COP 3530 Sorting Analysis
Leeson Chen (UFID: 79097-2212)

Introduction:
In this project I wrote three sorting algorithms (selection sort, insertion sort, and heap sort) and compared their time complexity for files of different sizes and arrangements. Selection and insertion sort are quadratic, and generally take longer than heap sort under most circumstances. The files contained 5000, 10000, 20000, and 50000 integers in arrangements of either increasing order, decreasing order, and random order. The sorting algorithms put the integers into increasing order when finished, meaning that some files had a best case scenario, and some had a worst case scenario.

Code:
First I created the total of twelve files with a small C++ program (not needed, thus not included, in this report). The code containing my algorithms was a larger C++ program. The program first takes in the file, reading line by line and turning all the strings into integers then placing them in an array. After the whole file is in a single array, the system clock is called to get the beginning time in milliseconds. One of the three algorithms is used on the array, then the system clock is called again. Subtracting these times shows how many milliseconds the algorithm took to sort the entire file. I originally had code to print out both the unsorted and sorted arrays to make sure my algorithms were running properly, but this was commented out during actual testing. An example of this preliminary code is shown below in Figure 1.

Figure 1: A screenshot of part of my code, and the debugging output when testing selection sort on a small array of 10 integers. Output arrays were removed during actual data collection.

Testing:
For the data collection, I made sure that my laptop was fully charged and connected to the charger to minimize variation in samples. For each individual sequence (e.g., Selection / 5000 / Random) I ran five trials and took an average. The raw data is included in Figure 2.

Figure 2: Sorting Analysis Raw Data
Five trials were done for each sequence, and the average is shown in the rightmost column. Averages are the values used to plot the graphs.

Sorting Analysis Raw Data

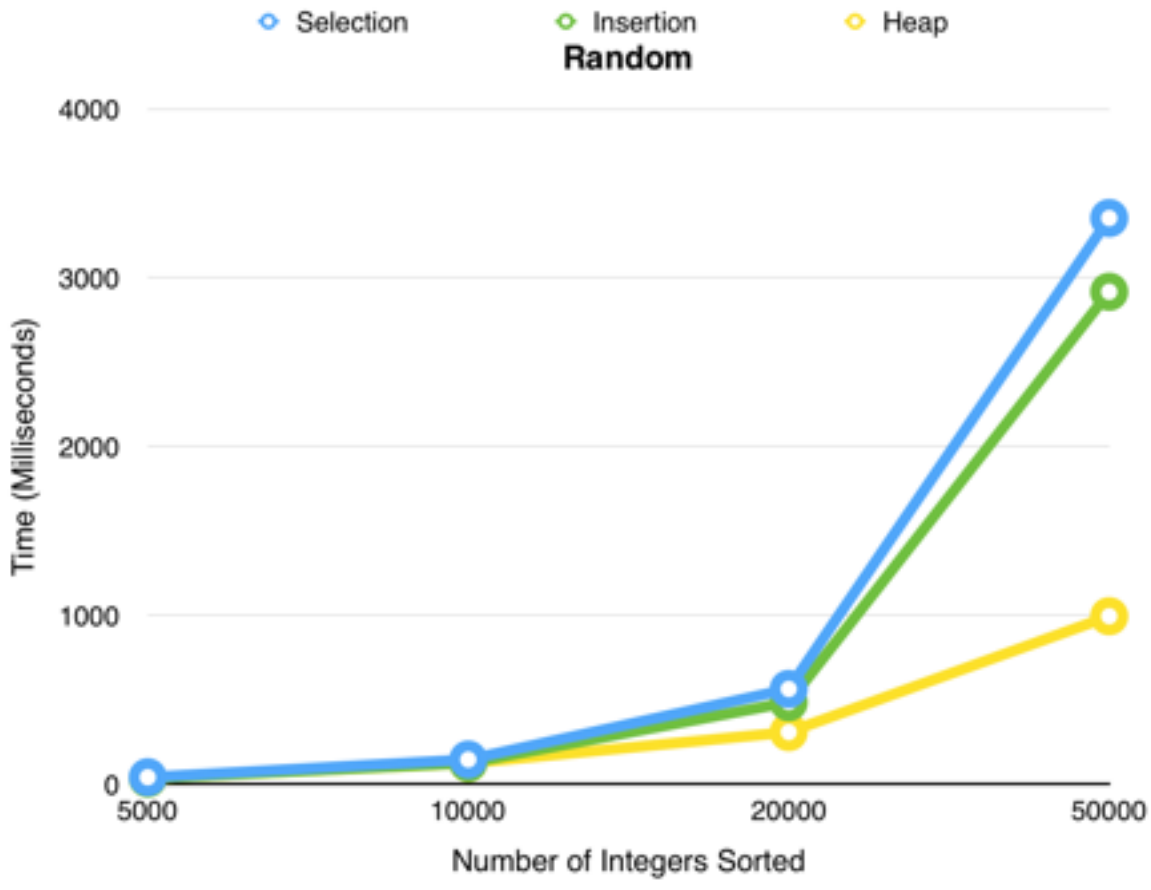
			Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Avg
Selection	**5000**	**Random**	38	42	37	43	37	39.4
		Decreasing	39	35	37	40	37	37.6
		Increasing	35	36	34	37	35	35.4
	10000	**Random**	155	143	148	137	138	144.2
		Decreasing	149	152	150	153	146	150
		Increasing	142	144	140	144	145	143
	20000	**Random**	567	561	565	555	565	562.6
		Decreasing	613	616	582	568	584	592.6
		Increasing	574	567	546	554	546	557.4
	50000	**Random**	3361	3448	3262	3383	3313	3353.4
		Decreasing	3474	3419	3517	3694	3703	3561.4
		Increasing	3362	3335	3321	3359	3545	3384.4
Insertion	**5000**	**Random**	36	33	31	31	33	32.8
		Decreasing	60	65	58	60	59	60.4
		Increasing	4	3	4	3	4	3.6
	10000	**Random**	126	127	123	116	128	124
		Decreasing	251	236	244	248	248	245.4
		Increasing	8	6	7	7	8	7.2
	20000	**Random**	489	482	493	498	459	484.2
		Decreasing	923	923	957	913	933	929.8
		Increasing	17	14	14	15	16	15.2
	50000	**Random**	2959	2917	2938	2858	2912	2916.8
		Decreasing	5406	5244	5505	5237	5751	5428.6
		Increasing	42	42	41	40	39	40.8
Heap	**5000**	**Random**	31	34	33	34	30	32.4
		Decreasing	32	31	31	33	31	31.6
		Increasing	32	31	32	33	31	31.8
	10000	**Random**	121	132	120	120	121	122.8
		Decreasing	124	122	127	120	122	123
		Increasing	126	122	127	126	128	125.8
	20000	**Random**	311	308	298	313	305	307
		Decreasing	308	323	310	334	304	315.8
		Increasing	284	307	318	316	302	305.4
	50000	**Random**	1003	997	984	923	1062	993.8
		Decreasing	962	988	974	1004	1023	990.2
		Increasing	994	996	1022	1003	983	999.6

