**Final Report: Implementing and Testing Various Sorting Algorithms**

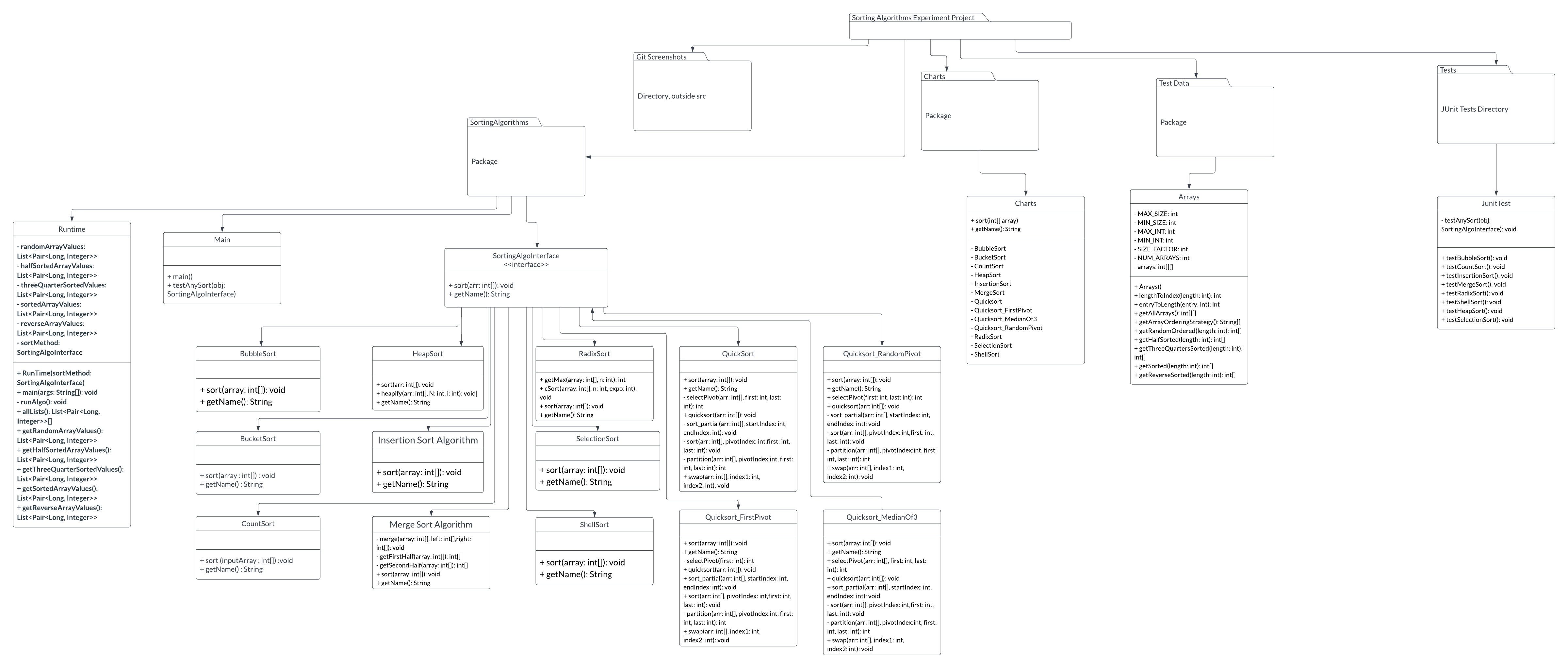
**Team Maroon 4**

Project Manager: Edale Miguel   
Design Manager: Rojee Koju   
Development Manager: Kelvin Rajbhandari   
Version Control Manager: Rory Hackney

Part 1: The Experiment

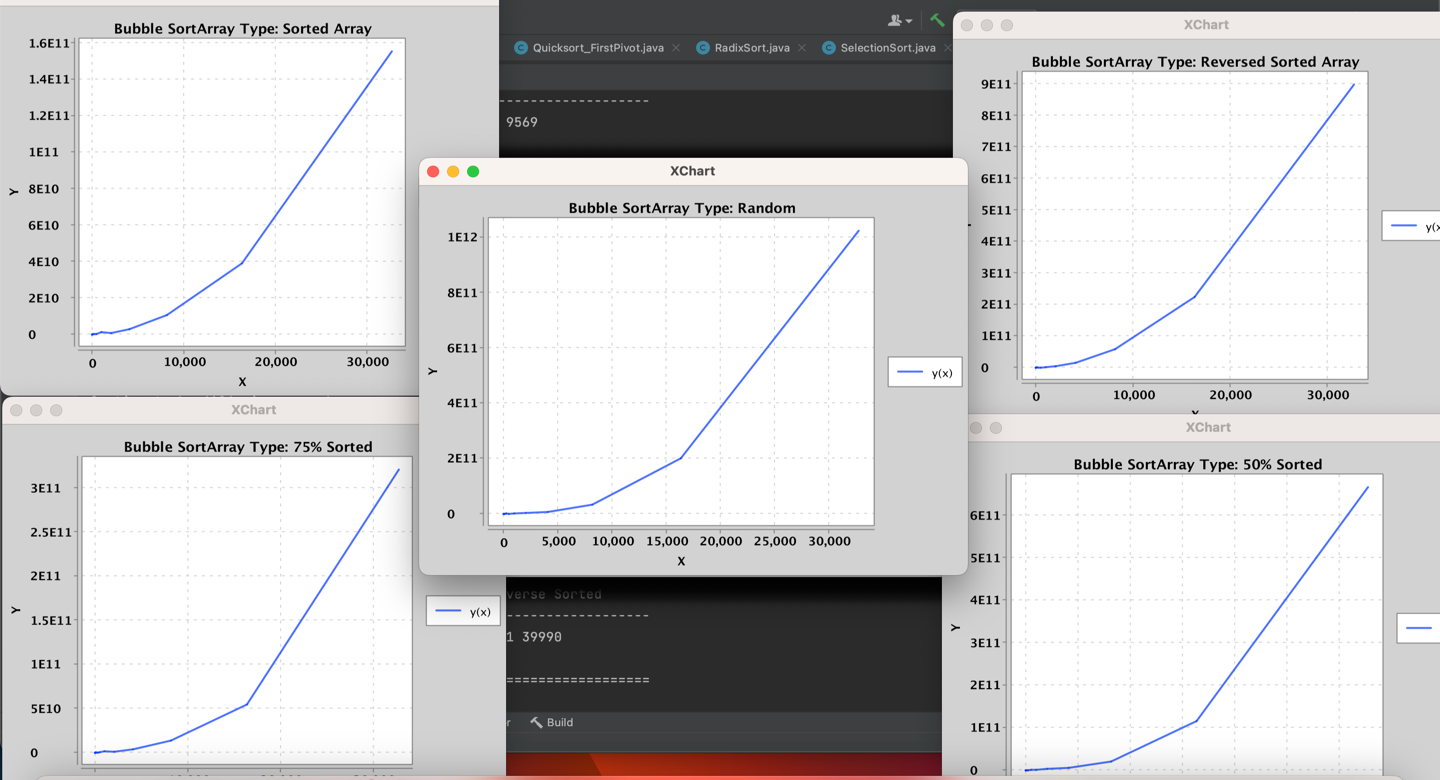
As a team, we implemented different sorting algorithms and tested their runtime against arrays of integers in various states, including randomly ordered, 50% sorted, 75% sorted, 100% sorted, and reverse 100% sorted.

