**Adv. Comp Prog: Sorting and Searching Algorithms**

**Selection Sort**

| Attribute | Number of Objects | Sort Time (ms) |
| --- | --- | --- |
| Dog Weight (Double) | 10 | .6678 |
| Dog Weight (Double) | 100 | 2.7864 |
| Dog Weight (Double) | 1000 | 59.9294 |
| Dog Weight (Double) | 10000 | 146.6335 |
| Dog Weight (Double) | 100000 | 11267.274799 |

**Quick Sort**

| Attribute | Number of Objects | Sort Time (ms) |
| --- | --- | --- |
| Dog Weight (Double) | 10 | .1007 |
| Dog Weight (Double) | 100 | .4223 |
| Dog Weight (Double) | 1000 | 3.0657 |
| Dog Weight (Double) | 10000 | 5.226499 |
| Dog Weight (Double) | 100000 | 107.7128 |

***Analysis***

Selection Sort predicted timing from O(n^2) is at least under the average “worst-case” complexity. For example, in the case of the number of objects increasing from 10 to 100, the time taken predicted was 66.78 milliseconds while the time taken was *much* less than. But then again, results from the lower numbers may not accurately represent O(n^2). It should also be noted that in the case of the jump from 10,000 objects to 100,000, the predicted time would be around 14600, while the actual sorting time was around 11300. This shows the algorithm approaching the asymptote upper bound as we calculate a larger size of values.

We can also see something similar in Quick Sort with its O notation, O(, which is the average “best-case” complexity. In the case of the number of objects increasing from 10,000 to 100,000, the estimated time would be 332 milliseconds. Although this is a little far from 107, it should be noted that notation represents an average and computers can take a range of milliseconds.

As we compare both tables, we see that Selection Sort is significantly slower than Quick Sort, mostly because partitioning is a much faster process than swapping; it also checks out Selection Sort’s O notation compared to Quick Sort’s O notation, because for any large n, is greater than .

Thus, yes, our algorithm analysis conclusions from class apply when we are sorting objects.

**Sequential/Linear Search (Shuffled)**

| Attribute | Number of Objects | Sort Time (ms) |
| --- | --- | --- |
| Dog Name (String) | 10 | .0432 |
| Dog Name (String) | 100 | .1258 |
| Dog Name (String) | 1000 | .6403 |
| Dog Name (String) | 10000 | 2.1814 |
| Dog Name (String) | 100000 | 9.629 |

**Sequential/Linear Search (Sorted)**

| Attribute | Number of Objects | Sort Time (ms) |
| --- | --- | --- |
| Dog Name (String) | 10 | .0021 |
| Dog Name (String) | 100 | .0068 |
| Dog Name (String) | 1000 | .2037 |
| Dog Name (String) | 10000 | .2937 |
| Dog Name (String) | 100000 | .9113 |

**Binary Search (Sorted)**

| Attribute | Number of Objects | Sort Time (ms) |
| --- | --- | --- |
| Dog Name (String) | 10 | .009601 |
| Dog Name (String) | 100 | .0139 |
| Dog Name (String) | 1000 | .0104 |
| Dog Name (String) | 10000 | .004799 |
| Dog Name (String) | 100000 | .0521 |

***Analysis***

With regard to the linear search being performed on a shuffled array of object attributes, we see that searching is done in a short time when the list size is less than or equal to 1000. However, once we start increasing from there, we see that search time significantly increases. Because of the nature of the linear search, where it goes by the list sequentially until it finds the target, a large list size hinders its efficiency, which matches the conclusion we reached in class.

I also did a comparison between a binary search and a linear search for sorted data. As seen, binary search is significantly faster (and shows no substantial increase in time taken as the number of objects greatly increases), while linear search, once again, takes a longer amount of time searching through an increased number of objects, which checks out the observations we made in class about linear search best being for a small amount of data while the binary search is extremely powerful in sorted data. Although this isn’t necessarily about sorting, our class conclusions about search algorithms also apply when dealing with objects.