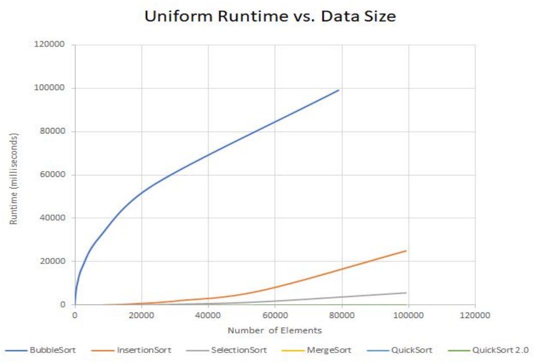
In this experiment, all of the sorting algorithms contain my own original implementations based on explanations of how these operations execute. These implementations were derived without any assistance of pseudocode. As such, it is possible that there are some minor deviations that can lead to a slight increase in overhead cost. However, these algorithms were finetuned and tested thoroughly for optimization. BubbleSort, InsertionSort, SelectionSort, and MergeSort algorithms contain no optimization techniques to improve its runtime so much of the focus was centered around QuickSort. The QuickSort algorithm presented in this report contains implementations of selecting a pivot in the middle of the list or using the recursive median of medians algorithm in order to compute the pivot. Within this experiment, I can conclude that selecting a pivot within the middle of the dataset yields a faster runtime compared to computing the median in linear time. This choice yields a worst-case time complexity of $\Theta(n^2)$ for quicksort compared to the $\Theta(nlogn)$ worst-case time complexity when selecting a pivot using the median of medians approach. However, iterating through the dataset recursively for a list of medians and partitioning the list recursively until the median is located incurs a significant overhead cost. This overhead cost can even multiply quicksort's runtime when operating on relatively large datasets despite mathematically being proven optimal regardless of the size of subsets used in the median of medians approach i.e size of 5, 7, or 11. As such, using a simple approach to select a pivot was apt for this experiment. Any meaningful comparisons between other algorithms will be used using that simple approach. The code implementation for both can be found in the source code and results are shown in this experiment.

This experiment was conducted by comparing the performance metrics i.e size of dataset and degree of sortedness against the selected algorithm's time and space complexity for each algorithm and each dataset. The degree of sortedness was computed by calculating the number of inversions needed to be made in the dataset in order to sort the dataset and divided it by $n(n-1)/2$ where n represents the size of the dataset. This represents the number of unique subsets of size 2 that are in the dataset (an inversion compares 2 elements and thus, can be represented as a subset). This computes a number between 0 and 1 where 0 means the list is already sorted whereas 1 means the list is completely disordered (in reverse). The time and space complexity were computed using System.nanoTime() and the RunTime Java object respectively by computing the difference in time and memory usage before and after the algorithm's execution. There are some large and interesting variations in terms of memory usage due to the nature of recursion in mergesort and quicksort which will be discussed later.

There are 4 datasets that were used in this experiment, 2 of which were drawn from distributions and the other 2 were drawn from real datasets which are included in this submission. The synthetic datasets use a Uniform and Gaussian distribution to make more meaningful comparisons. Uniform distribution implies that the probability of an element's occurrence in the dataset is the same whereas Gaussian distribution uses a bell-shaped curve in which the probability of x is the highest towards the median of the data. These variations will enable us to make meaningful comparisons and analyze the strengths and weaknesses of each algorithm. The other 2 datasets that were used were derived from datasets regarding the economic ranking of each nation in the world from 1970-2018 whereas the other one records the AP performance levels (in this case the number of test takers) in respect to their testing location, ethnic group, and socioeconomic status. The former is more closely related to a uniform distribution, but is not subjected to probability whereas the other dataset has a more dynamic range of values that is predicated on numerous different variables. As such, both datasets' characteristics can derive some interesting results and conclusions.

