CS315 Programming Assignment 1- Sorting Writeup:

Running Code Instructions:

To run the program, change your terminal directory to data, which contains the files needed in the assignment.

My personal example:

```
PS C:\Users\rylee\CS315\data>
```

Then run the code under that directory. It will prompt to insert a file name, so type whichever file within the data folder that you want to sort. (Input must be of the form: *fileName.csv*)

Insertion Sort:

To implement insertion sort, we must iterate through the entire given array. To do this, I used a for loop iterating from the beginning to the end of the array (range(1, len(array))

We must then look at the value of the current index of the array, array[index] and compare it to the value before it, array[i]; i = index – 1. If array[i] is greater than array[index] swap their positions within the array. Else, leave them where they are.

My code then increases the index and compares array[index] to the values in the positions below it, hence the while loop “while i >= 0”, switching positions whenever array[i] is greater than array[index].

My code iterates through this process until the end of the array, which results in its sorted version.

Quick Sort:

In order to implement quick sort, I created another function—partition()—to use within QuickSort(). It assigns the pivot to the last value in the array, and val to the beginning of the array. It then iterates through the array, starting from the beginning, and compares the value at the current index, i, to the pivot value. If the value is less than or equal to the pivot value, place it before the pivot and increase val. If it is greater, place it after the pivot.

Moving on to the Quick Sort function itself, it first checks whether or not the array has one value, if it does, then we cannot partition the array, and the function terminates. Else, if starting index is less than the end index, partition the array and perform quicksort on both the left and right sides of the array.

Merge Sort:

While the length of the array is greater than 1, my code finds the middle of the array by dividing its length by two. It then splits the array into the bottom half, l, and the right half, r. It continues to do this recursively to each new half, until there is only 1 value in each.

After this, I assign iterators to go through the right half, the left half, and the main array. If the value at the index of the left array is less than that of the right, place the left value into the current index of the main array, and increase the left iterator by 1. Use the same logic for if the right value is less than the left – right value goes into current position at the current index of the original array, and right iterator increases. Continue until the main array is full and sorted.

Comparison Counts:

File:	**Isort**	**QSort**	**Msort**
pokemonRandomSmall.csv	330	192127836042210632904721845161010163927375531673535906126	1158
pokemonRandomMedium.csv	862	154872456570567531448499565400843397806108378431851911275982443838474779465837269301019242406877703585870361392428927030936643738743689 8	3648
pokemonRandomLarge.csv	1598	319663278309373637928533278876818954543560285020160174027764520244405592373529717982323155423342138387357911327867070860327819973538151538079603216477529526243792635330385955129808280410955181982701637080551176521681858122767014059006138048630125 0	7456
pokemonReverseSortedSmall.csv	330	92127836042210632904721845161010163927375531673535906126	1158
pokemonReverseSortedMedium.csv	862	154872456570567531448499565400843397806108378431851911275982443838474779465837269301019242406877703585870361392428927030936643738743689 8	3648
pokemonReverseSortedLarge.csv	1598	319663278309373637928533278876818954543560285020160174027764520244405592373529717982323155423342138387357911327867070860327819973538151538079603216477529526243792635330385955129808280410955181982701637080551176521681858122767014059006138048630125 0	7456
pokemonSortedSmall.csv	330	92127836042210632904721845161010163927375531673535906126	1158
pokemonSortedMedium.csv	862	154872456570567531448499565400843397806108378431851911275982443838474779465837269301019242406877703585870361392428927030936643738743689 8	3648
pokemonSortedLarge.csv	1598	319663278309373637928533278876818954543560285020160174027764520244405592373529717982323155423342138387357911327867070860327819973538151538079603216477529526243792635330385955129808280410955181982701637080551176521681858122767014059006138048630125 0	7456

(I have a feeling there was an issue with my comparison counts because of the duplicate numbers for each individual size, i.e. all ..small.csv = 330 for insertion sort

Time Complexities:

Insertion sort – O(n^2) for average and worst case, O(n) for the best case.

Merge Sort—O(nlogn) for all cases.

Quick Sort – O(nlogn) for the best and average cases and O(n^2) for the worst case.

Analysis:

- Based on the time complexity of **Insertion Sort**, the number of comparisons should be around n^2 or n, respectively. Based on my own model above, it seems as if all files were the average case, falling between the estimated count of n and n^2 for the function itself. (personal results are skewed because I could not get the iterator to work correctly despite working on it for hours)
  The highest value of comparisons should have been from each reverse-sorted file. The average or middle case would be caused by the random files, and the best cases would be the sorted files.

- Based on the time complexity of **Merge Sort**, all orders of elements in an array are best, average, and worst case, ignoring size for now. Since all of my values for each sort of the same size were identical, one could assume that that the estimated time complexity is accurate given the input files, but when they're exactly the same, it hints at something being wrong with the comparison iterator—which there probably is. (Numbers may appear larger because I iterated through until every element was sorted, resulting in multiple loops and instances of recursion within the function.

- Based on the time complexity of **Quick Sort** and my implementation of it, inputting the files would always result in the worst case complexity, since I chose my pivot as an extreme point. This would explain the really large numbers corresponding to the function in the above table. I also recursively called Quick Sort within the function itself until the array was completely sorted, which would also make the values higher than expected. I doubt that the numbers for each of the same-sized array would actually be identical, since they are all sorted differently.