Ignat Miagkov

CSCI 241

February 8 2020

Project 1: A Battle of Sorts

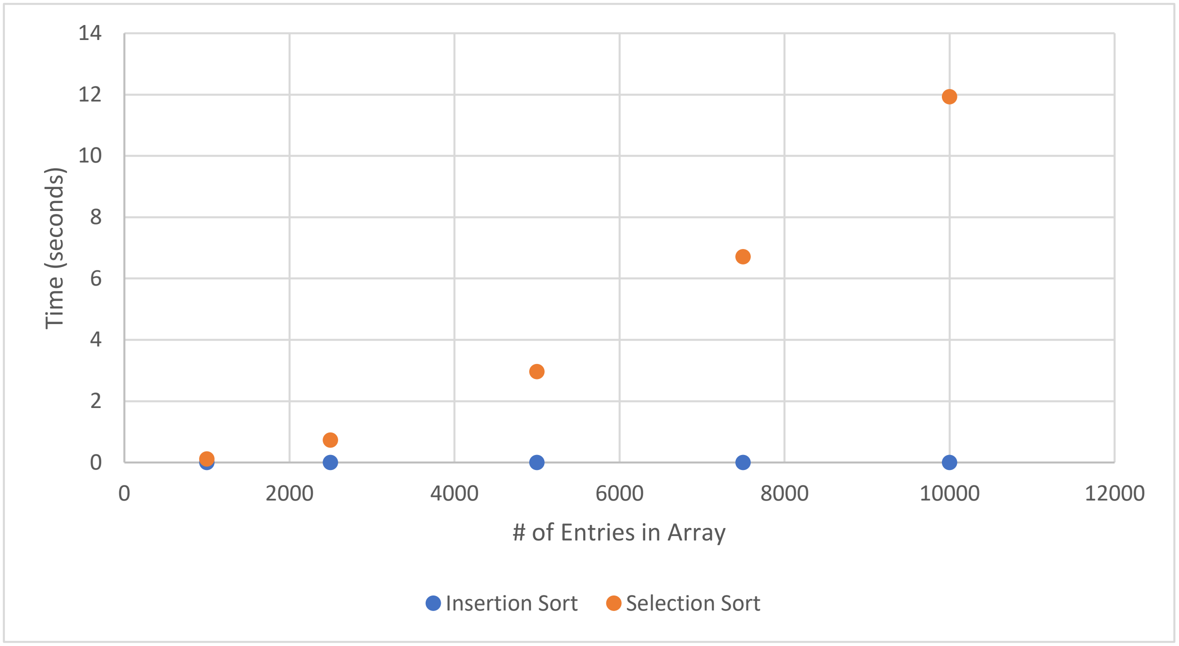

Figure 1. Plot of performance of insertion sort and selection sort on an array of increasing values.

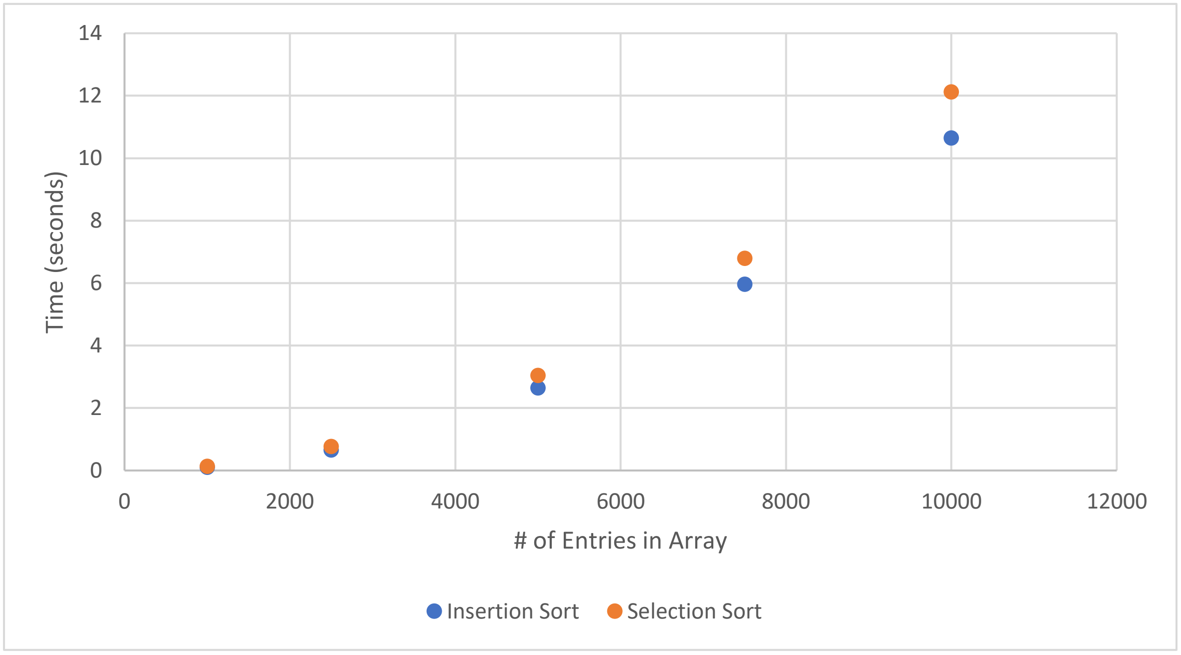

Figure 2. Plot of performance of insertion sort and selection sort on an array of decreasing values.