We decided to measure the efficiency of our algorithms using time of execution in microseconds because we were interested in the real-time performance of the sorting algorithms using our own computers to see how they might perform in the real world.



Part 2: The Results

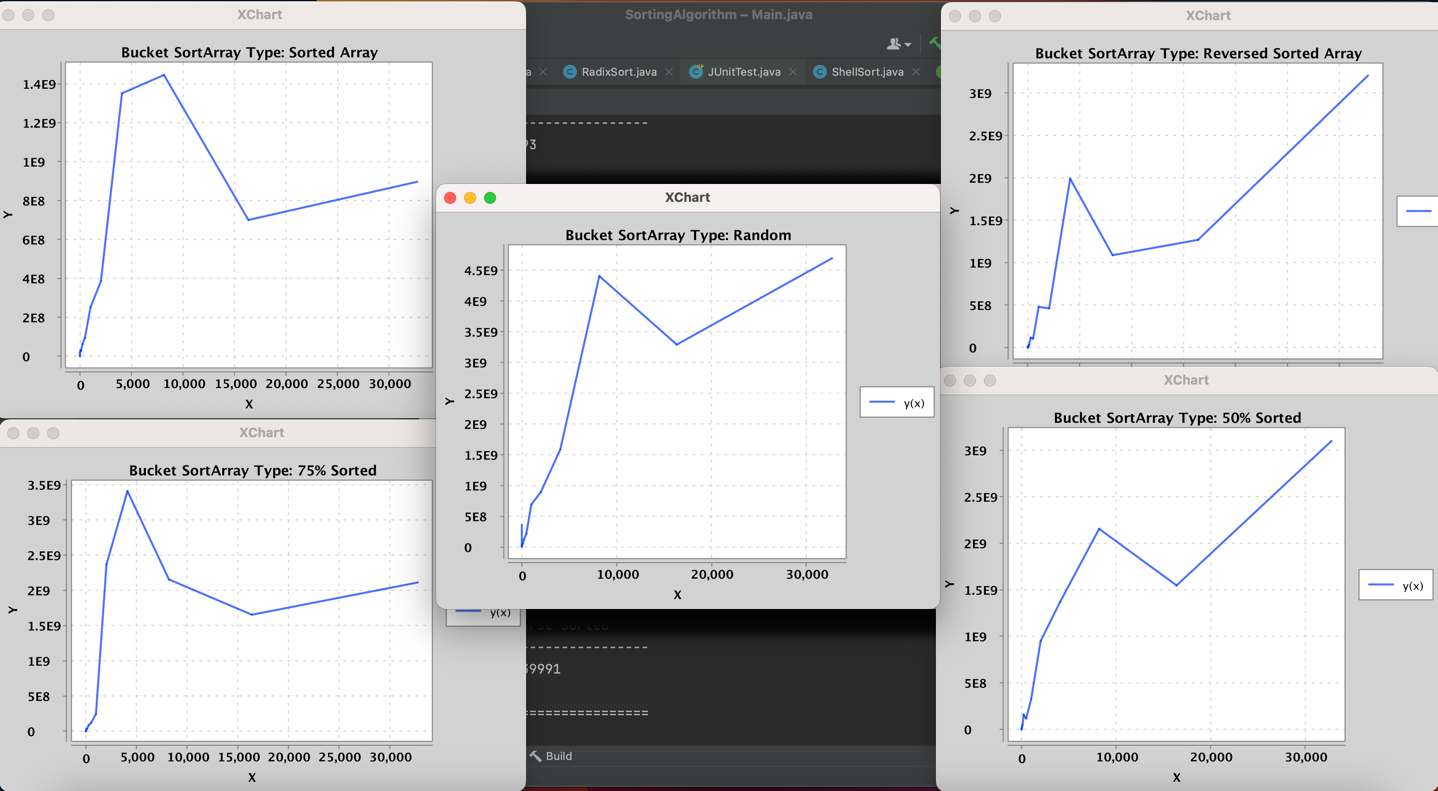
Do your findings meet theoretical expectations? Why did you choose the implementations that you did for the sorting algorithms? Why did you reject other implementations that you researched? All of these questions must be addressed by sorting algorithm. Compare the performance of all the algorithms in words. Which were fastest for smaller data sets and which for larger? Discuss what sort of tradeoffs each algorithm makes and what algorithms are suited for what type of applications, such as large systems, distributed systems, embedded systems, mobile systems, etc.