Throughout this experiment, the most generalized conclusions that could be drawn is bubble sort having the worst time complexity and bubble sort, insertion sort, and selection sort having worse time complexity than both merge sort and quick sort regardless of the pivot selection in this experiment. This is because the average time complexity of bubble sort, insertion sort, and selection sort is $\Theta(n^2)$ whereas quick sort and merge sort have a time complexity of $\Theta(nlogn)$. This is because of the recursive nature of both merge sort and quicksort compared to the iterative nature of bubble sort, insertion sort, and selection sort. Bubble sort requires an n number of comparisons between each element in order to swap them adjacently. Insertion sort iterates through the entire list and swaps a selected element with previous elements in the list until it is in order. Selection Sort has to iterate through the entire list to find the minimum of the list and then iterates again to swap with previous elements until it reaches the jth location where 0 <= i < n and i < j <= n. Merge sort and quick sort however, exponentially divides the list in half by recursing through two different sublists which reduces the time complexity. Merge sort however, must iterate over all elements after the recursive calls have finished to merge the two halves of the lists into a new sorted list. Quick sort has to iterate through all elements in the sublists to sort the pivot in the right place. There are n pivots and quicksort sorts through each pivot. This extra linear work gives both merge sort and quicksort a time complexity of $\Theta(n^2)$. Bubble sort has the worst time complexity of them all because insertion sort and selection sort algorithms inherently retains a sorted sublist as it continues sorting the list whereas bubble sort does no such thing.

Within this experiment, the algorithm's performances were first measured in the Uniform dataset as shown below:



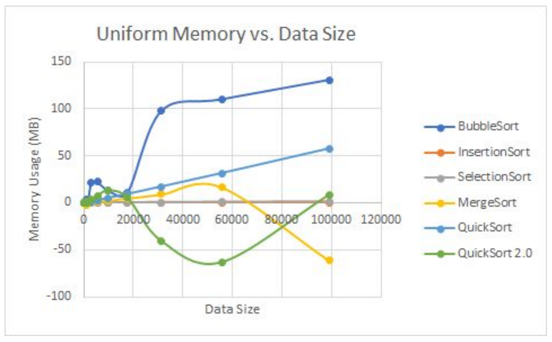
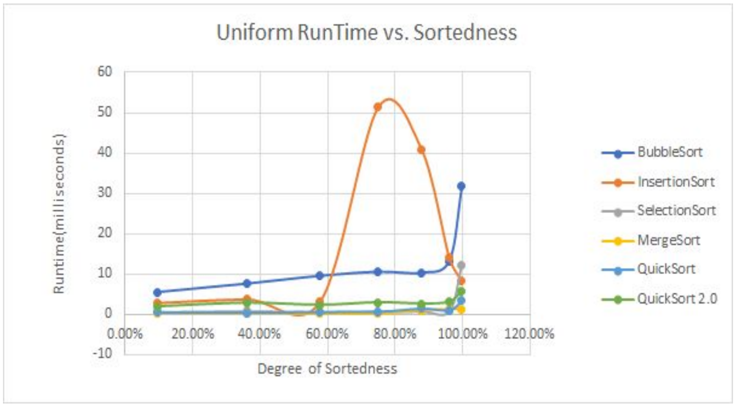