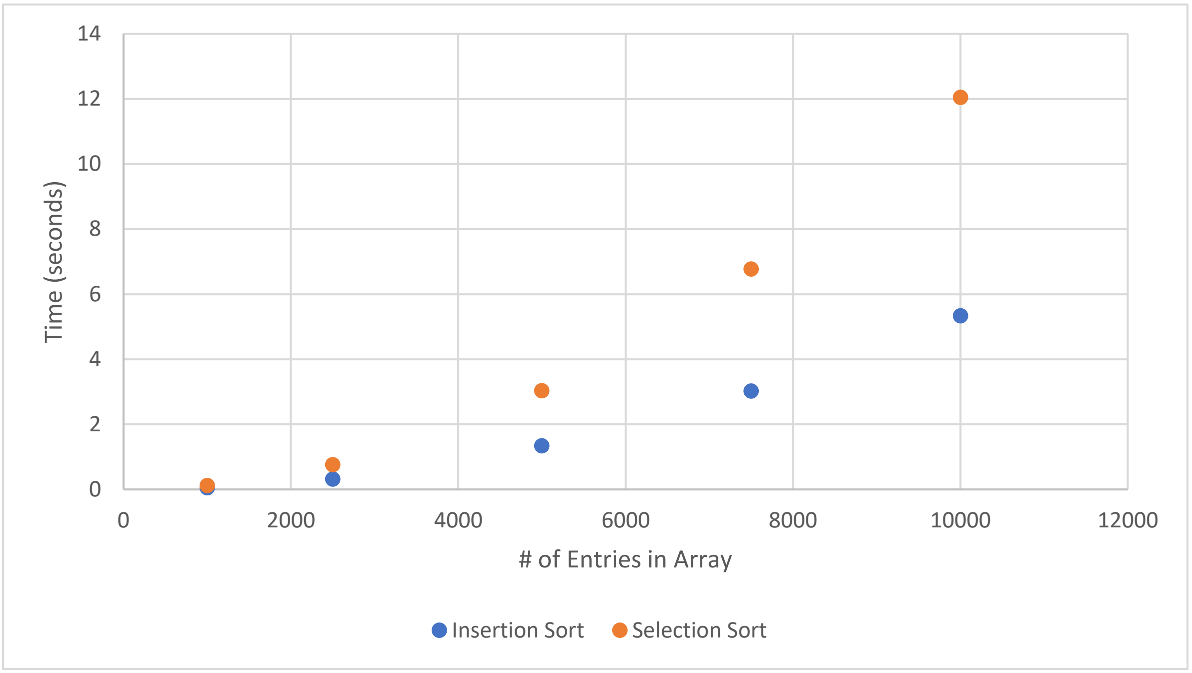

Figure 3. Plot of performance of insertion sort and selection sort on an array of randomly generated values.

<u>Explanation</u>

Insertion sort traverses the sorted values of an array, plus one additional value (which is the next one in the array), and then places the said next value in its appropriate location amongst the already sorted values. Let's say the sorted section consists on [3,4,6] and the next value in the array is 1. Insertion sort will then place the 1 first in the sorted section, making the new sorted section as [1,3,4,6]. Selection sort instead traverses all the values of the values of the array, finding the smallest value, and then placing it first in the array, making it the sorted section. Next, it will traverse all the values again, except the smallest value from the previous generation, and find the smallest value again, placing it after the smallest value found previously. For example, let's say we have an array with values [4,1,3,6]. Selection sort will traverse the entire array and find the minimal value, 1, and swap it with 4. This cycle is repeated for the unsorted section of decreasing length, until all values are sorted. Thus, selection sort will make the same amount of comparisons for arrays of equal size, no matter how well it is sorted initially. On the other hand, insertion sort will only start sorting the next value in line only if it is not in order already, thus reducing the amount comparisons that are needed by a significant amount in all scenarios, except its worst case.