Algorithms:
Here, it is helpful to reference a chart of how these three algorithms generally perform under different circumstances. As shown below in Figure 3, selection sort runs at quadratic time $O(n^2)$ regardless of whether it is given a pre-sorted array or unsorted array. Insertion sort also runs quadratically on average, but in the best case scenario (it is given an array already sorted in the desired arrangement) it can run linearly, at O(n) time. Heap sort was chosen as the non-quadratic algorithm and runs at linearithmic time O(n log(n)), again regardless of whether the input array is already sorted or not. This behavior can be visualized in the following graphs, which demonstrate each algorithm's behavior as file size increases, categorized into the best, worst, and average cases.

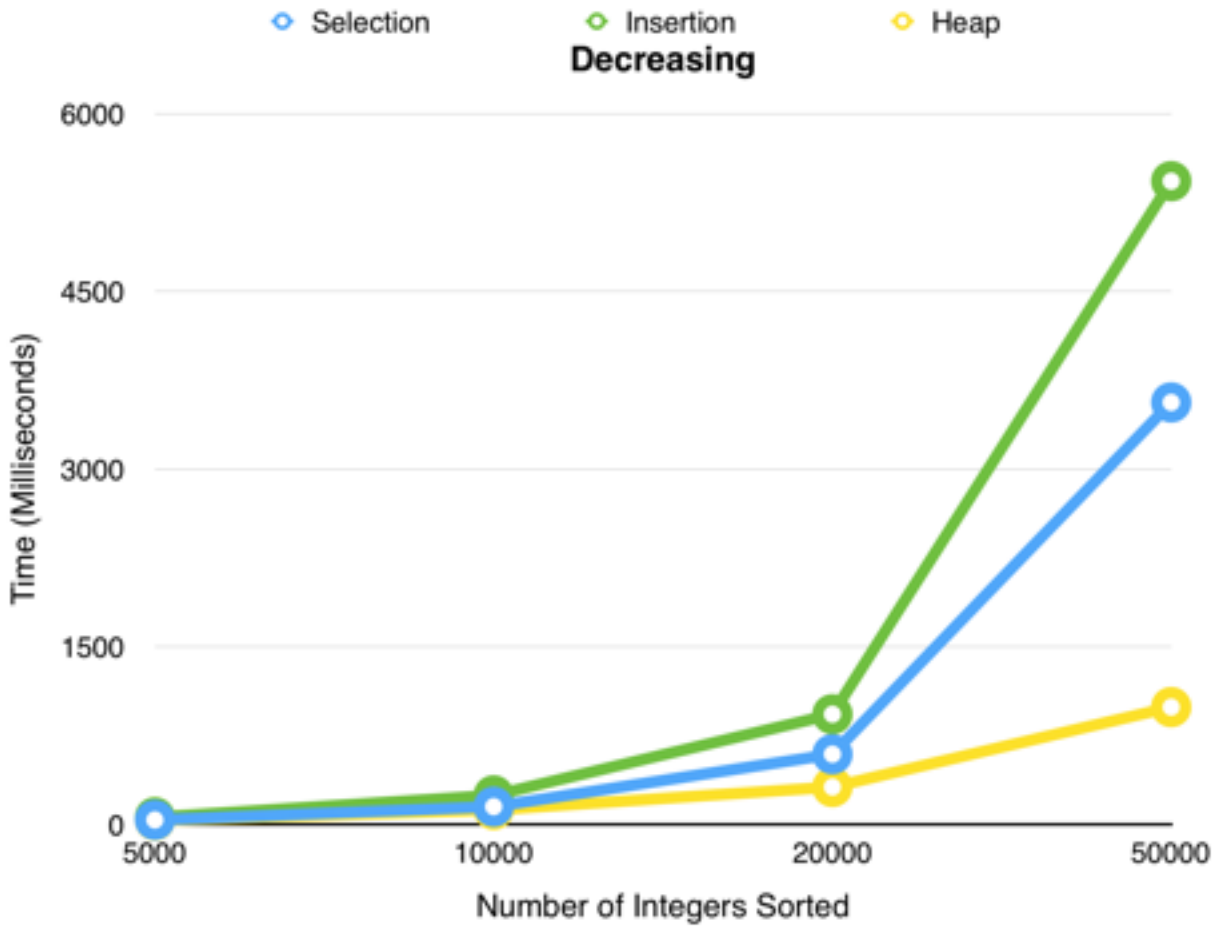