**Bubble Sort**

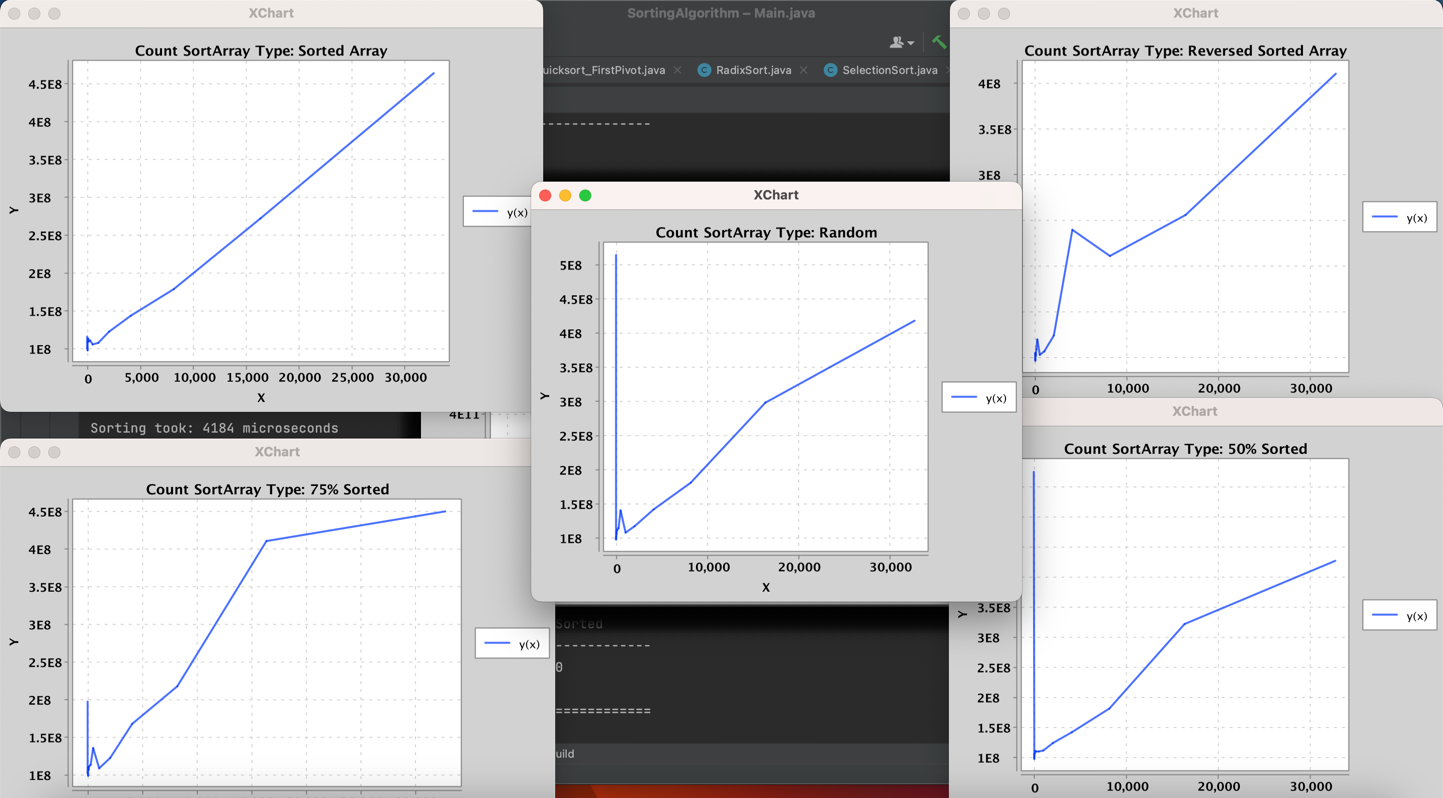
Bubble Sort is a stable algorithm that swaps adjacent pairs if a larger number is to the left. Its theoretical worst-case performance is which occurs when an array is reverse sorted. Its best case is when it's a fully sorted array and there are no swaps to execute. Due to its large worst-case runtime, it should not be used for large datasets. However, its simplicity and ￼ space complexity is the tradeoff and an asset for learning sorting. It also works great for a small dataset or in embedded systems where time is not a problem. Implementation was done with the most straight forward and simplistic approach using 2 nested loops. Comparison of values happens inside the inner loop, swapping them as needed in each cycle of the inner loop which speaks for the constant space complexity.  
All in all, one of the lower ranking sorting methods when it comes to speed/time complexity.

Our theoretical results were the same for all sort orders: O(n2)



