My name is Kian Rahimi and this is my third semester at SMU. I transferred here back in Spring 2020 and I love the change of environment that the school provides.

Project 3 of Data Structures tests my ability to identify sorts and their efficiencies in certain cases. The way that this project does this is by providing me 5 different sorts (I do not know which is what at first), and from there, use what is provided to me to determine which sort each one identifies to. I am trying to identify which provided sort is of the following five: Bubble Sort, Insertion Sort, Merge Sort, Quicksort, and Selection Sort.

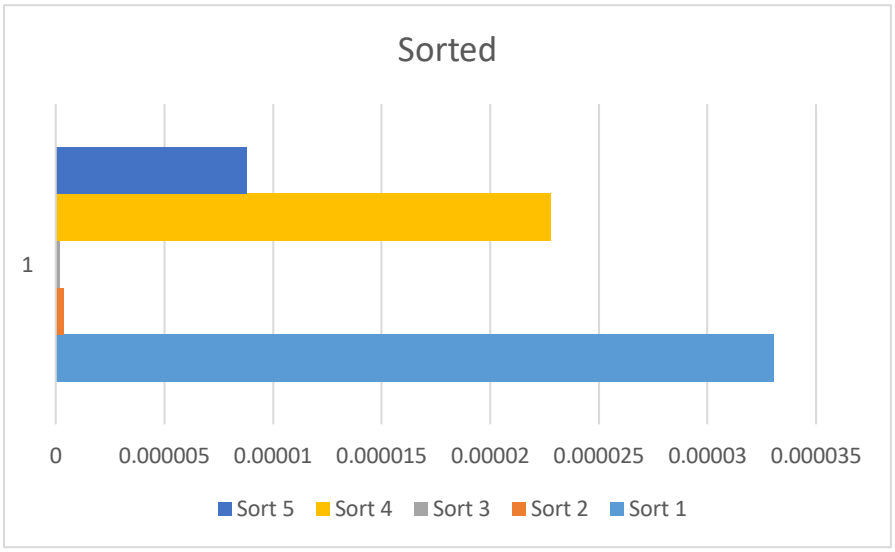
In order to identify each "mystery" sort, I must first use the "chrono" time library to determine the amount of time each one takes to perform what is asked. Of course to do, the same array must be ran across all sorts to determine the differences. With this in mind, I have created three different arrays for each one to run on. One that is already sorted, one in reverse, and one that holds randomized numbers. This way I can really see what is going on when provided different tasks rather than just one version of it. Using this process as well as help from outside resources, I can determine what each sort is through analysis based on time complexity and real-time numbers.

Below are three graphs that represent the performance of the 5 mystery sorts. Each array consisted of 300 elements, with sorted and reverse arrays being the same every run. The only thing dynamic were the order of elements in the random array. Every time I ran the program, a new random element array would be created. After doing so, the 5 sorts would run against the same array until next run time. This was done to show how each sort would perform in different situations.

Below are three graphs that show performance times of the 5 mystery sorts. In each graph, every sort was ran 5 times in which the average of the 5 were taken to create the graph.