Figure 3: Chart displaying the different performances for best, worst, and average cases of the three algorithms used

Algorithm	Best Case	Worst Case	Average Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))

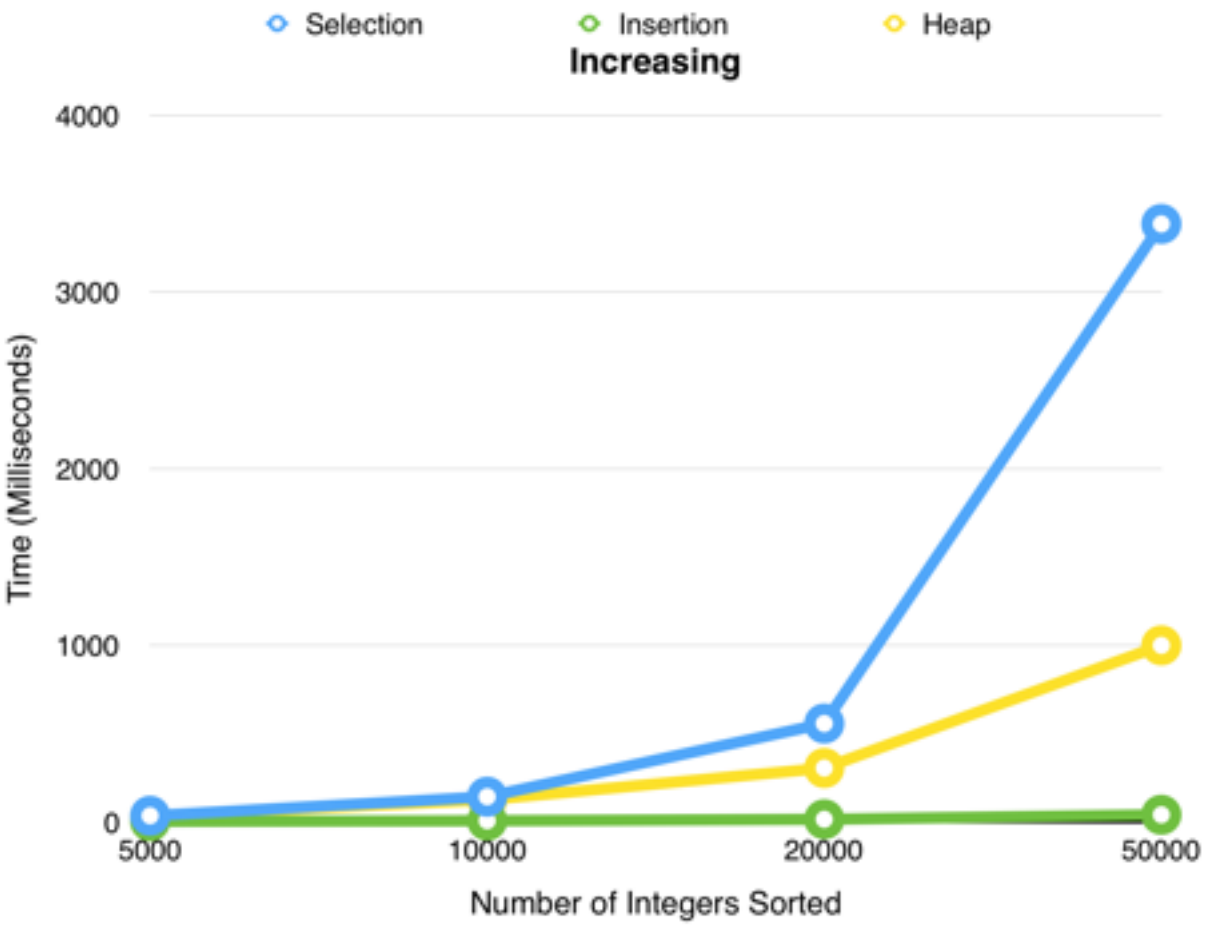
Random Order (Average Case):
Giving the algorithm a randomly-ordered file results in an average-case time complexity. Here, we see that selection and insertion sort run at roughly the same time, which makes sense as both should run at $O(n^2)$ on average. This quadratic behavior is noticeable as file size grows from 5000 sorted integers to 50000 sorted integers, and the time complexity increases rapidly each time, with the jump from 20000 to 50000 being roughly 6x larger. Heap sort runs much faster, demonstrated by the smoother increase at O(n log(n)) rate.