**Bucket Sort**

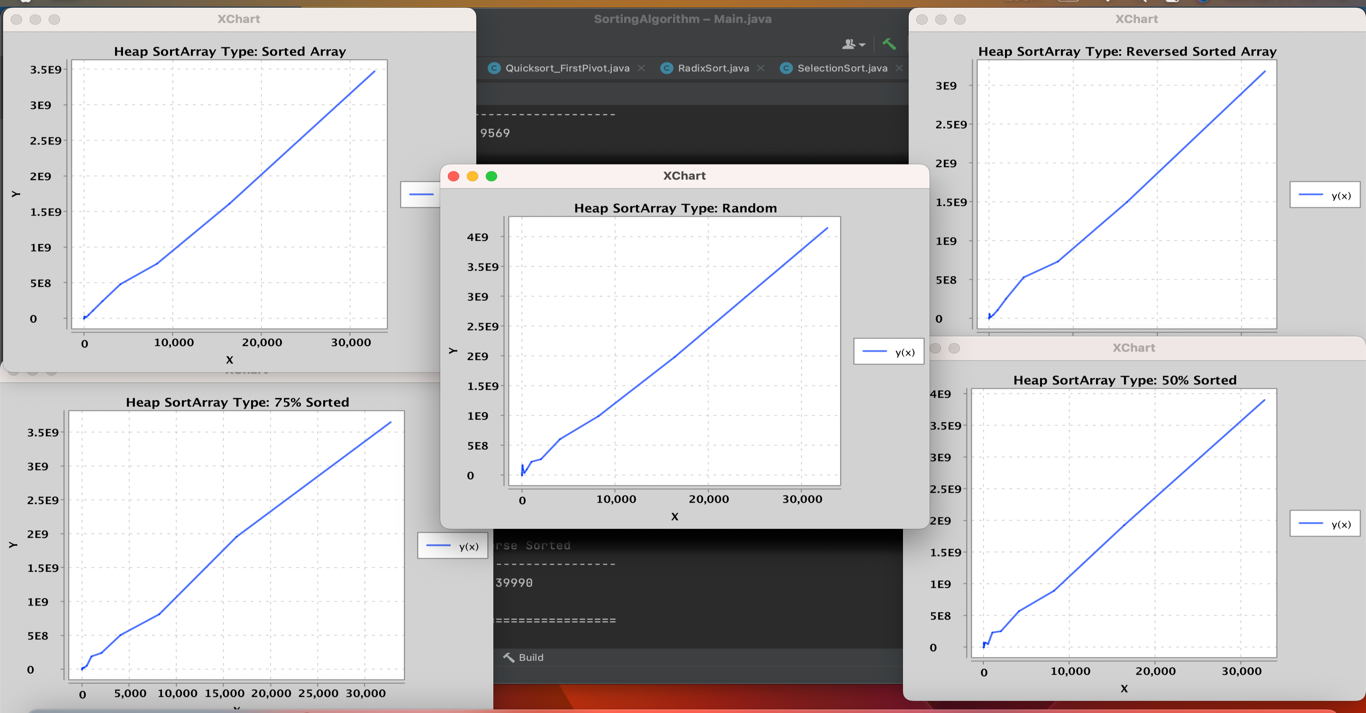
The Bucket Sort is an algorithm that sorts an array/list using buckets to categorize the elements. The number of buckets would depend on the implementation. Each bucket would store elements with a uniform incremental range. Once elements have been placed into their respective buckets, the buckets themselves would be sorted individually to store their elements in order. Finally, all elements from each bucket are concatenated in order to result in a sorted array. The best case in time complexity would be and worst case being .  
The worst-case space complexity would be if algorithm is being implemented with the number of buckets being utilized to the number of elements to be sorted.   
We stayed away from this approach as it isn’t a very optimal situation where each bucket only holds one element each and this in a way defeats the purpose of bucket sort. The goal is to distribute elements evenly among buckets to achieve a balanced workload. Also, the number of buckets directly affects the space complexity and the overhead associated with managing each bucket. Therefore, we decided to have the number of buckets = ￼ which ideally strikes a balance between having enough buckets to distribute the elements evenly while minimizing overhead.



**Count Sort**

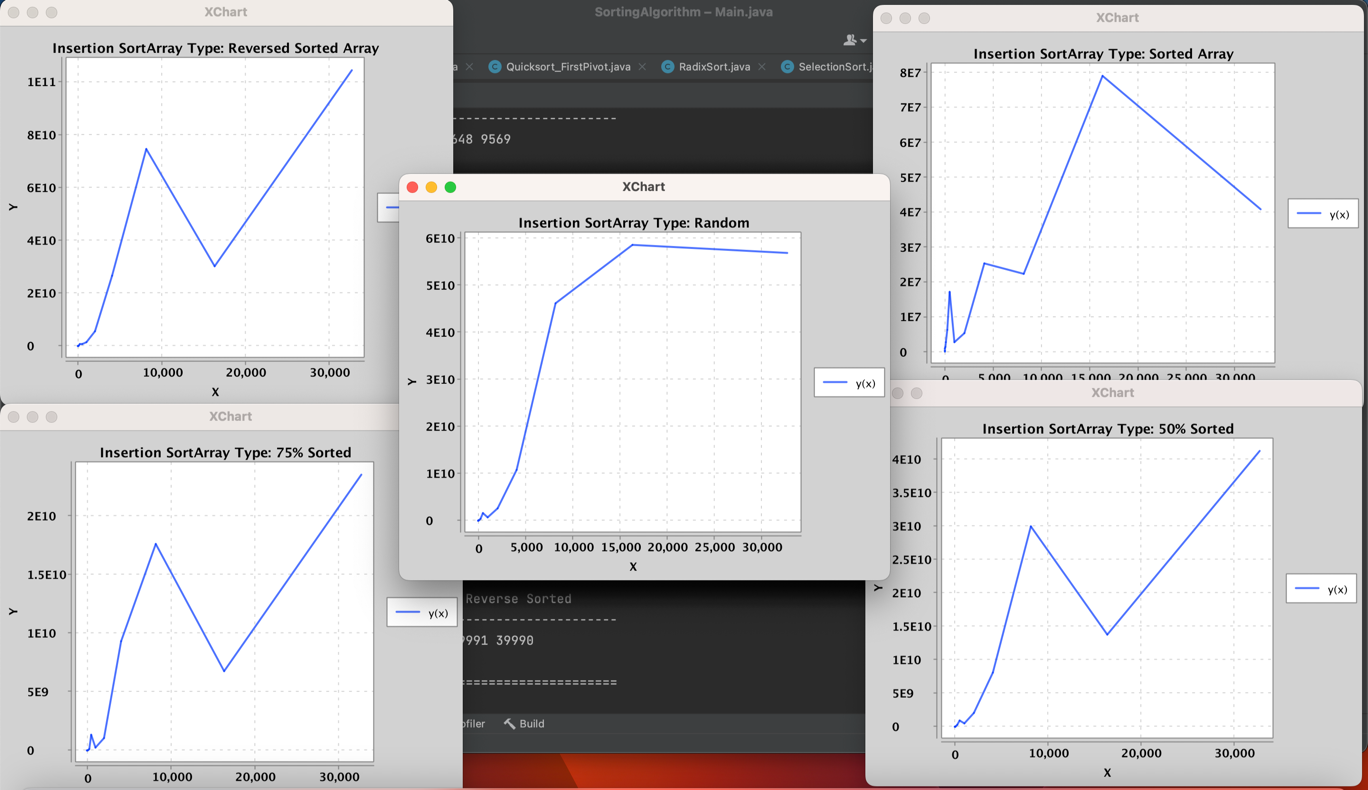
The Counting Sort is an algorithm that performs sorting by counting objects having distinct key values. Counting sort counts how many objects have each unique characteristics and then calculates where each object should go in the final sorted sequence. Counting sort is O(n+k) at its best, average, and worst-case complexity because the algorithm goes through (n+k) times, regardless of how the elements are. Counting sort is effective when range is not greater than number of objects to be sorted and are placed in the array. Our theoretical results were different for all sort orders.

