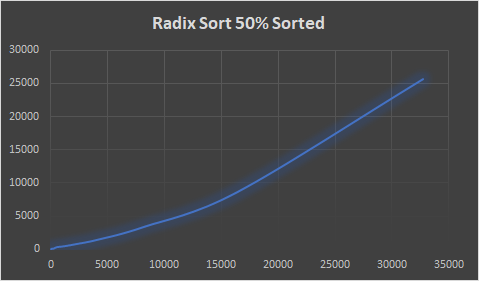
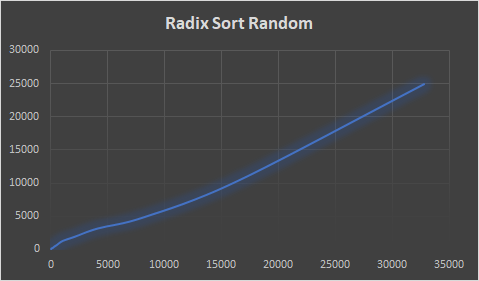
****

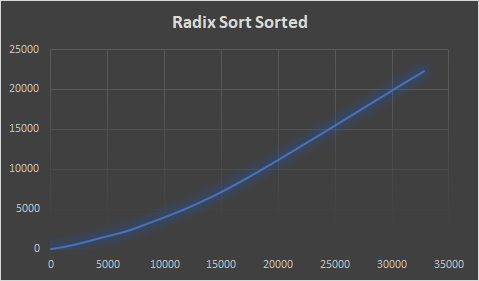
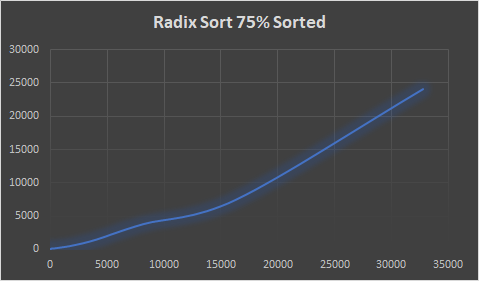
**Radix Sort:**

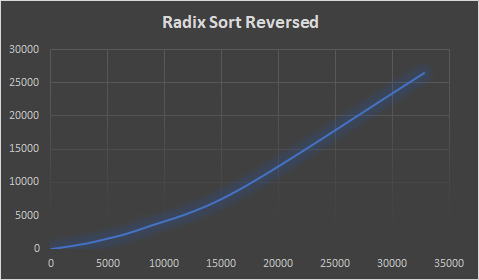
Disclaimer: These graphs are not updated with the real values, because in the implementation of my sorting algorithm there is a Math.pow() calculation I did not correctly place, which increases the time complexity of the algorithm by quite a lot. With the updated code in the repo, it had much better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







