My table of timings, along with my 3 graphs composed of 2 plots each for the 2 different sorting algorithms are at the bottom in the appendix.

First, I will discuss the timing difference on the array of increasing values passed to both sorting algorithms. As we can see, selection sort is dramatically slower than insertion sort. This is because the array is in increasing order. We must examine how both algorithms handle this array to fully understand this timing difference. Insertion sort takes the array, sets the variable representing the current value to the value at the index where the outer loop is at, and sets the value of j equivalent to the index where the outer loop is at too. It then checks if the value in the index before it is greater than the current value and if the value of j is greater than 0. The array we are given is in increasing value, so this is never true as the values are strictly increasing in value and each value is followed by a value that is greater. Thus, selection sort continues to loop and rapidly go through the rest of the array without ever entering the additional while loop needed to swap the smaller and bigger value. Unlike insertion sort, selection sort takes longer due to the 2 loops it must enter. It sets the start and smallest value to the first index in the array and sets the inner loop incrementer i to the value of the outer loop k is at. This is to use the while loop to loop through an array inside that of the entire array. As the outer loop that k takes part in gets smaller because k gets larger, so does i, reducing the subarrays length and the distance the inner loop must traverse. Continuing the example, selection sort enters the while loop while i is less than the length of the array. It then loops from index i to the end of the array checking if every value is larger than the value of the variable smallest (the first value in the subarray). This is always true in the array of increasing values. Then it exits the while loop and goes to the outer loop, incrementing it by 1 and then re-entering the inner loop. Simply put, selection sort forces the array to be examined through 2 loops every time, making it much more time-intensive when sorting through an array of increasing values.

Next, we examine the timings on the array of decreasing values. This time, the algorithms are similar in their timings. From the graph, we can see that both timings follow what looks like an exponential trend. This is because they both act similarly given that they both have while loops nested inside of their for loops (2 sets of loops) that they must traverse and thus behave identically when passed arrays of decreasing values. Let us see how both algorithms handle the decreasing arrays. Insertion sort takes the array, sets the value of current to the second value in the array, and sets the inner loop incrementer j to the current value of the outer loop variable k. Then insertion sort checks whether the while condition is fulfilled. This is fulfilled as j is always bigger than 0 on the first check and the value before the current value is bigger than the current value every time because the array is filled with decreasing values. Therefore, the insertion sort algorithm enters the while loop every time and loops through pushing the current value to the beginning of the array before returning to increment the outer loop once more by 1. Selection sort acts in a similar fashion. It starts at the first index of the array and sets the variables start and smallest equal to the value where the incrementing value k is (starts at arr[0]). Then, it always enters the inner while loop. This creates a slowly decreasing subarray on every loop of the outer loop. Inside the while loop, it first checks if the first value in the array is smaller than than the current value. If it is, it continues the loop. Otherwise, it sets the value of the variable “smallest” to the value of the newfound smallest value and sets the variable of

index_of_smallest equivalent to the value of the index in order to remember where this smallest value is located. At the end, after it has looped through the entire sub-array, it swaps the value of the current position in the array with the newfound smallest value. Then it executes the outer loop and enters the while loop to examine the subarray once again. Because the array is an array of decreasing values, on every increment through the subarray, it must save that value as the newfound smallest value, taking a lot of time as well. It should be noted that insertion sort is slower by a good amount (2-4 seconds) as the value of the array increases. This is due to the fact that it needs to swap the larger and smaller values every single time before it decrements j and repeats the process. Selection sort is a bit faster in that it remembers the values and only completes the swap once it has finished looping through the subarray.