In a reversed array, where elements are arranged in descending order, counting sort remains efficient. However, it may not take full advantage of its linear time complexity in this case. In 75% of the array, counting sort can still perform efficiently. The counting phase will identify the occurrences of each element, and the reconstruction phase will arrange them accordingly.



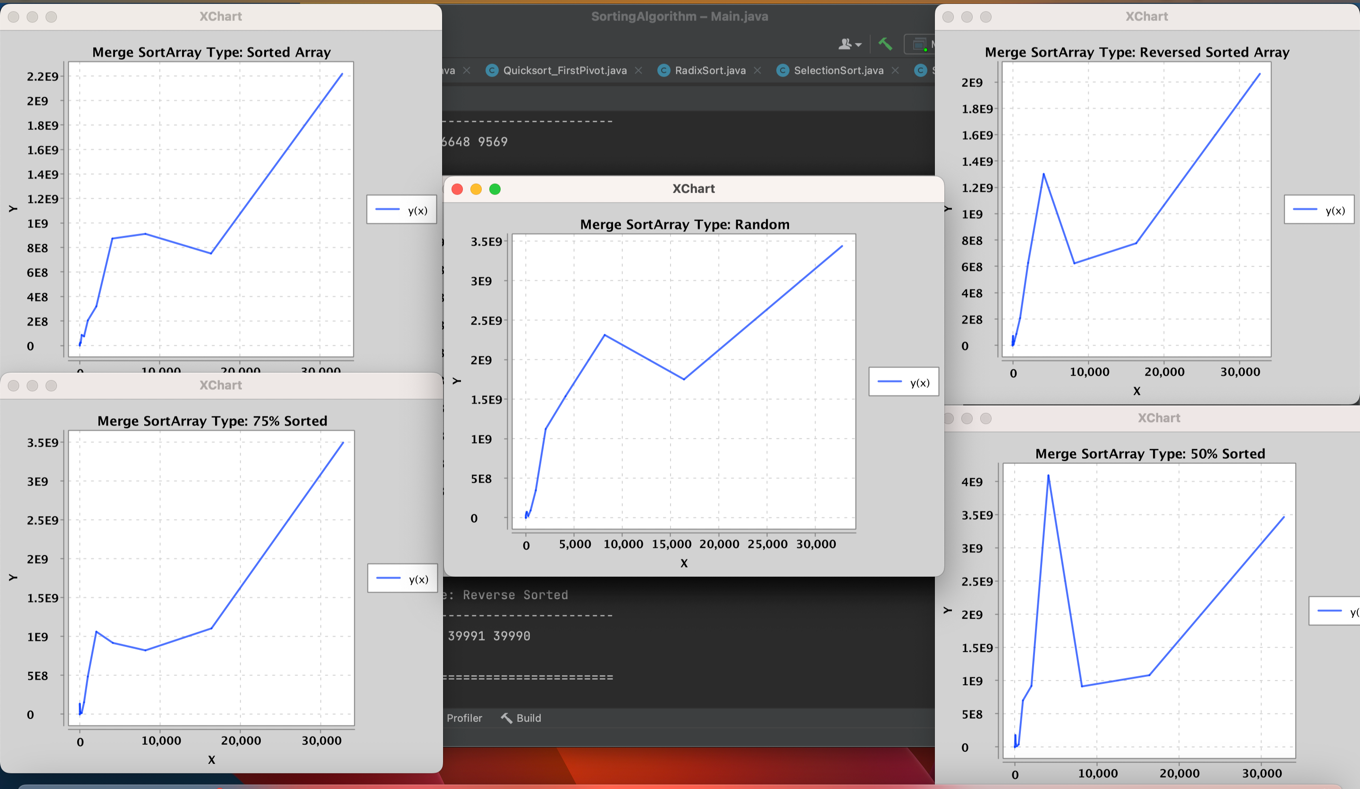
**Heap Sort**

The Heap Sort is an unstable algorithm that places an array into a max heap, then removes the max item from the root, places it at the end of the array, and reheaps the remainder until it’s a maxheap again. This repeats until the array is sorted. When done in place, turning the array into a heap, it does not require auxiliary memory. Heap Sort is O(n log n) for worst, average and best cases, so it is best used for sorting large numbers of items compared to some quadratic time complexity algorithms. We got kind of similar graph of all sort array except for Random. For revered array, the build heap take longer, as each element needs to be moved around to create the heap structure. The efficiency of heap sort is affected by the initial order of elements. If the array is already sorted, extracting the maximum repeatedly involves moving the root element to the end, resulting in a sorted array.