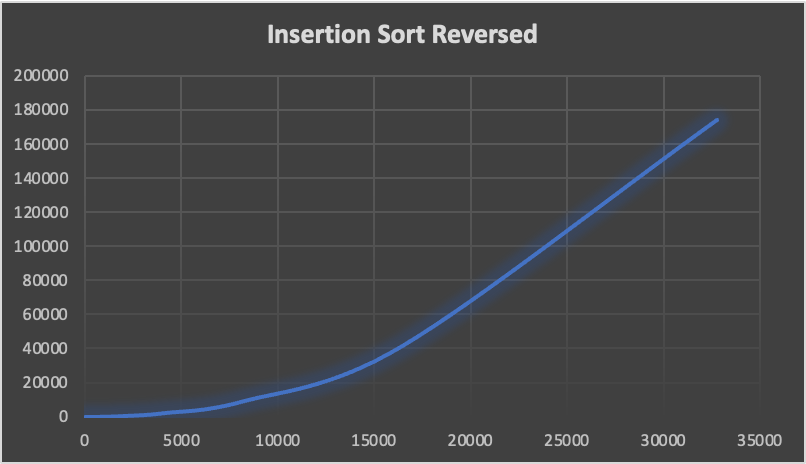
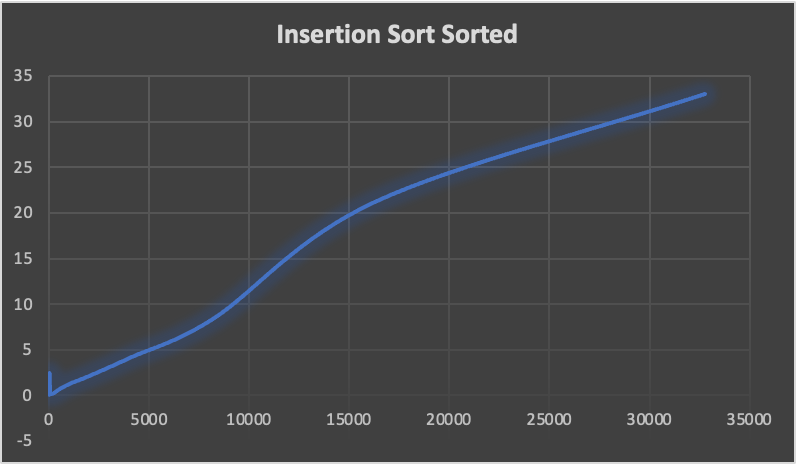
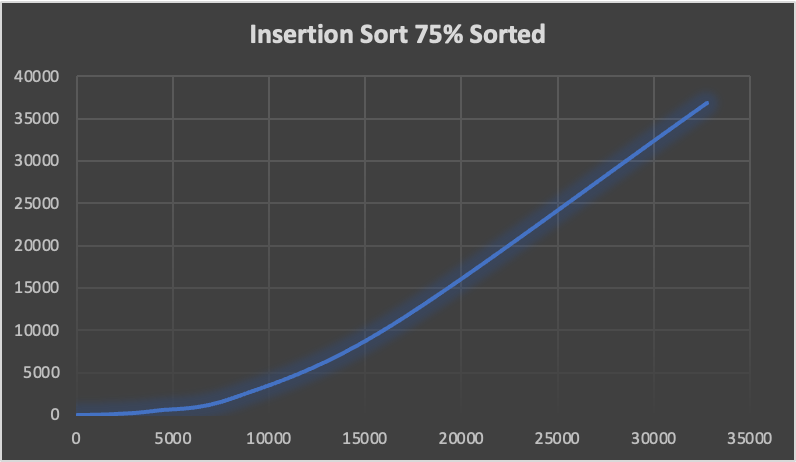
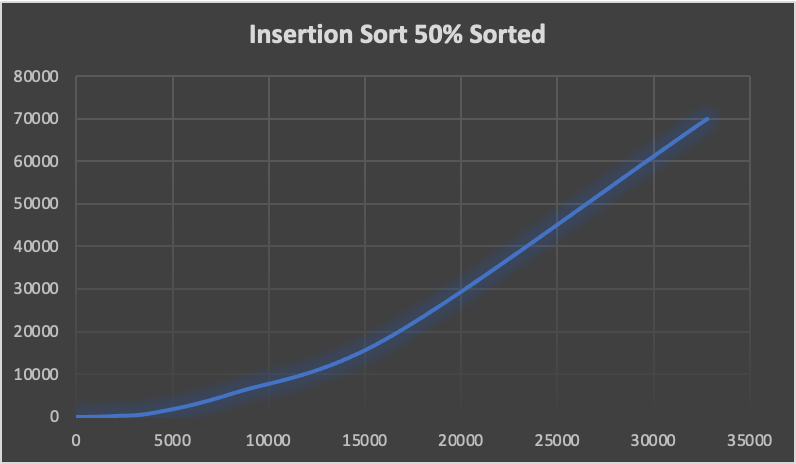
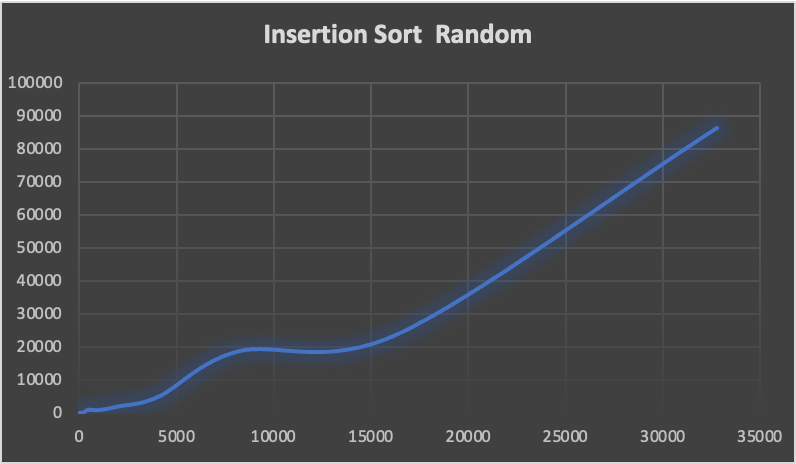
Analysis of Max’s Algorithms:

