Will Horn

Project 3

- **For each sorting algorithm, explain the difference in runtimes for the different type of data. That is, why do you think the randomized, ascending, descending data yields different/similar runtimes for the merge sort algorithm; is there an inherent reason with the way the code works? Answer this question for each algorithm; i.e. for heap sort, merge sort, quick sort and insertion sort. Explain please.**
    - **Heap Sort:** For the heap sort algorithm, no matter what type of data you have, the algorithm will still have to perform deletemin() or deleteMax() n times and rearrange to maintain the heap properties. This means that theoretically, for any input, the run time should be of order nlogn. This does not appear to be the case though. When using a very large set of data, the algorithm improved in speed from the random integers to the presorted increasing integers, and it sped up even more when run on the presorted decreasing variables. This could simply be some form of error in my code, or it could just be slight variances from one data set to another. I believe that the latter possibility is what was occurring in this situation.
    - **Merge Sort:** Merge sort slightly better run times from one type of data to the next, that is, the data it sorted the fastest was the descending array, then the ascending array, and the slowest was the randomized integers. Although they were slightly different, they were all still quite similar which makes sense considering that no matter what type of data you input into the algorithm, it will always run at around nlogn time, because it has to split up all of the different arrays and sort them accordingly. Even if the data is already sorted, it will still separate everything into sub-arrays and be forced to run in nlogn time. This explains why all of the runtimes were so similar.
    - **Quick Sort:** Quick sort is somewhat similar to the heap sort and merge sort algorithms, being that as long as a good pivot is chosen, the algorithm will run in about nlogn time. It will see improvements in speeds overall compared to the other two algorithms previously mentioned, but the main thing I have noticed is that the quicksort algorithms runtime for an ascending array is significantly lower than its runtime for the random array and the descending array, which had very similar runtimes. I believe that this is caused by the fact that the algorithm did not have to call any std::move() operations for this array, which would inherently lower the runtime, since it had to perform less operations on the data.
    - **Insertion Sort:** This algorithm was the most fun to work with, because the runtimes for different types of data were all extremely different. First of all, all of the runtimes differed. The random integers took a decent amount of time, 37 seconds for 100,000 values, the sorted array was always finished in well under a second regardless of the size of the array, and the decreasing array took almost twice as long as the random array. I believe that this is due to the fact that the decreasing array is the absolute worst case for this algorithm, since by the end, it

would have to make the n moves to get the last digit to the first place, whereas the random array was an average case where every move would also be a random number similar to all of the data in the array. The ascending array always took a very small amount of time, because the algorithm iterates through the array, sees that ist is already sorted, then it returns the same array, because it doesn't need to make any changes. During class it was mentioned that for an array with a size of one million would take a very long time to sort using insertion sort. This was very true, but I wanted to know just how long it would take. I originally ran the 10,000 and 100,000 tests to get that data, then knowing that for a random array and a descending array, it would be running in n^2 time, I crunched the numbers to find out just how long it would theoretically take to fully sort the final, massive arrays. After doing my math based on the fact that the random array took 37 seconds for the array of size 100,000, I found that, theoretically, it should take approximately one hour, one minute, and twenty seconds to sort an array of size one million. Just to see if I had found this correctly, I let the program run for a few hours and came back to find that I was only about forty seconds off, because it took 3582.76 seconds to run, which is about fifty-nine minutes and forty-two seconds. But that was not the end of it for me, since I had also calculated how long it should take to sort the descending array of size one million. My math showed me that the algorithm should take approximately two hours, two minutes, and ten seconds to fully sort the descending array of size one million. Once again, I was fairly close to being exactly correct, since the program took 7006.29 seconds to sort the descending array. This would be equivalent to one hundred sixteen minutes and forty-six seconds, or one hour, fifty-six minutes, and forty-six seconds, so again I was only off by a handful of seconds. This showed me that the runtimes can be generally followed, but the theoretical runtime will be a bit off from the real runtime. Due to knowing the theoretical runtime of this algorithm, I was able to gauge where the largest runtime will fall with fairly close accuracy.

- **Explain the difference in runtimes between the insertion sort algorithm and quick sort. Why do you think there are differences/similarities?**
    - The insertion sort and quicksort algorithms have one main thing in common, and that is their speed for sorting a presorted array. Both of them had the increasing array as their fastest sort. The reason they are so similar is that for both algorithms, there are operations that are not performed when sorting an ascending pre-sorted array. For quicksort that operation is the std::swap operation, whereas the insertion sort is skipping over the std::move operation. The runtimes are still different for these two algorithms, and that is because the quicksort algorithm is still breaking up the array into other arrays, calling the median3 functions, and attempting to sort the array, while insertion sort looks it over once and sees that it is sorted, therefore running in linear time.
    - As for the differences between the two algorithms, I think it is fairly apparent that there is a large difference in the runtime from quicksort to insertion sort for both

the random array and the descending array. For quicksort, It sorted the descending array faster that the random array, due to it being able to obtain perfect pivot points every time wue to the nature of a descending array. Insertion sort took almost double the time to sort the descending array compared to the random array, because the descending array is its absolute worst case for efficiency. The number of moves to move the last item in the array to the first position in the array would be equivalent to the size of the array - 1. This is what causes these major differences.

- **Explain the differences/similarities in runtimes between the heap sort, merge sort and quick sort algorithms. Why do you think there are differences/similarities?**
    - Heap sort is most similar to merge sort, since all of their runtimes were within .13 seconds of each other. They both also sorted the descending array the fastest and the random array the slowest. This is due to them both running in nlogn time for any type of data. The biggest difference comes when you compare those two algorithms to the quicksort algorithm. Overall, quicksort is the fastest, which you could probably have guessed by its name. This is because of how it sorts the data. It divides everything around pivot points to sort all the data, which breaks up the data faster than dividing everything into individual components and sorting from there, and it is faster than performing n deletemin() operations and having to re-heapify the array every time one of those operations is used. Due to this, the fastest quicksort was the increasing array, and the slowest quicksort was the random array. This is very different compared to the other two, whos fastest sorts were the descending array and their fastest being the random array.

number of integers N	runtime									theoretical Big-Oh runtime		
	randomized integers			presorted in increasing order			presorted in decreasing order					
	10,000	100,000	1,000,000	10,000	100,000	1,000,000	10,000	100,000	1,000,000	random order	increasing order	decrease order
heap sort	0.004	0.055	0.796	0.005	0.052	0.693	0.004	0.05	0.67	nlogn	nlogn	nlogn
merge sort	0.005	0.055	0.71	0.004	0.046	0.566	0.003	0.044	0.56	nlogn	nlogn	nlogn
quick sort (no cutoff)	0.002	0.024	0.287	0.001	0.001	0.134	0.002	0.019	0.237	nlogn	logn	nlogn
insertion sort	0.356	37.637	3582.76	0	0.002	.021	0.719	73.431	7006.29	n^2	n	n^2