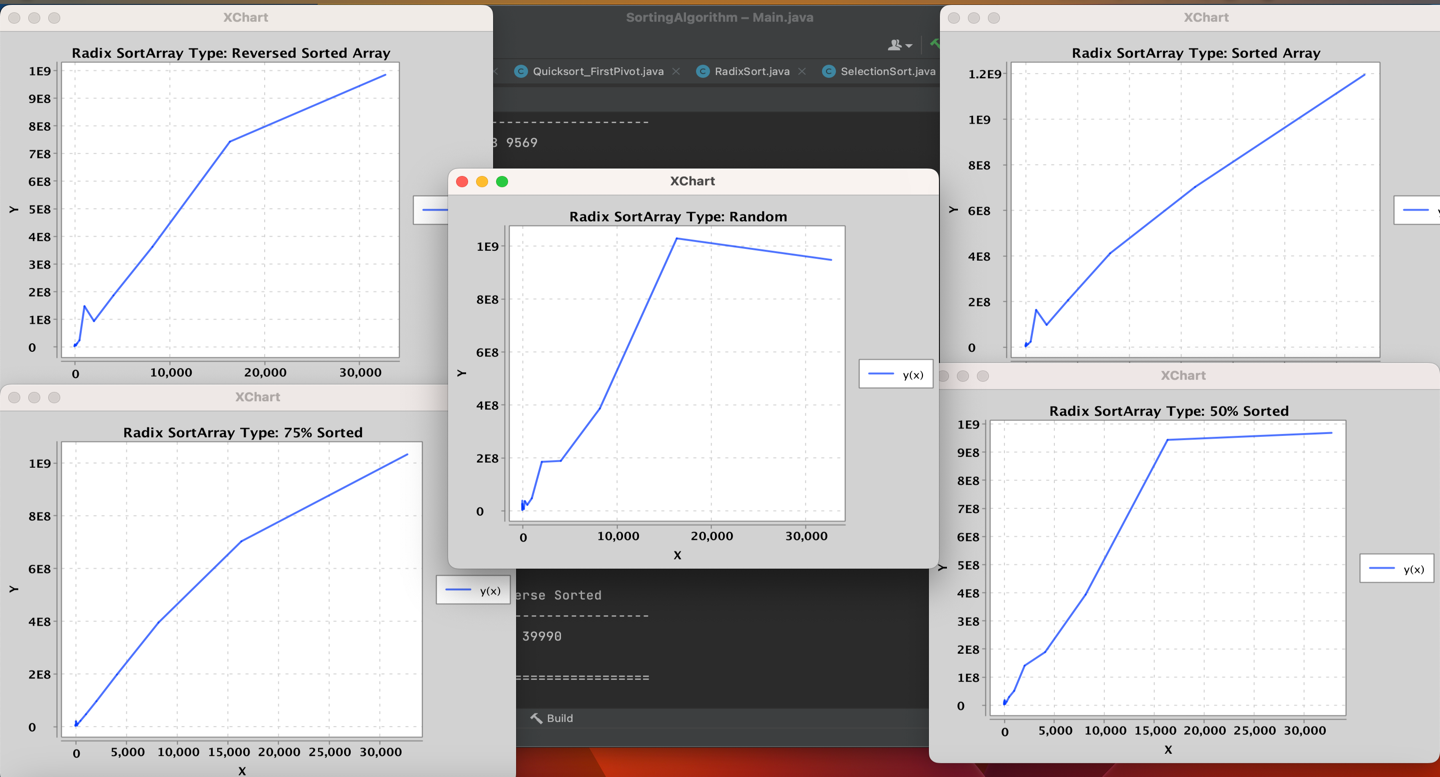
**Insertion Sort**

Insertion Sort is a straightforward way of arranging numbers in a list by repeatedly picking one number at a time and placing it in its correct order. The expected time it takes, on average, is O(n), which means it might not be the fastest for large lists. However, it has its advantages, especially for small lists or when the numbers are nearly in order already. The code provided efficiently implements Insertion Sort by going through the list and inserting each number where it belongs, making sure the list stays sorted. It's a good choice when dealing with small lists or when the numbers are mostly in order already. Its simplicity makes it easy to understand. However, for really large lists, other sorting methods like Merge Sort might be more suitable due to their faster average times. Insertion Sort is great for small lists or mostly ordered data, and this code does the job efficiently by placing each number in its right spot step by step.