Figure 1a) Runtime vs. Data Size

Figure 1b) Memory usage vs. Data Size



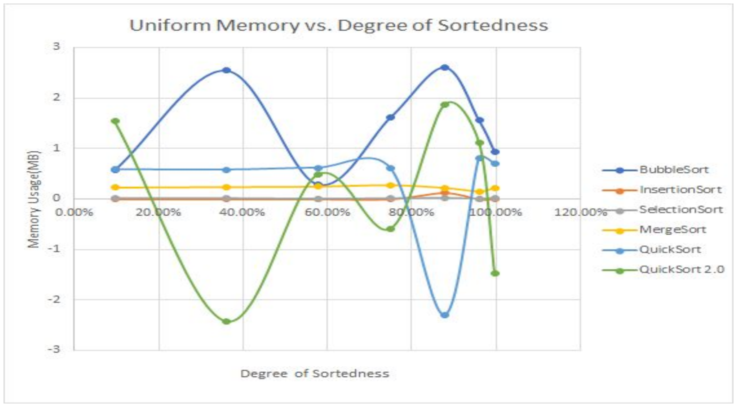
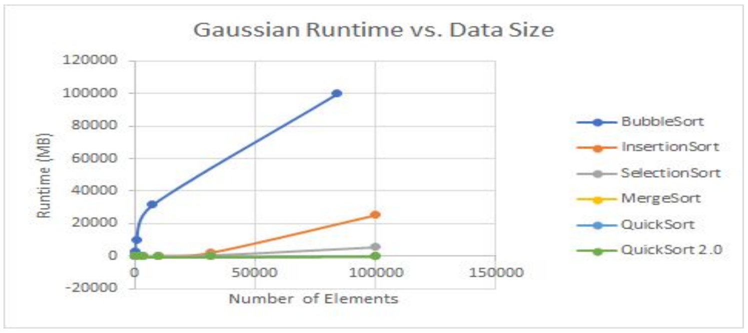


Figure 1c) Runtime vs. Degree of Sortedness

Figure 1d) Memory vs. Degree of Sortedness

Because of how the algorithms work, the runtimes for bubble sort, insertion sort, and selection sort are explicitly shown to be much higher than merge sort and quicksort regardless of time complexity as shown in Figure 1a. Bubble sort doesn't retain a sorted sublist and makes more comparisons and swaps than both insertion sort and selection sort. Selection sort however, has an even greater time complexity than insertion sort because selection sort because the speed of copying integers to another memory address is slower than the cost of making comparisons and selection sort only requires to perform swaps between 2 integers whereas insertion sort requires multiple. Merge sort and quicksort barely increase in respect to data size compared to the other sorting algorithms, but the data actually concludes that merge sort was faster because the worst case time complexity of merge sort is $O(nlogn)$ whereas the worst case for quicksort is $O(n^2)$. This remains to be true in Figure 1c where merge sort is evidently faster than even quicksort regardless of pivot selection (because median of medians incurs a significant overhead cost). However, insertion sort was even slower than bubble sort when 60-80% of the list was unsorted when data size was constant. It was only until the list was almost sorted initially that insertion sort outperformed bubble sort and converged with the runtime of selection sort.

Figures 1b and 1d demonstrate that insertion sort and selection sort has no memory footprint or a very marginal memory footprint due to its constant space complexity regardless of data size. Bubble sort had the highest space complexity whereas merge sort and quicksort using median of medians, even resulting in having a negative memory footprint presumably because of the recursive stack deallocating memory and median of medians requires recursive calls on top of quick sort. Quicksort with a simple pivot approach had a more linear increase in space complexity in respect to data size. However, it resulted in more memory being deallocated after execution when the list was heavily unsorted and became more constant and comparable to other algorithms when the list's degree of sortedness was closer to zero. The inverse was true for quicksort when using median-of-medians approach. It is evident that for a uniform distribution, merge sort is the best in respect to optimizing time complexity and space complexity as both are less dynamic and volatile compared to quicksort which is comparable in time complexity, but volatile in memory usage.



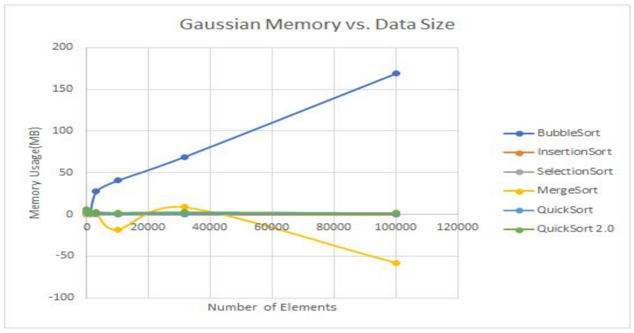
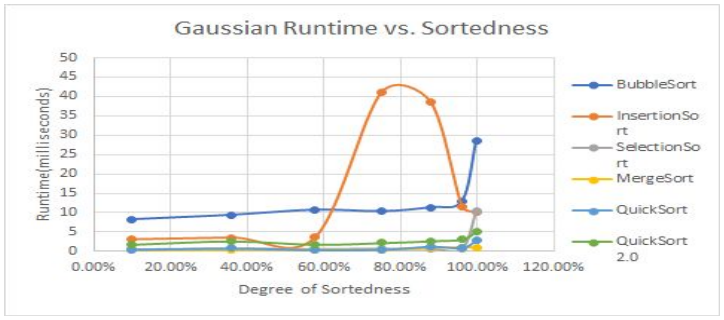


