David Giacobbi
CPSC 223: HW#3 Write-Up

A. Sorting Time Cost Tables

n = 10000

	MergeSort()	Std::Sort()	InsertionSort()
Ascending	0.002049 secs	0.0000114 secs	0.0000874 secs
Descending	0.0020048 secs	0.0000174 secs	0.350464 secs
Random	0.0029344 secs	0.0003588 secs	0.175663 secs

n = 20000

	MergeSort()	Std::Sort()	InsertionSort()
Ascending	0.0043344 secs	0.0000198 secs	0.0001746 secs
Descending	0.0042926 secs	0.000034 secs	1.40844 secs
Random	0.0061982 secs	0.0007554 secs	0.704573 secs

n = 40000

	MergeSort()	Std::Sort()	InsertionSort()
Ascending	0.0094992 secs	0.0000408 secs	0.0003546 secs
Descending	0.00906 secs	0.0000688 secs	5.62367 secs
Random	0.013239 secs	0.0016462 secs	2.80234 secs

B. The merge sorting algorithm did far better than the insertion sort, especially as the n size increased in number. However, it is important to note that the insertion sort still outperformed the merge sort when the list was already in sorted order. Even though the list is sorted, merge sort still must go through the process of breaking up the vector into subsections and sorting each part, giving it a slower performance for ascending. It still appears that standard sort is the best performing of the sort. Again, given that it is provided by the C++ language, the algorithm under the hood must have a lot of slight modifications and check stoppers that allow it to run so efficiently. Overall, when given an unknown list, merge sort is a solid option to choose because it is not only quite simple to implement, but it is also one of the most well-rounded performing algorithms. Whether it was ascending or descending or random, merge sort seemed to perform all three in roughly the same amount of time.

C. This sorting algorithm was only a couple of lines of code. It was also easy to implement because the functions that broke down some of the algorithm's main calls made it easier to understand. The hardest part, however, was trying to understand how to piece together the vector after it has been recursively divided into singular elements. Even though the temporary array provides you with a workspace to sort the elements in the two current subsections, it was hard to visualize how exactly the elements should be moved around in the vector, without needing to create another temporary vector to work with. I found it surprising that a recursive sort like this still only required a couple of loops to perform the whole entire algorithm.