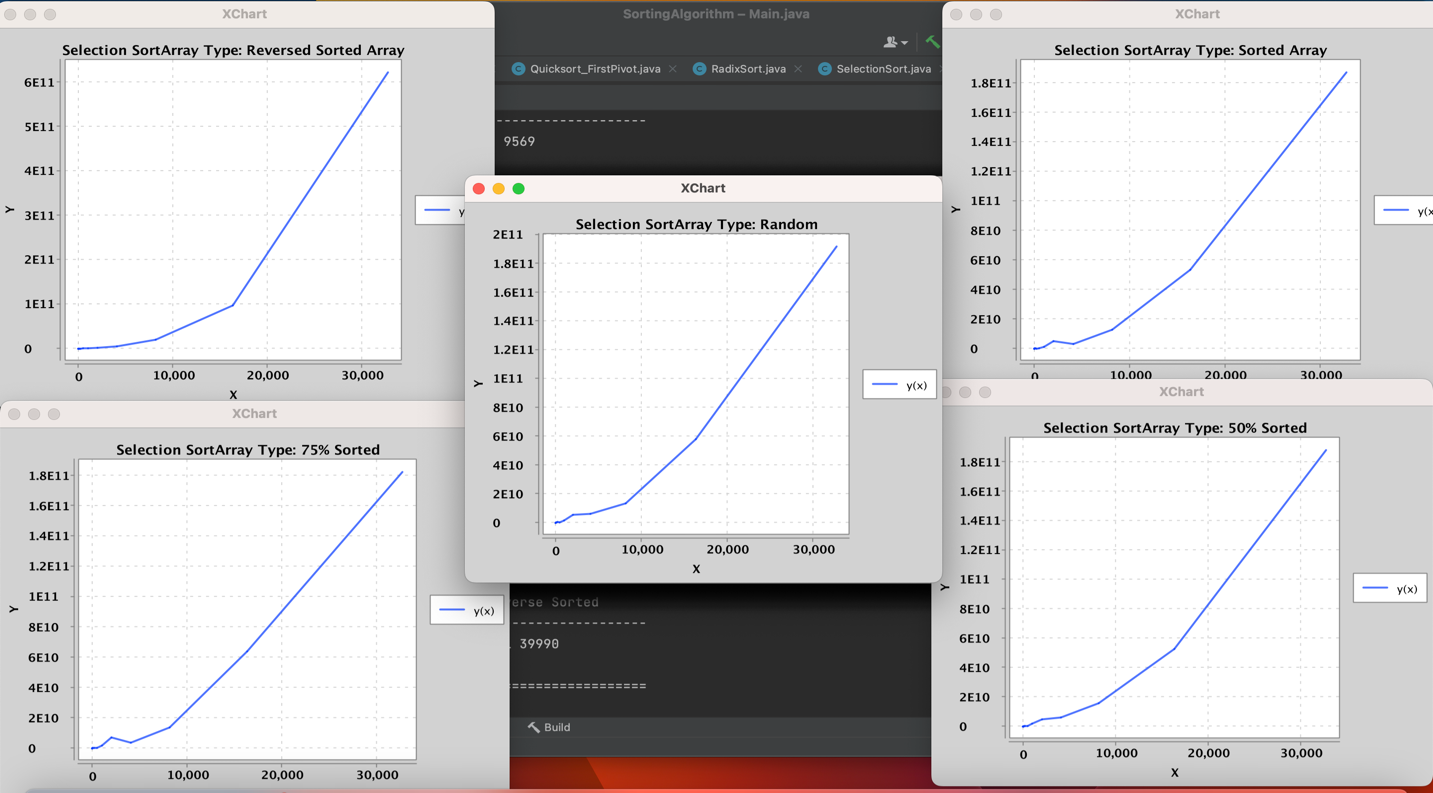
**Merge Sort**

Merge Sort is an efficient sorting algorithm that works by dividing the array into smaller parts, sorting those parts individually, and then merging them back together into a sorted array. Its expectation, or average performance, is O(n log n), making it efficient for sorting large datasets. Merge Sort's main advantage lies in its predictable performance and stability. It maintains its efficiency even with a large number of elements and is suitable for sorting linked lists as well as arrays. However, Merge Sort requires additional memory space for merging the smaller parts, which might not be ideal for memory-constrained environments. The code efficiently implements Merge Sort by breaking down the sorting process into smaller, manageable parts and merging them together systematically. It recursively sorts the array by dividing it into halves, sorting each half, and then merging them. This implementation ensures that the array is sorted correctly while maintaining efficiency. Merge Sort is best used when stability, predictable performance, and efficiency on larger datasets are important considerations.



**Radix Sort**

Radix Sort works by organizing numbers based on their digits, from the least significant digit to the most significant one. Its efficiency depends on the size of the numbers and how many digits they have. The expectation, or average time it takes, of Radix Sort is O(n) and its generally good, especially for sorting large sets of numbers with a limited range of values. However, Radix Sort requires extra memory space, which might be a drawback in memory-constrained environments. It's best used when dealing with numbers of fixed size or a small range, and when the time it takes to sort is more important than saving memory. The code efficiently implements Radix Sort by using a method called cSort, which helps in sorting the numbers based on their digits. This code iterates through each digit position of the numbers, sorts them accordingly, and combines them to get the final sorted array. Overall, Radix Sort is a practical choice for sorting integers efficiently, especially when their range is known, and memory usage isn't a primary concern.



**Selection Sort**

Selection Sort is an algorithm that repeatedly selects smallest (or largest) element from unsorted portion of list. Once it found, it swaps this smallest (or largest) element with the first element in the unsorted portion. This portion continues with the unsorted portion getting smaller each time, until the entire list is sorted. The time complexity of selection sort is O(N square 2). Disadvantage of selection sort Algorithm is that it has time complexity of O(N^2) in the worst and average case. The results of all sort array look same for selection sort which is the result as expected. Selection sort can be suitable for sorting small datasets where its simplicity and ease of implementation might outweigh its inefficiency. We got similar data for selection sort. This shows that the algorithm performs a similar number of comparisons and swaps for each type of input pattern. While the actual number of operations may vary, the overall trends in time complexity remain consistent.



**Shell Sort**

Shell Sort is an extension of Insertion Sort that begins with selection a gap sequence between elements to be compared and swapped as needed. It performs multiple passes over the array using the gap, decrementing it till the gap is 1 which lastly leaves a final standard Insertion Sort. Over each pass executed, the algorithm considers subarrays created by selecting elements that are separated by the current gap being utilized and performs independent Insertion Sort on those subarrays. As the algorithm progresses and the gap sequence decreases, the array becomes more sorted and the next subsequent passes benefit from the partial sorting as a result. By the time the gap decreases to 1, the final iteration of Insertion Sort efficient.  
Shell Sort has a worst-case time complexity of , which occurs when the array is in a reverse-sorted sequence and also depends on the gap sequence chosen.  
Best case would be , which occurs when the initial order of the elements is such that the chosen gap sequence can effectively reduce or even eliminate the number of swaps needed.  
  
Shell Sort doesn’t have the best worst-case time complexity and is similar to other quadratic sorting algorithms but it improves significantly when the array is partially sorted.

**Quicksort – Various Pivot Schemes**

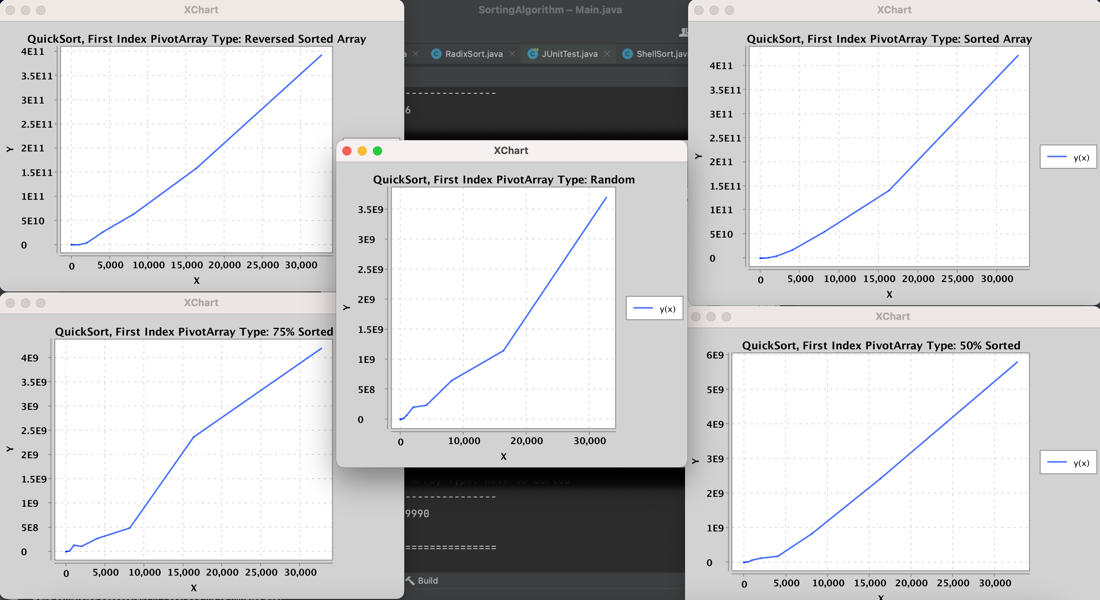
Quicksort is a sorting algorithm that recursively selects a pivot, then places it into its final spot by sorting smaller items to the left and larger items to the right. When a good pivot is usually chosen such that each time the array is roughly divided in half, this can be a very efficient algorithm, of O(n log n), but if a bad pivot is chosen every time, such as the smallest or largest item, it can be O(n^2).