Figure 2a) Runtime vs. Data Size

Figure 2b) Memory vs. Data Size



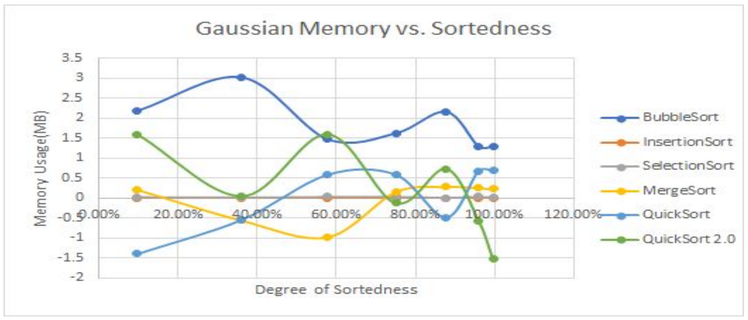


Figure 2c) Runtime vs. Degree of Sortedness

Figure 2d) Memory vs. Degree of Sortedness

Figure 2a and 2c demonstrates a similar trend that was encountered in Figures 1a and 1c regardless of the difference in the distribution of data. In Figure 2c, the difference in runtime between merge sort and both variations of quick sort is more pronounced compared to Figure 2a similar to how it was for Figure 1c relative to Figure 1a. As such, we can conclude that merge sort operates even better regardless of the degree of sortedness compared to quicksort where quick sort performs noticeably worse when nearly all of the elements are disordered. Figure 2b also establishes a different trend compared to what was the case for uniform distributions. The memory footprint for nearly all of the algorithms was the same except for Bubble sort which was significantly higher. However, merge sort seemed to dereference objects that were being used when the data size was even greater. Figure 2d demonstrates the same case where garbage collection occurred during the execution of merge sort when the list was nearly sorted whereas both variations of quicksort had a more parabolic pattern. Quicksort's memory usage actually decreased the more ordered the list was whereas quicksort using median of medians approach began to decrease and then increase. For a Gaussian distribution, mergesort and quicksort without using median of medians because both are the highest in respect to time complexity, but the latter forces garbage collection regardless of size and degree of sortedness (for the most part). Selection sort proves to also be useful due to its absence of memory usage as well as its runtime complexity being closer to mergesort and quicksort's.

