Cecilia Albert-Black
DSE511
Homework 4

Part A: Sorting Showdown

Since I'm new to coding with Python, I needed some help coding for the insertion sort function. I modified the code from Geeks for Geeks on "Python program for insertion sort" (link provided in the code) to write the function. Then, I decided to compare Python's built-in sort() function and the insertion sort function to quicksort, since it's known to have an extremely fast run time. Again, I didn't know how to code for this, so I used the Geeks for Geeks article on "Python program for quicksort" (link also provided in the code) to write this function as well. I chose sample sizes of 10^3, 10^4, 10^5, 10^6, as was suggested in the homework assignment. Runtime was lowest for quicksort across all sample sizes, followed by Python's built-in function sort(), and the longest run time was for insertion sort. The linear plot of sample size vs. runtime shows how insertion sort has a Big-O of O(n^2), which means it takes a long time to run nearly all sample sizes. On the other hand, sort() appeared to have a Big-O of about O(n), meaning it linearly increased as sample size increased. Even better was quicksort, which seemed to have a Big-O of O(1), at least for the sample sizes I used. Therefore, my observations align with the theory of Big-O.

Part B: Data Structures in Action

For this section I compared membership query performance (run time) of different data structures (list, set, dictionary) of varying sample sizes (n). I found that membership query performance was nearly zero across all sample sizes for the sets and dictionaries, which is indicative of O(1) complexity. In other words, the complexity of the operations is constantly very low across all sample sizes. On the other hand, the membership query performance of the list appeared to increase as the sample size increased exponentially, indicating O(n^2) behavior which is not favorable.

Part C: Reflection

In Part A, the experiments demonstrated the known Big-O behavior of sorting algorithms and data structures I learned in class. Insertion sort gave hints of O(n^n) behavior, even though the log-log graph I produced in Part A did not capture very clearly. I did not calculate insertion sort's performance above 20,000 observations because it would have taken too long; thus the plots could not fully capture its bad performance at high sample sizes. The exponential behavior of insertion sort was more evident in the linear graph, where it nearly represented a vertical line. Although true exponential behavior was not fully captured in either of these graphs, it was still evident that insertion sort was not a favorable sorting algorithm compared to python's sort() and NumPy's quicksort. Even at low sample sizes, insertion sort is not a viable sorting algorithm, since it performs nearly 4 orders of magnitude worse than quicksort and 3 orders of magnitude worse than sorted() function, as shown in the linear plot.

Python's sorted() function performed better than insertion sort across all sample sizes and performed only 1 order of magnitude longer/worse than quicksort at very low sample sizes. However, the difference in efficiency of sorted() compared to quicksort grew over time; in other words, sorted() increasingly became slower with increasing sample size whereas quicksort remained relatively low across all sample sizes. This was a clear indication that sorted() exhibited O(n) behavior while quicksort exhibited O(1) behavior, which is their expected Big-O behavior.

In Part B, I did not expect that generating random queries for sets and dictionaries would run faster than the ordered list because it was not intuitive that unordered data structures would be faster to run than ordered data structures. As a human, I tend to think order is always better than disorder; however, not in this case. It may also be that generating random queries from an ordered data structure like a list generates more complexity whereas creating a random query from an already randomly ordered data structure does not change the complexity and may even keep it simple (like a well-shuffled deck of cards). This simple code helped me realize that it's not always better to have a sorted data structure before performing certain functions, because it may not improve the function's efficiency.

Overall, I learned about sorting algorithms for the first time and that quicksort is the best sorting method regardless of sample size/complexity and is easy to use if NumPy is available. I also learned about different data structures and their efficiency in run times for a given function. This information will be very applicable as I slowly become a data scientist.