**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:*



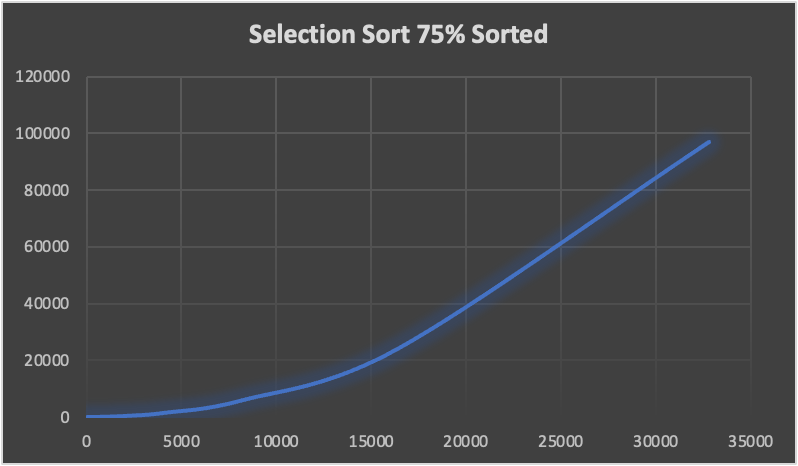
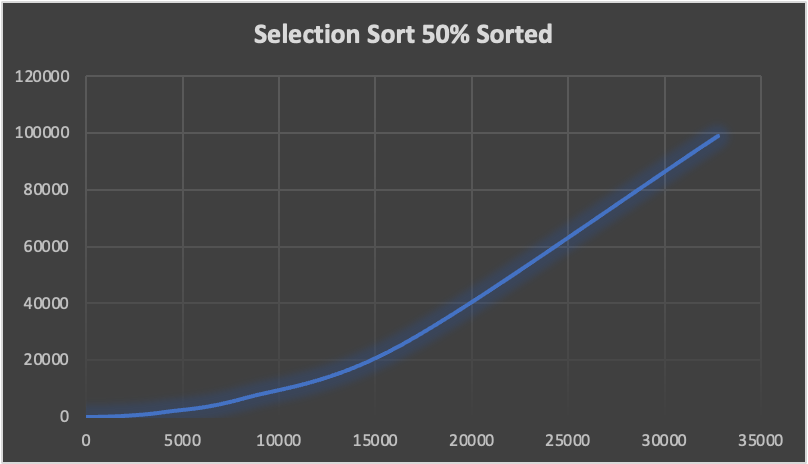
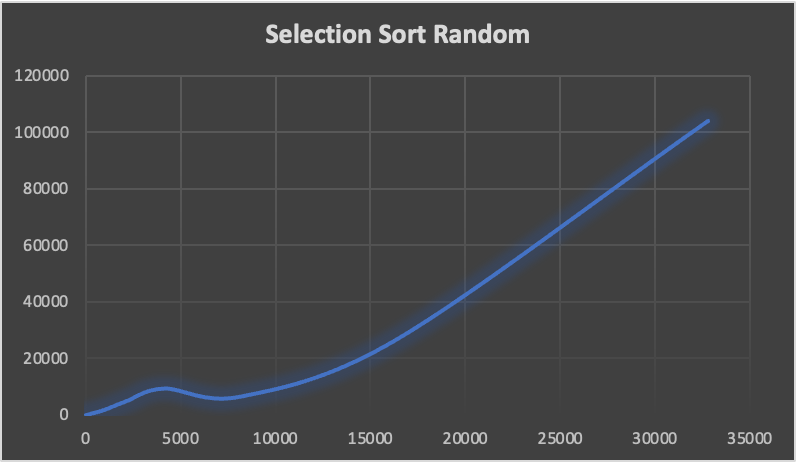
**Selection Sort:**

