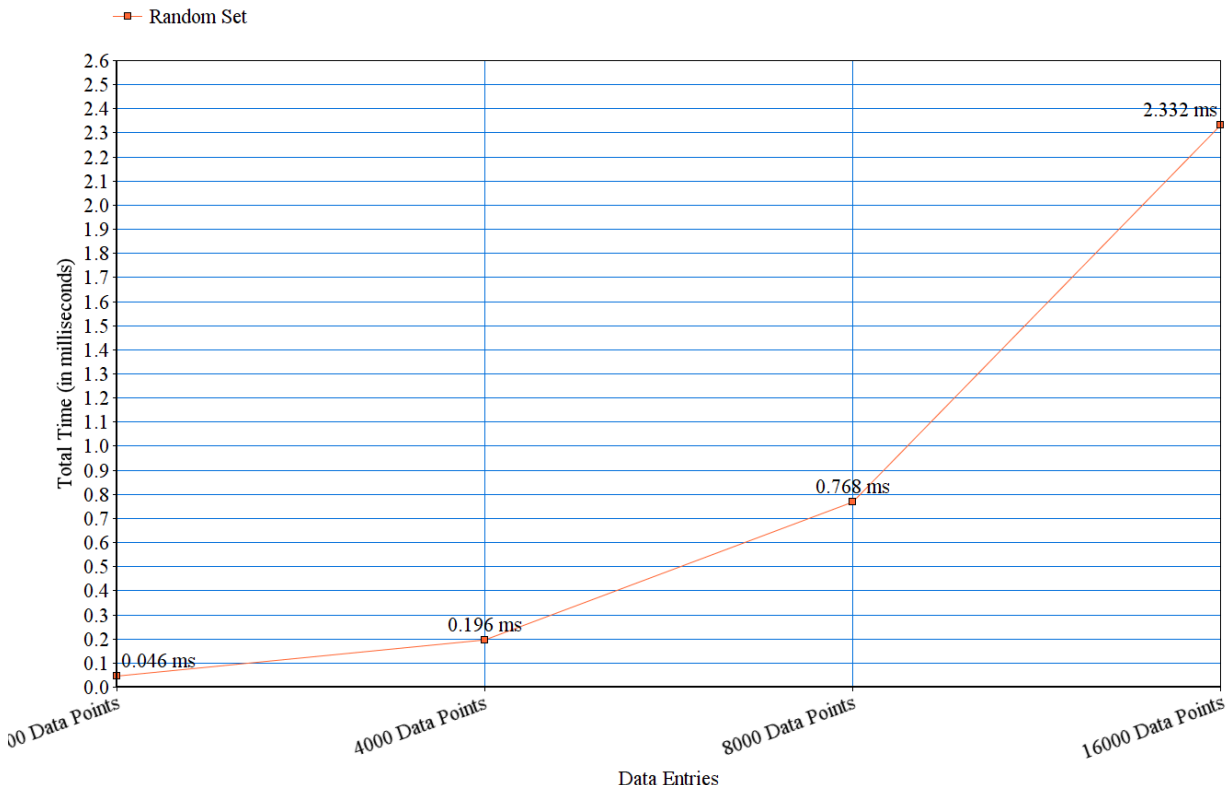
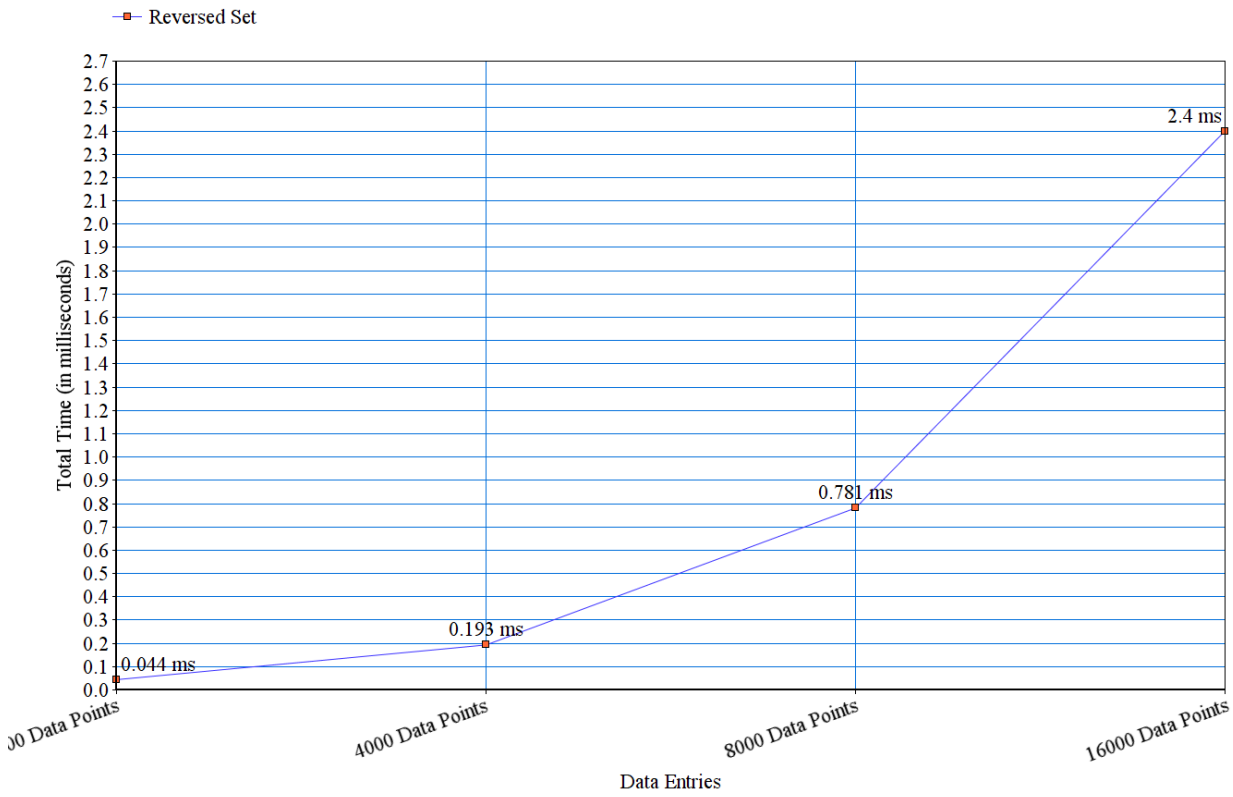
Parankush Bhardwaj