Finally, we will examine how both algorithms treat the array of random values. Both algorithms handle the array of random values similar to how they handled the array of decreasing values. Insertion sort behaves identically to how it did in the beginning of receiving an array of decreasing values: it enters the outer loop and sets the value of current to the current value in the array, beginning with the second value in the array. Then it diverges from the behavior it exhibited when receiving an array of decreasing values. It again checks whether j (which is set equal to the index that the outer loop is at) is greater than zero, which it passes, and also checks if the value before current is greater than the value of the current. This is where the differences occur. Because it is an array of random variables, this happens some of the time. Unlike the array of increasing numbers, we can't be certain it never happens and unlike the array of decreasing numbers, we can't be certain it happens every time. Therefore, sometimes it enters the while loop, swaps the values, decrements j, and then re-checks the requirements of the while loop. Looking at the graph, we can see that it does in fact enter the while loop a good amount and have to work to swap the values and continue decrementing j as the graph looks similar to that of the decreasing values graph for insertion sort. It should be noted that for values of 2500, 5000, 7500, and 10000, insertion sort is about 2 seconds faster every time than it was for decreasing values. This is probably because sometimes it didn't have to decrement all the way back to the beginning of the array nor did it have to even enter the while loop, like it did for the array of decreasing values every single time. For selection sort, we also see similar behavior to how it handled receiving an array of decreasing values. Again, it enters the outer loop and sets the values of start and smallest to that of the value at the index the outer loop is positioned at currently. It also sets i, which determines where the subarray begins that the while loop goes through, equal to where the outer loop index k is at. It then always enters the while loop. This causes the slight difference in time between selection and insertion sort for the array of random numbers, as selection sort enters this loop every single time. It checks if smallest defined as the starting value of the subarray is smaller than the next value in the subarray and continues if that is true. If not, it sets the value of the variable "smallest" to the value of the newfound smallest value and sets the variable of index_of_smallest equivalent to the value of the index in order to remember where this smallest value is located. At the end after it has looped through the entire sub-array, it swaps the value of the current position in the array with the newfound smallest value. Then it executes the outer loop and enters the while loop to examine the subarray once again. Looking at the timings, it looks slightly faster than it was for the array of decreasing variables - probably because it has to

loop through the subarray every time like before, but has to do less swapping of values because the array is not composed of strictly decreasing numbers.

Appendix (diagrams below)

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1000 Increasing Insertion	0.000153	0.000171	0.000181	0.000153	0.000153	0.0001622
1000 Increasing Selection	0.073698	0.078835	0.080857	0.079629	0.081483	0.0789004
2500 Increasing Insertion	0.000478	0.000463	0.000406	0.000395	0.000599	0.0004682
2500 Increasing Selection	0.478652	0.495594	0.500738	0.485585	0.549945	0.5021028
5000 Increasing Insertion	0.000725	0.000969	0.000728	0.000767	0.000774	0.0007926
5000 Increasing Selection	1.974762	2.079288	1.984843	1.865053	2.081405	1.9970702
7500 Increasing Insertion	0.001203	0.001187	0.001147	0.001192	0.001129	0.0011716
7500 Increasing Selection	4.452044	4.448252	4.271917	4.601417	4.533897	4.4615054
10000 Increasing Insertion	0.001657	0.002773	0.001449	0.001556	0.001514	0.0017898
10000 Increasing Selection	7.550403	8.190998	7.512235	9.812035	7.982539	8.209642
1000 Decreasing Insertion	0.108120	0.108230	0.107200	0.127225	0.113840	0.112923
1000 Decreasing Selection	0.081456	0.082219	0.083023	0.112760	0.089299	0.0897514
2500 Decreasing Insertion	0.715431	0.769657	0.742264	1.040052	0.743254	0.8021316
2500 Decreasing Selection	0.502080	0.580174	0.525567	0.818788	0.519963	0.5893144
5000 Decreasing Insertion	2.909669	3.534662	2.977209	3.517080	3.040351	3.1957942
5000 Decreasing Selection	2.021978	2.434544	2.174880	2.089222	2.405521	2.225229
7500 Decreasing Insertion	6.879156	7.149698	6.838027	7.224261	7.313200	7.0808684
7500 Decreasing Selection	4.785973	5.119100	4.812385	4.893272	4.940304	4.9102068
10000 Decreasing Insertion	12.045336	12.920044	13.255708	13.170844	12.700579	12.8185022
10000 Decreasing Selection	8.032677	8.513995	9.362508	8.859321	9.325039	8.818708
1000 Random Insertion	0.054810	0.055350	0.065697	0.064407	0.062944	0.0606416
1000 Random Selection	0.075164	0.076617	0.107269	0.085512	0.084797	0.0858718
2500 Random Insertion	0.368344	0.386029	0.410444	0.438053	0.396102	0.3997944
2500 Random Selection	0.471081	0.496689	0.538577	0.543731	0.507863	0.5115882
5000 Random Insertion	1.485192	1.660756	1.627153	1.976956	1.613281	1.6726676
5000 Random Selection	1.960026	2.003306	2.418940	2.041567	2.097774	2.1043226
7500 Random Insertion	3.444653	3.619255	3.690654	3.617898	3.764216	3.6273352
7500 Random Selection	4.429923	6.160252	5.175535	4.866293	4.518095	5.0300196
10000 Random Insertion	6.004257	7.145830	6.939633	6.929857	6.411300	6.6861754
10000 Random Selection	7.711443	8.171374	8.343826	8.291154	8.447128	8.192985