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:*

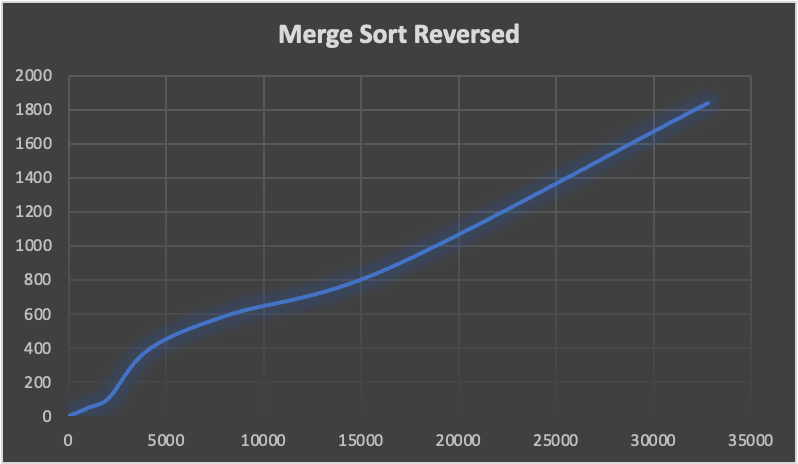
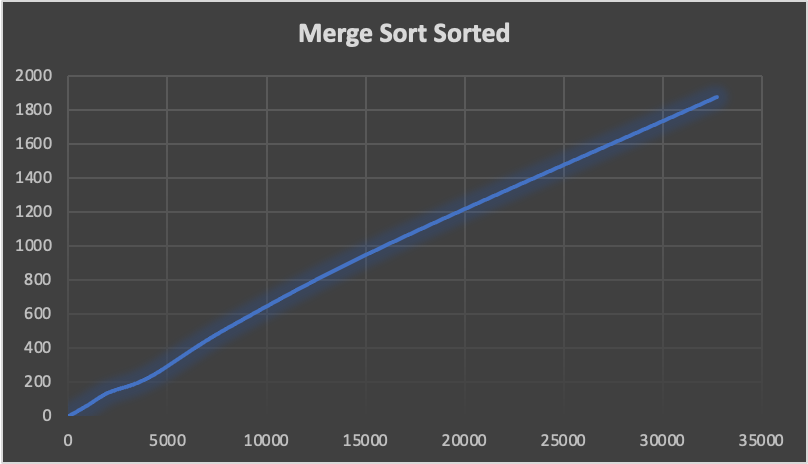
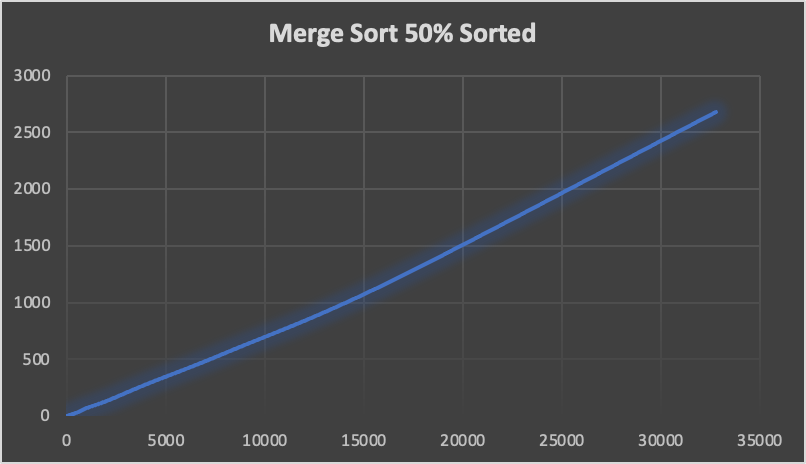
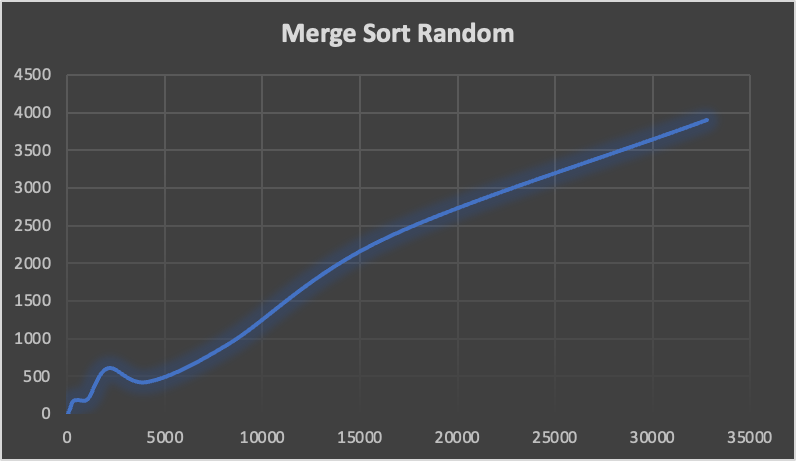