Quicksort also has a stack overhead due to the recursive calls, which with bad pivots can get very deep. At first, we had trouble collecting valid data, because certain arrays, particularly long ones that were already sorted where the pivot chosen was the first element, ended up with very deep recursion. As you can see in the charts below, the **First Element as Pivot** will perform badly (steeper than the other pivots), **Random Element** better, and **Median of Three** (choose median as pivot between 3 elements) best of all.

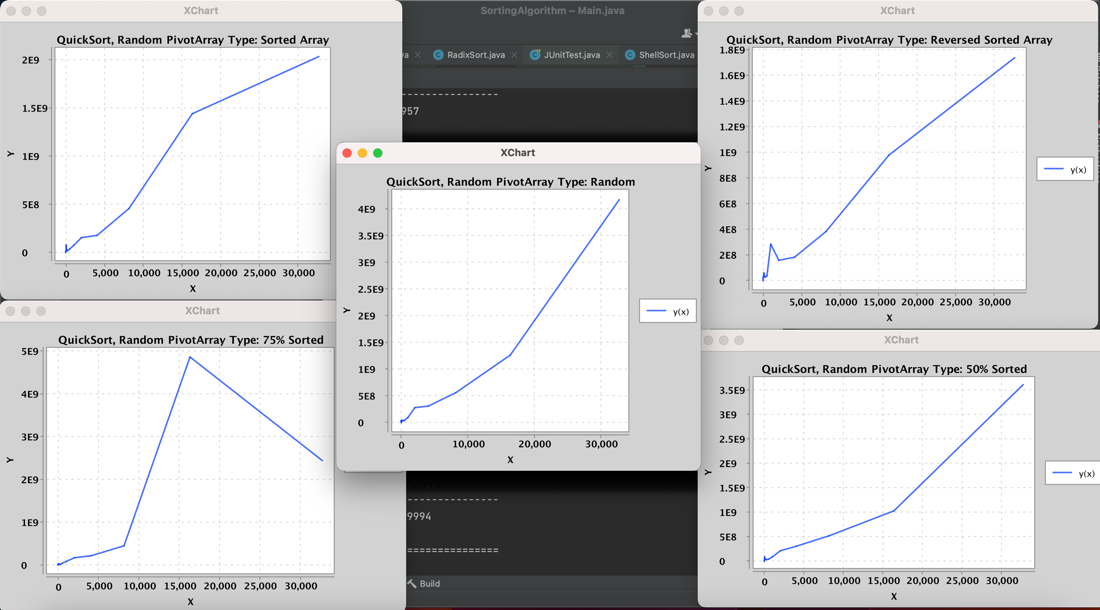
For implementation, we split the Quicksort into a base class, then three more classes that uniquely defined the pivot selection scheme and made other adjustments. This way, we were able to implement the interface for each pivot scheme and test each implementation separately. We also created various helper methods to help split up the complexity of the algorithm and provided static methods as options for sorting without instantiating the class, as well as a partial sort algorithm to allow us to easily sort the 50% and 75% sorted arrays for testing data in the Arrays class.

Quicksort is best used for larger datasets, not smaller ones, with a good pivot scheme, where sorting stability is not important. Quicksort is one of the fastest sorting algorithms available, so it is often a good option when these requirements are met.

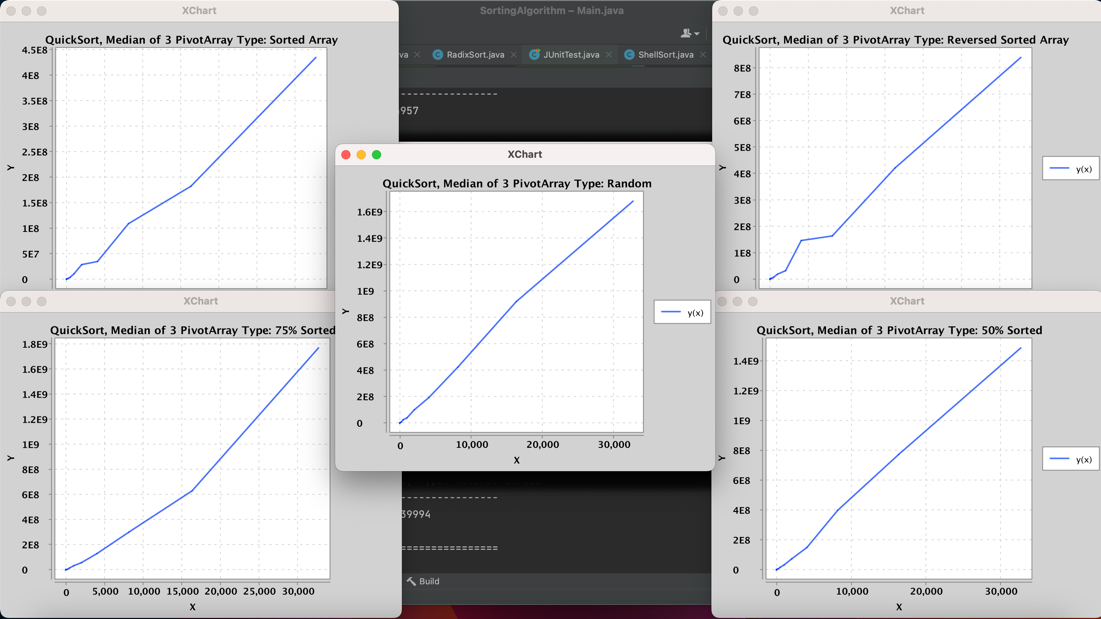
**Quicksort – First Element As Pivot**



**Quicksort – Random Pivot**



Quicksort – Median of 3



These are some of our attempts that we tried to do using Javafx.