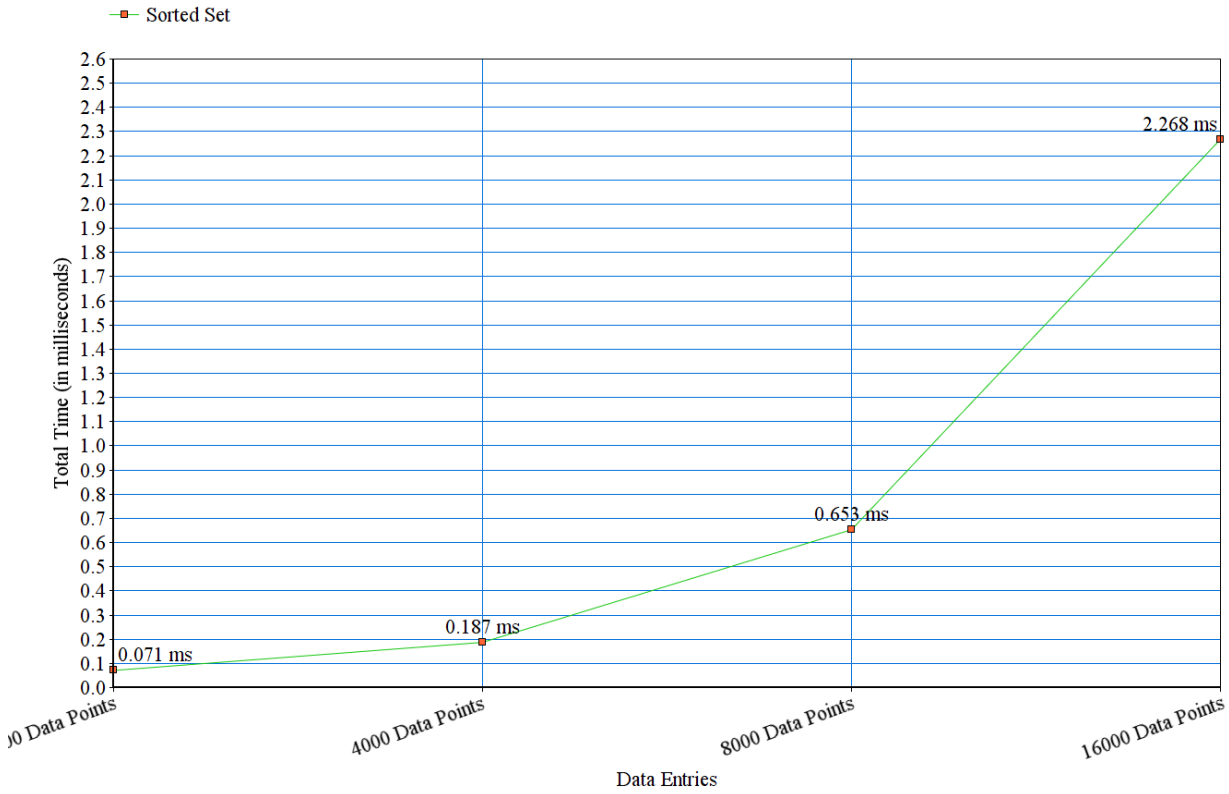
Workshop Week 4

2A. Graph the running times on each data file, for N=2000, 4000, 8000, 16000.