**Merge Sort:**

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:*

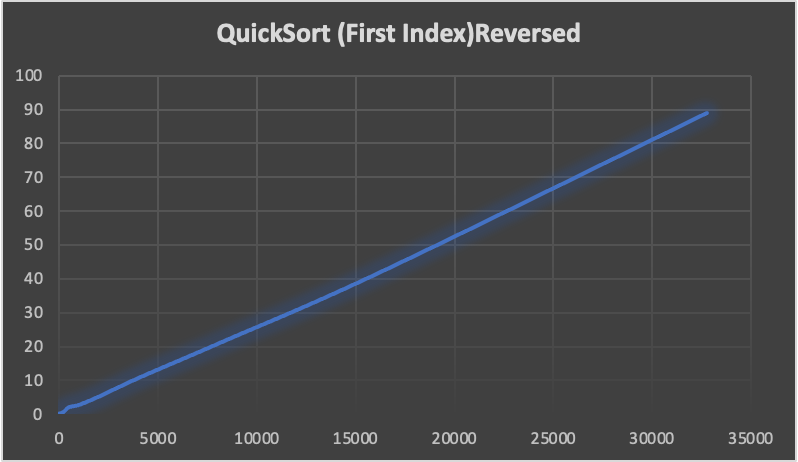
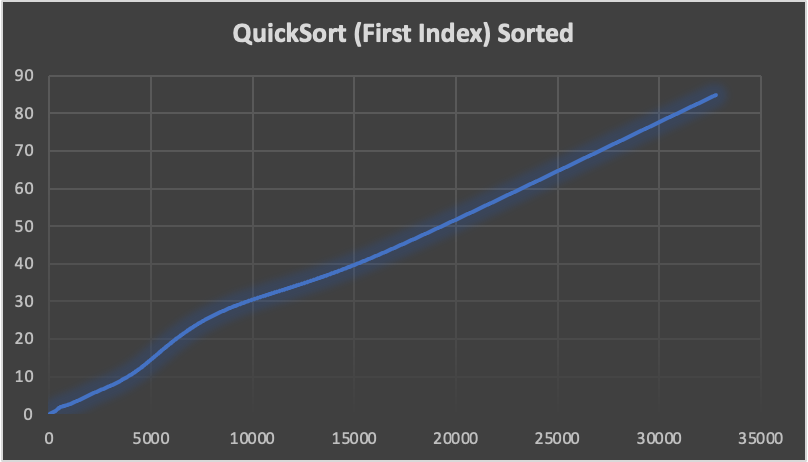
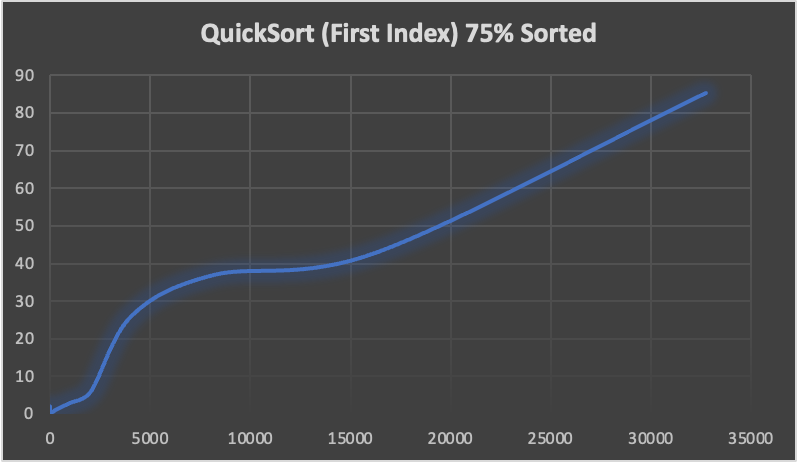