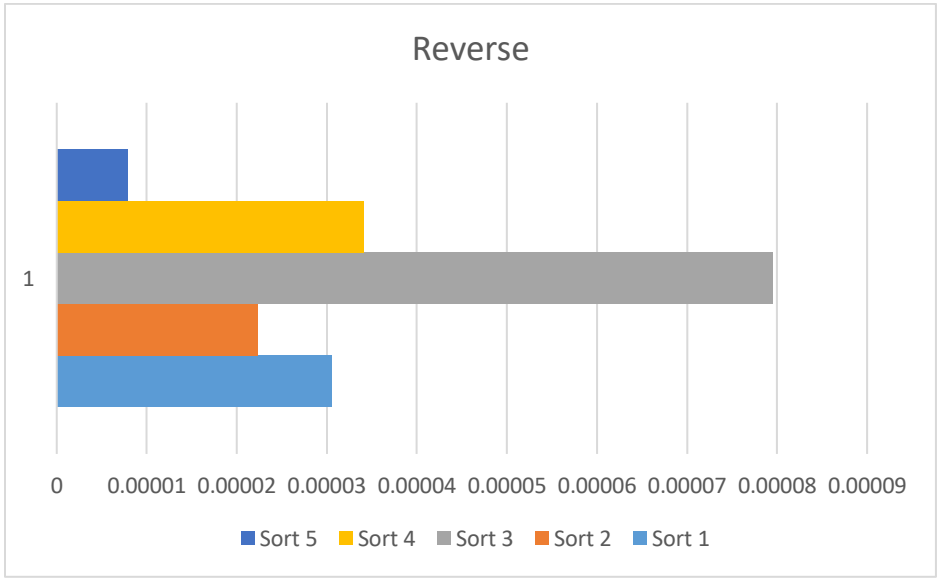
Sorted

Sorted	
Sort 1	0.00003306
Sort 2	0.00000034
Sort 3	0.0000002
Sort 4	0.0000228
Sort 5	0.0000088



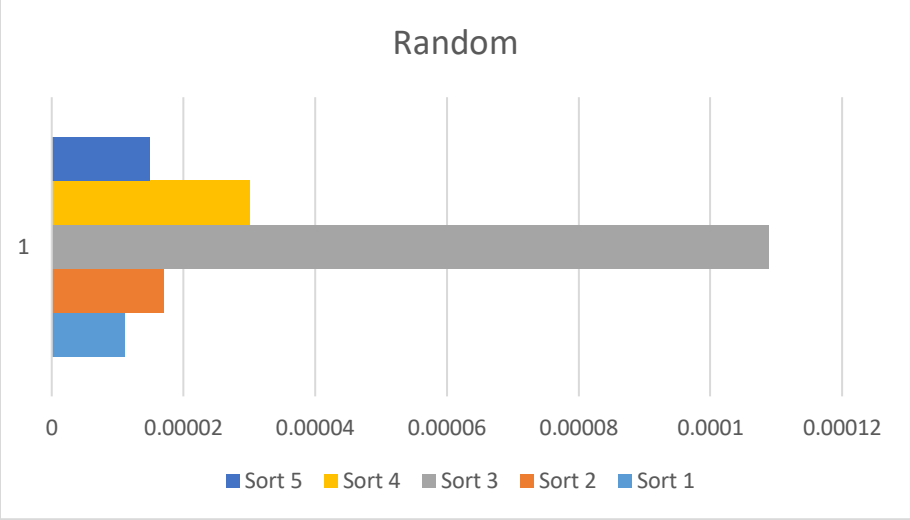
Reverse

Reverse	
Sort 1	0.00003058
Sort 2	0.00002226
Sort 3	0.00007954
Sort 4	0.00003404
Sort 5	0.00000784



Random

Reverse	
Sort 1	0.00001106
Sort 2	0.00001696
Sort 3	0.00010878
Sort 4	0.00003008
Sort 5	0.00001488



As one can see, right off the bat one can identify that sort 3 is a Bubble Sort. In its best case (when the array is sorted), the sort runs very quickly, however as soon as the sort takes a look at a reversed order or random, the sort runs much slower than the other 4. This reflects the Bubble Sort's big O time complexity, where in the best case it is O(n) meanwhile at its worst it is O(n^2).

Moving on, based on time complexity, the Merge Sort seems to be the most consistent sorting method that is fairly quick. Whether it be its best, worst, or average case, the merge sort

will always run at O(n log(n)). For this reason, and seeing that sort 5 is one of the more faster sorts in every single case, one can infer that sort 5 can be identified as Merge Sort.

Meanwhile, at the same time, Quicksort does generally run as fast as Merge Sort. At its best and average case, it also runs at a time complexity of O(n log(n)). However, when it comes to its worst case, Quicksort slows down to O(n^2). Interesting enough, this occurs when the array is already sorted, and depending on where the pivot is placed at the moment (in this case at the end of the array), the sort has a very tough time trying to organize the already sorted array. For this reason, we can identify that sort 1 is Quicksort.

Moving on to Selection Sort, the time complexity on this sort compared to the other 4 is the worst. On its best, average, and worst cases, the sort runs at O(n^2). This would consider this sort the slowest of all. With that being said, this can be described by Sort 4, since it is consistently one of the worst performing sorts.

Finally, with all sorts already identified, sort 2 can be considered as the Insertion Sort. This also properly represents the sort since based on its time complexity, the sort's best case is when the array is already sorted (O(n)) and one of the worst in every other case (O(n^2). This can be shown in the sorted graph, where the sort is extremely fast meanwhile it is generally of the slower ones in the others.