Assignment 2: Advanced Programming

Lingqing Song, Nikolina Milioni, Oreen Yousuf

1 Searching

In the first part, we implemented two variants of sequential search: iterator and indexing. Theoretically, time complexity for the two searching methods are both $O(n)$. In practice, average time for the two methods in 10 tests are 82.7 and 81.3 seconds. For this data structure, binary search was also implemented. Its theoretical time complexity is $O(\log n)$, and it is indeed a lot faster in practice, with an average time of 0.19 seconds in 10 tests.

Then we implemented the same three search methods on double-ended queue, a different data structure, and did 10 tests each. For both linear methods, time complexity is still $O(n)$. The actual time consumed by the first variant, iterator using enumerate, is similar with that on list, with an average of 82.6 seconds. However, the second variant, indexing, takes a lot more time than expected: 413 seconds on average. Reason for this might be that time complexity for searching by index in deque is higher. As for binary search in deque, theoretical time complexity is $O(\log n)$, and in practice the average time is 0.64 seconds over 10 tests. We find this confusing, since indexation for deque should be faster.

Finally, we conducted a simple search method on python dictionary. Since python dictionary is implemented using hash tables, the average time complexity for searching an item in a dictionary is $O(1)$. Worst case $O(n)$ happens when all valuees share a same hash bucket, and is not likely to happen. Average time for searching all words in `toSearchFor.txt` is 0.013 seconds.

2 Sorting

In the third part of the assignment, we implemented the Insertion and the Quick sort algorithm in the given framework and also based on the Selectionsort.py file. In the **Selection sort** in each pass in an array, the algorithm looks for the largest value and when the pass is done it swaps the element to place it in the proper location. The time complexity for this algorithm is O(n^2), meaning that the larger the list, the slower the performance. The **Quick Sort** is an elegant algorithm that follows the *Divide and Conquer* strategy. It is considered to be faster than the *Selection Sort* since the average case has $O(\text{n} \log n)$ time complexity because each of the n items needs to be checked against the pivot value and the split point will occur close to the middle of the list. In the worst case scenario though, time complexity is O(n^2) if the division in partition is uneven. The performance of the algorithm depends on the pivot we choose according to the level of the array's sorting. For instance, if the array is semi-sorted and we choose the first element as a pivot, then it may be possible that the performance would be the worst case or it would be better if we used a random pivot. Also, we expect the Quick sort algorithm to perform better in larger lists. Finally, the **Insertion Sort** runs in $O(n)$ time in its best case, however it runs in O(n^2) time in its worst and averages cases. In an optimal case scenario, the algorithm will traverse through the list and compare every pair of elements before swapping them if they're out of order. Its best-case scenario comes from when it's fed an array that is pre-sorted; only traversal of the array is needed and indeed performed by the algorithm, resulting in the $O(n)$ time. Its space complexity is less if it's fully or partially sorted, meaning it's adaptive. The opposite being when the input array is prepared (or comes by happenstance) in decreasing order. We tested our algorithm 3 times for the sake of getting multiple readings and being confident in presenting an average time, swap number and comparison numbers. We processed an input size (n) of 151, starting from 30 and incremented by 20, for the three methods of shuffle, miniShuffle, and reverse. The average swaps and comparisons for these respective methods are shown below:

Table 1: Average number of operations and time for all shuffle methods and list size

Sorting Algorithm	Average Swaps	Average Comparisons	Average Time
Quick sort	2229.5	55314	0.0061482085
Selection sort	1869	99480	0.005940636
Insertion Sort	1813.67	3693.67	0.00707143

According to the results, the *Selection sort* may be the fastest algorithm, however the number of operations made, and specifically the comparisons, is way too high. The number of the operations for this algorithm is not affected by the three different scrambling options (shuffle, miniShuffle and reverse), but it only increases as the size of the list increases. It was expected to perform worse than the *Insertion Sort*, which is apparent when it comes to the number of the executed operations, but not when it comes to time complexity. The scrambling options do not highly affect the performance in this algorithm, but generally the *reversed* arrays need more swaps and comparisons and the *miniShuffle* less. Finally, concerning the more complex *Quick sort* algorithm, that was slightly slower than the *Selection sort* even though the operations were less than the latter. The reason for that could be the position of the pivot, which is [0], meaning it is the first element of the array. The results of our test also have shown that this algorithm performs better in randomly shuffled lists when pivot takes position [0]. According to the time in which each algorithm performs depending on the size of the lists, this increases in all algorithms as long as the size of the list increases. This is probably another proof that assigning the pivot as the first element is likely to lead to a worst case of time complexity. Generally, we expect the Quick sort algorithm to perform better with a randomly positioned pivot or in the middle point of a semi sorted list.

Table 2: Average number of operations and time for Quick Sort algorithm using pivot in position [0]

Scrambling method	Average Swaps	Average Comparisons	Average Time
Shuffle	146,57	809	0.001030407857143
miniShuffle	166,57	2200,28	0.002084360714286
reverse	89	4849	0.004648921571429