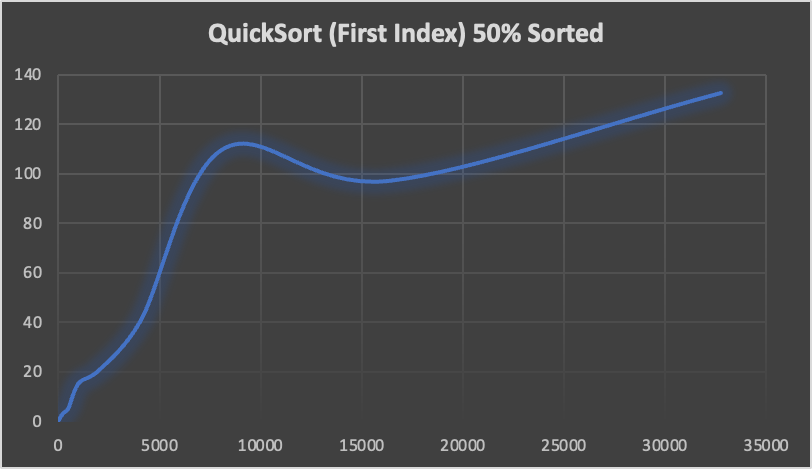
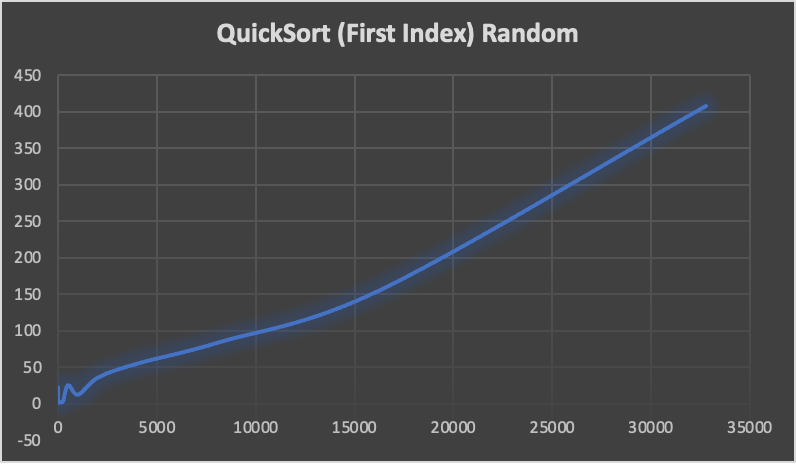


**Quick Sort, First Index:**

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).