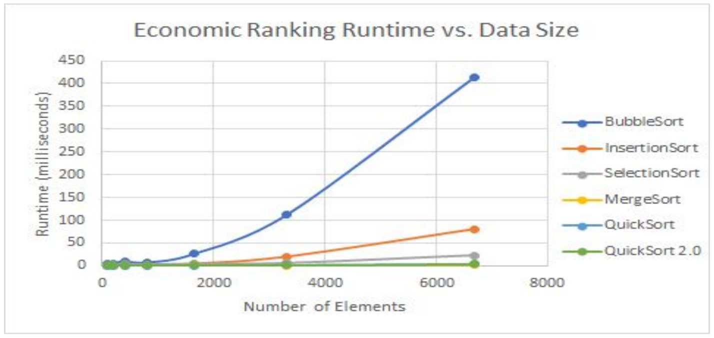


Figure 3a) Runtime vs. Data Size

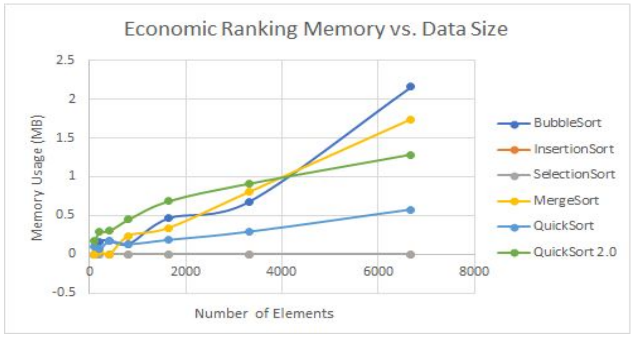


Figure 3b) Memory vs. Data Size

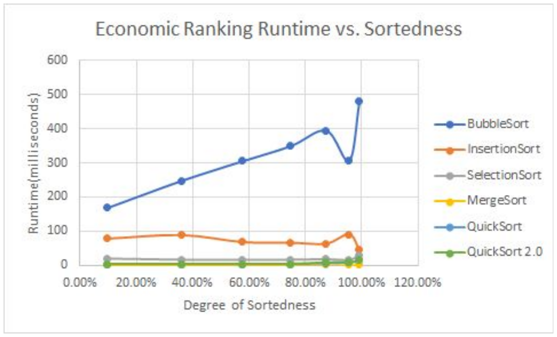


Figure 3c) Runtime vs. Sortedness

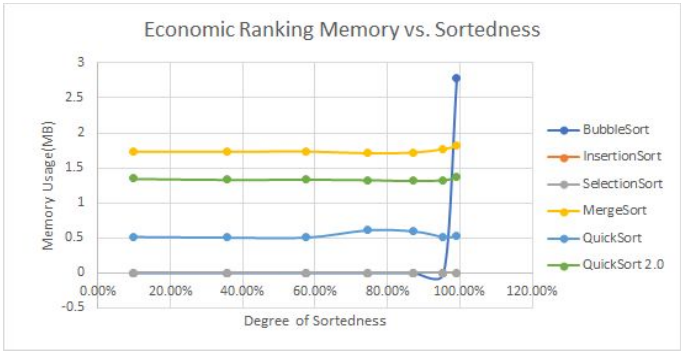
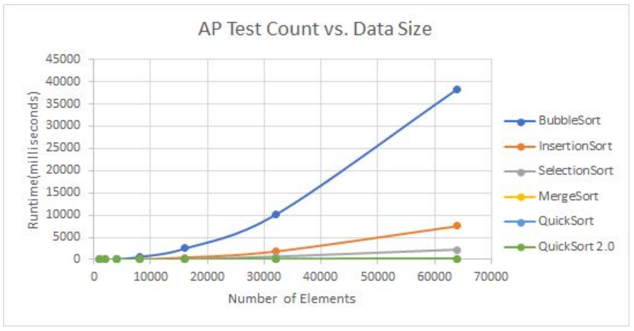


Figure 3d) Memory vs. Sortedness

Figure 3a demonstrates a similar trend as previous datasets. However, insertion sort's runtime remained relatively low and constant regardless of the degree of sortedness. This is presumably due to the fact that this dataset studies the same statistic over many years, so repeated occurrences of values that are deterministic resulted in a very stable runtime. In respect to both merge sort and quicksort, merge sort was actually slower than quicksort as the data size increased presumably because of the inherent nature of the dataset even though quicksort using median of medians approach was slower. Merge sort's time complexity relative to the degree of sortedness still remained lower. In Figure 2b, every algorithm had a linear relationship between memory usage and the size of data. Merge sort actually used more memory than insertion sort, selection sort, and quicksort and actually converged with the memory usage found in bubble sort in some cases. In Figure 2d, the memory usage of memory sort consistently remained the highest regardless of the degree of sortedness despite using quicksort with median-of-medians within this analysis. It is clear that while merge sort still has the best time complexity of $\Theta(nlogn)$ with a worst case of $O(nlogn)$ without the significant overhead of median of medians, it also has the worst memory footprint. Therefore, quicksort is the superior algorithm in order to optimize memory usage over many elements while mergesort remains the best in regards to time complexity.