[Graphs are also viewable as separate .png files on GitHub]







3. From your timing data analyze the rate of growth of SelectionSort using the doubling rule.

Every time the data is doubled, the time to complete the sort also increases at a rate of approximately 2^(n).

This is because when we went from 2000 data entries to 4000 data entries, the time doubled from about 0.1 milliseconds to 0.2 milliseconds.

Moreover, when we went from 4000 data entries to 8000 data entries, the time quadrupled from 0.2 milliseconds to approximately 0.8 milliseconds.

And when we went from 8000 data entries to 16000 data entries, the time had a growth of 8x, from 0.8 milliseconds to about 2.4 milliseconds.

We also know that 2^1 = 2; 2^2 = 4; 2^3 = 8; and that 2^4 = 16.

So, we can infer from this data that our time doubled when n, which is a integer on the x-axis, was 1 (meaning it represents the 2000 data points). When n was two (meaning it represented the 4000 data points), its output, again in terms of growth rate, was 4. When n was three (8000 data points), we found its growth rate to be 8. This follows the 2^n relationship.

To conclude, this is not a linear relationship. Doubling the output does not guarantee having twice the time for completing the sort.

4. Analyze the SelectionSort code: use number of array accesses as your cost model, and describe the rate of growth in tilde (~) notation.

If we wanted a approximation of its run time, we can count the steps from the loops (since they are the most costly parts of the sorting algorithm). Indeed, we see that there are two for-loops in the program. Thus, if we were to assume that the list of things to sort has length N, that means the program goes through each index in N exactly N times. In other words, every index in the array is compared to every other index in the array. It takes length N to go through every index, but for each index we are also making comparisons that take N steps as well. This results in a time complexity of about ~(N^2).

We can more accurately analyze the code by creating a example of something it needs to sort. If we were to give it a random unsorted array of: [5, 3, 7, 1, 2, 4], we can then trace the program to see its costs. I've done so here below:

At the first pass, 5 and 1 are swapped [according to the rules of selection sort]. The cost of this includes 5 comparisons (the integer was compared to 3, 7, 1, 2, and 4), 3 steps where we found a possible best value for swapping (3, 1, and 2), and 3 steps for swapping. The total is 5 + 3 + 3, which is 11 steps.

List after the first pass: [1, 3, 7, 5, 2, 4]

At the second pass, 3 and 2 are swapped. This pass included 4 comparisons, 2 assignment statements for declaring the possible best value, and 3 steps for the swap. Total steps of this pass is 9.

List after the second pass: [1, 2, 7, 5, 3, 4]

At the third pass, 3 and 7 are swapped. This pass would have 3 comparisons, 3 assignment statements, and 3 steps for the swap. The total for this pass is 9.
At the fourth pass, 5 and 4 are swapped. There were 2 comparisons, 2 assignment statements, and 3 steps for the swap, resulting in a pass with a total of 7 steps.

List after the third pass: [1, 2, 3, 5, 7, 4]

At the fourth pass, 5 and 4 are swapped. There were 2 comparisons, 2 assignment values, and 3 steps for the swap, creating a pass with 7 steps.

List after the fourth pass: [1, 2, 3, 4, 7, 5]

Finally, in the final pass, 7 and 5 are swapped. The is just 1 comparison, 2 assignment values, and 3 steps for swapping, creating a pass with 6 steps

List after the last pass: [1, 2, 3, 4, 5, 7]