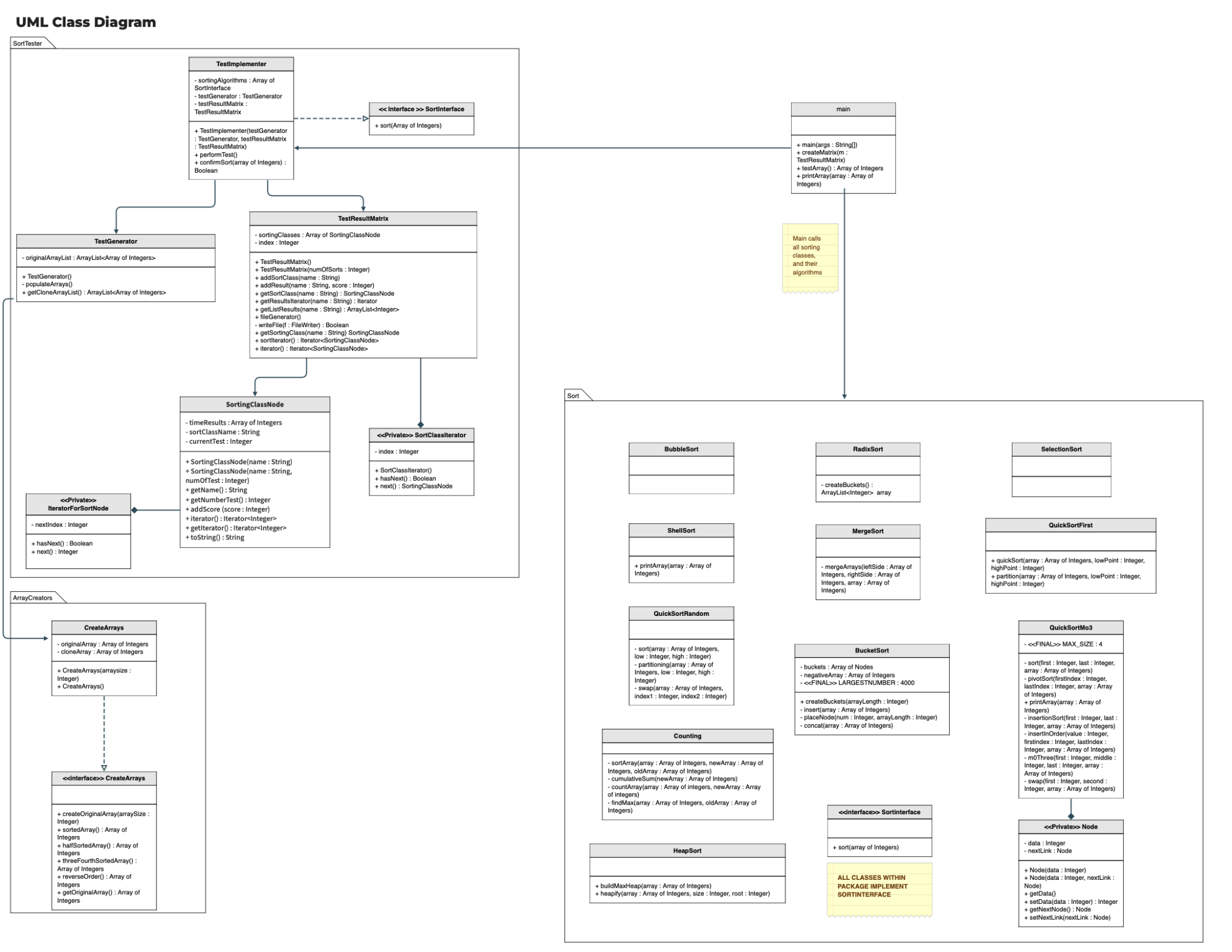
Sort is our package that contains all the different sorting algorithms that will be tested. All of the different classes within were required to implement the SortInterface, that is it required them to have a method called sort(). Initially, our idea was hopefully to make it easier to call the different 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** |
|  |  |
|  |  |
|  |  |