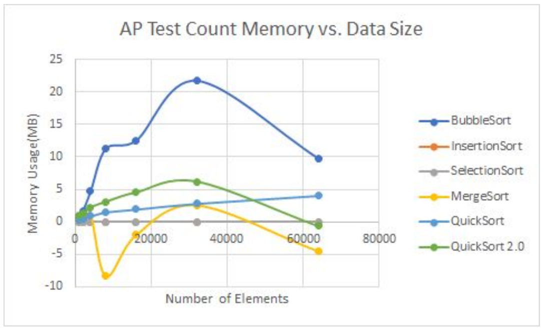
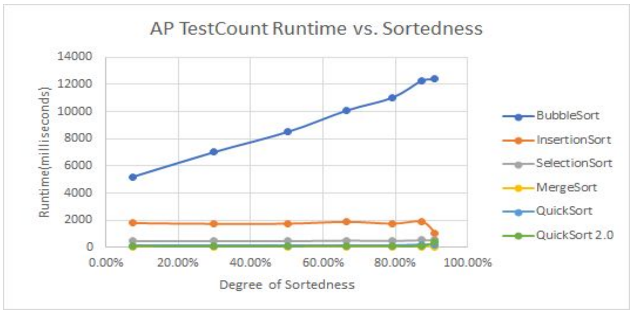


Figure 4a) Runtime vs. Data Size

Figure 4b) Memory vs. Data Size



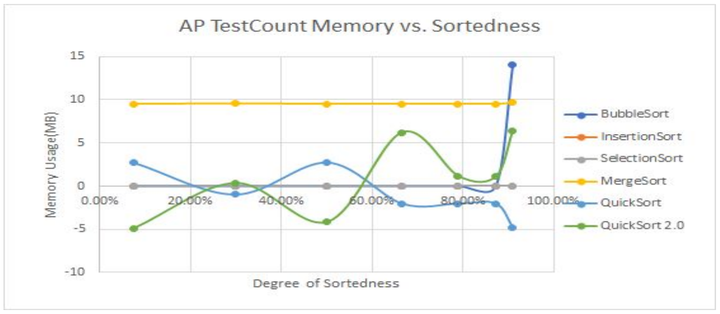


Figure 4c) Runtime vs. Degree of Sortedness

Figure 4d) Memory vs. Degree of Sortedness

Figure 4a demonstrates a similar relationship between runtime and the size of the data for each sorting algorithm compared to the other datasets. However, in Figure 2d, it is imperative to mention how bubble sort has the worst time complexity than both selection and insertion sort even at its lowest when the list is mostly sorted. This is likely due to the fact that bubble sort iterates over the entire list and performs multiple swaps whereas insertion sort and selection sort retains a sorted subset of the list and performs much less copies and comparisons. Merge sort still remains to have the lowest time complexity and in this sample in particular, the difference between mergesort and quick sort's time complexity was ten-fold when we approached a data size of over 64000. With a more dynamic dataset, merge sort performs significantly better, not only because of the superior time complexity and lack of overhead when compared to the median of medians approach, but with less duplicates means more recursive calls to make. This remains to be the case with merge sort and quicksort where quick sort performs significantly worse when the list is more disordered compared to merge sort where it remains more stable. This is because in the merge function, merge sort doesn't care how elements are arranged. The arrangement of the left and right subsets are always the same because each subset is sorted at the previous recursive call. This means that with a more disordered list, merge sort is the best in respect to time complexity. In respect to memory usage, Figure 4b illustrates that bubble sort has the highest memory usage in respect to the size of elements. Merge sort's actually remained lower than both quick sort variations in respect to data size, but much greater in respect to the degree of sortedness (Figure 4d). We can conclude that merge sort gives the greatest results in respect to runtime when operating on real data sets, but quicksort has superior space complexity when operating on real data sets. Which algorithm to select is contingent upon the constraints of the developer.