Decreasing Order (Worst Case):
This performance is theoretically similar to the average case, because worst case performance for all three algorithms is the same as their average case performance. However, these data showed that while selection and heap sort operated at about the same time as average, insertion sort took double the amount of time for all four file sizes compared to its average case performance. The average time for insertion sort, respective to each file size, came to 32.8, 124, 484.2, and 2916.8 milliseconds. In comparison, its worst case performance respective to the same file sizes were 60.4, 245.4, 929.8, and 5428.6 milliseconds. Each increment performed at roughly x1.9 slower than the average case for insertion sort.
The reason for this significant delay is quite clear, when one observes a visualization of insertion sort. In a worst case scenario such as this one, each element being sorted is guaranteed to be compared to every element preceding it. Whereas for an average case, each element should only be compared to about half of the preceding elements. Therefore, a worst case scenario for insertion sort guarantees a maximum number of comparisons, which is roughly double the number of comparisons in an average case—thus doubling the amount of time taken to sort the array.



Increasing Order (Best Case):
Inputting an array where the elements are already sorted into increasing order, where the algorithm desires to sort the elements into increasing order, results in the best case scenario. Once again, selection sort and heap sort cannot fully appreciate the burden already done for them, because both algorithms perform at $O(n^2)$ and $O(n \log(n))$ respectively, regardless of how their input data is already arranged. Thus their performance remains nearly identical to average and worst case scenarios. However, insertion sort demonstrates an extremely noticeable performance boost, operating nearly instantly for all file sizes, even 50000 integers! This is once again explained by a visualization of insertion sort, which would show that each element is compared only once to its immediate predecessor. The algorithm recognizes that the current element is correctly sorted, and moves onwards. This means that only one comparison happens for each element, and therefore the algorithm runs linearly at O(n) time, corroborating the best case time complexity for insertion sort in Figure 3.



Conclusion:
After synthesizing the raw data into graphs, it becomes very clear that insertion sort will change significantly between $O(n^2)$ and $O(n)$ depending on whether it receives a sorted or unsorted array. Its worst case performance is also double its average case performance, due to the number of comparisons happening for each element. Selection sort and heap sort fail to recognize whether the provided array already meets sorting requirements or not—this is both a strength and a weakness, because it guarantees $O(n^2)$ and $O(n \log(n))$ time complexity respectively, and does not fluctuate wildly the way insertion sort does. However, while selection sort and heap sort will never sharply increase their time complexity, they will never have a lightning-fast best case scenario. These “sturdier” sorting algorithms may be a better choice for some circumstances, such as when input data may possibly be organized in the least effective manner. On the other hand, when there is a chance for input data to already be optimized, an algorithm with best and worst cases such as insertion sort may be preferable.