In figure 1, it is seen that insertion sort takes almost zero time to sort the already sorted array. That is because since all the values are in order already, insertion sort has to make zero changes to the array, as only one comparison is made with each loop, thus reducing the run time to almost zero (Windows machines display this time as zero due to operating system limitations). Selection sort on the other hand, still must traverse all the values, finding the minimal value with each unsorted portion of the array, and putting that value in the sorted section. Since selection sort traverses the entire array every time, despite the different ordering, the runtimes for selection sort will all be very similar. However, the already sorted array in the best-case scenario for insertion sort, thus it takes minimal time to complete.

In figure 2, both insertion sort and selection sort take similar time to complete sorting the array of decreasing order. An array of decreasing order is the worst-case scenario for insertion sort; thus, it must traverse the entire array and with each comparison and place the next unsorted value in the correct spot in the already sorted section. Selection sort still must traverse the entire array, as it does per usual. It will still find the minimal value and swap it to the current position of that it has marked at, the position at which the sorted section of the array grows. Both sorts have similar performance in an array of decreasing order because both sorts must traverse the entire array, and both make an equal amount of comparisons.

In figure 3, it is seen that selection sort still takes similar time to complete the sort of an array with random values, while insertion takes about half the time that of its performance in sorting the array of decreasing order. This is because insertion sort, on average, must make half as many comparisons as when it must sort an array of decreasing values. Selection sort still makes the same amount of comparisons: it will always traverse the entire array, thus taking a similar amount of time, no matter what the initial ordering is. However, for insertion sort, due to the nature of randomness, it will take, on average, about the time halfway in between increasing order (where it makes n comparisons) and decreasing order (where it makes n^2 of comparisons).

Selection sort will always run in O(n^2), no matter the circumstances, since this sort will always traverse all the values and make the same amount of comparisons every time. The sorted section will always grow at the same rate, one value at a time, thus explain the consistent run time. However, insertion sort will not always make the same amount of comparisons, since once the next value has been sorted, it does not have to compare it to all the values that are already sorted before it in the array, making its efficiency anywhere between O(n) and O(n^2). This optimization of insertion sort makes it more efficient than selection sort in almost all cases, except for when they have similar run times for arrays of decreasing order.

Appendix 1. Raw Data

	1000 Inc Ins	1000 Inc Sel	2500 Inc Ins	2500 Inc Sel	5000 Inc Ins	500 Inc Sel	7500 Inc Ins	7500 Inc Sel	10000 Inc Ins	10000 Inc Sel
1	0	0.125	0	0.75	0	2.969	0	6.672	0	11.875
2	0	0.125	0	0.734	0	2.984	0	6.766	0	12.063
3	0	0.109	0	0.718	0	2.937	0	6.609	0	11.813
4	0	0.125	0	0.734	0	2.953	0	6.766	0	11.813
5	0	0.125	0	0.75	0	3.016	0	6.781	0	12.094
	1000 Dec Ins	1000 Dec Sel	2500 Dec Ins	2500 Dec Sel	5000 Dec Ins	5000 Dec Sel	7500 Dec Ins	7500 Des Sel	10000 Dec Ins	10000 Dec Sel
1	0.094	0.125	0.656	0.766	2.703	3.016	6.094	6.844	10.875	12.203
2	0.109	0.125	0.641	0.75	2.625	3.109	5.891	6.859	10.578	12.266
3	0.109	0.125	0.641	0.766	2.609	2.984	5.922	6.672	10.563	11.906
4	0.094	0.125	0.656	0.75	2.625	3.016	5.969	6.719	10.578	11.953
5	0.109	0.125	0.656	0.781	2.641	3.063	5.938	6.859	10.625	12.281
	1000 Ran Ins	1000 Ran Sel	2500 Ran Ins	2500 Ran Sel	5000 Ran Ins	5000 Ran Sel	7500 Ran Ins	7500 Ran Sel	10000 Ran Ins	10000 Ran Sel
1	0.063	0.109	0.328	0.766	1.359	3.078	3.078	6.813	5.422	12.078
2	0.063	0.125	0.328	0.75	1.375	3.031	3.016	6.856	5.281	12.25
3	0.047	0.125	0.313	0.766	1.328	2.984	3.031	6.656	5.359	11.859
4	0.047	0.125	0.313	0.75	1.328	2.984	3.016	6.688	5.25	11.938
5	0.047	0.125	0.328	0.75	1.313	3.078	2.969	6.859	5.375	12.141

Appendix 2. Averages

Increasing Order	Insertion	Selection
1000	0	0.1218
2500	0	0.7372
5000	0	2.9718
7500	0	6.7188
10000	0	11.9316
Decreasing Order	Insertion	Selection
1000	0.103	0.125

2500	0.65	0.7626
5000	2.6406	3.0376
7500	5.9628	6.7906
10000	10.6438	12.1218
Random	Insertion	Selection
1000	0.0534	0.1218
2500	0.322	0.7564
5000	1.3406	3.031
7500	3.022	6.7744
10000	5.3374	12.0532