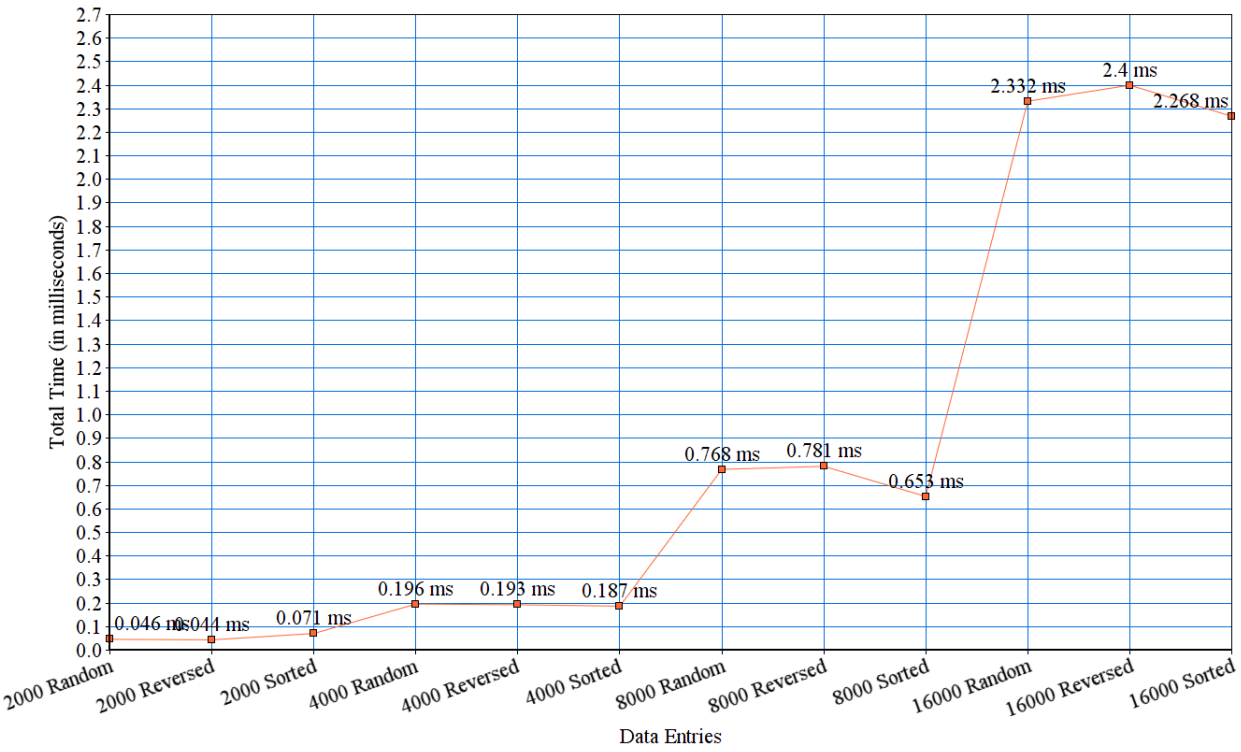
If we were to add up all the steps of all 5 passes, we get exactly 42 steps. Because there were 6 elements in this list, according to our earlier claim of this algorithm having a average time complexity of about ~(N^2), the total average time complexity of this list would be 36. 42 is pretty close to this number (the additional costs are for the comparisons and swaps). Again, most of the costs are going to the loops in the algorithm.

5. How does the order of the input data affect SelectionSort? Explain why, by referring to the algorithm or the code. (How many times `less` and `exch` called?)

The order of the input doesn't alter the average N^2 time complexity of the program. In the previous question, we used the code to sort 6 random integers and found it to have a time complexity that was near N^2. If we use the code again, but this time sort 6 already sorted integers, then the total number of steps ends up being 35. In addition, if we give it a reversed ordered list of 6 integers, the total number of steps is 52. Both cases are very close to the 36 steps we expect from a average N^2 time complexity. Thus, we can then infer through our examples that because sorted, reversed, and random lists all give us the same average time complexity, then the order of the input data doesn't affect the algorithm's average time complexity.

The number of times "less" and "exch" are called do not vary by the order of the input. Those only change based off of the length of the data that must be sorted. This is because if we trace the program using a sorted list of length N and trace a different, but random, list of length N, we find that the number of comparisons and array accesses remain constant.

I created the graph below based on my data to better show the minimum differences between the time complexities of data with the same length but different input. We can see that the time only varies dramatically when the length of the input changes.



Parankush Bhardwaj

Data collected (if interested)

[there are three entires for each set, the values on the graphs are the averages of those three entries]

Random 2000: 0.052, 0.044, 0.043

Reversed 2000: 0.047, 0.043, 0.042

Sorted 2000: 0.059, 0.093, 0.062

Random 4000: 0.181, 0.176, 0.233

Reversed 4000: 0.216, 0.185, 0.178

Sorted 4000: 0.188, 0.183, 0.191

Random 8000: 0.73, 0.808, 0.765

Reversed 8000: 0.746, 0.81, 0.785

Sorted 8000: 0.69, 0.638, 0.63

Random 16000: 2.341, 2.185, 2.472

Reversed 16000: 2.386, 2.273, 2.541

Sorted 16000: 2.052, 1.675, 3.077