Assignment 2: Search, sort, and linked lists

Evelyn Kai Yan Liu, Kris Farrugia

February 19, 2021

1 Searching

The searching algorithms implemented were the sequential search and the binary search, both applied to normal lists and deque lists storing words. Furthermore, dictionary searches were compared to the mentioned algorithms.

With regards to normal lists, the sequential search was roughly three times slower than the binary search. This reflects the predetermined time complexities of these search types, the former with a complexity of *O(n)* whilst the latter with a complexity of *O(log n)*. Interestingly, by implementing an iterator on a sequential search (enumerator), the time taken was slightly decreased.

On the other hand, when executed on a deque list, the difference between the sequential and the binary was much larger. Moreover, the iterator in the sequential search played an important role in improving dramatically the time complexity (by 320 seconds). This seems sensible given that without an enumerator, the algorithm would be iterating over the range **and** indexing into the deque, creating a polynomial time complexity $O(n^x)$. Conversely, iterating only through the deque list creates a linear time complexity.

Finally, as anticipated, searching in dictionaries was the fastest method used, with the average time complexity of *O(1)* and worse time complexity of *O(n)*.

2 Sorting

For bubble sort, time complexity highly depends on the size of the list as well as its initial order. Since it traverses the list and exchanges pairs when the order is not correct, each comparison can cause an exchange, resulting in a huge amount of times for exchanges as well as comparisons.

Having run the algorithm with lists of different sizes, it is evident that as the size of the list grows, the number of time that comparisons and exchanges have to be made increases linearly. When the size of the list is comparatively small, for example 10, if one only shuffles the list in a small range (mini shuffle), the numbers of comparisons and exchanges made are on par with the ones from the selection sort algorithm. However, if the order of the items in the list is reversed, then one swap has to be made

for every two comparisons to sort the items. The experiment, therefore, has shown that the theoretical expectation and the implementation are aligned for the bubble sort algorithm.

The shell sort, based upon the insertion sort method, was the best performer when compared to the previously mentioned bubble sort and the selection sort algorithms. This was to be expected given that shell sort has the best time complexity, falling somewhere between $O(n)$ and $O(n^2)$. The main reason for that is due to its highly adaptive nature, meaning that it has the advantage to adapt in the current order and work faster within the existing elements. Essentially, this occurs when creating the sublists (with accordance to the chosen gap) and implementing the insertion sort on each of them. In turn, this process facilitates the second implementation of the insertion sort (requires less swaps and comparisons), this time however on the whole list.