Course: Advanced Programming for Language Technologists

Assignment 2: Search, Sort, and Linked Lists

Flor Del Rocio Ortiz and Laura Janulevičiūtė

February 18, 2021

1 Searching

It is the case that background processes running on a computer may slow down the results when we run our code, therefore the running time can vary. In order to get more accurate results, we calculated the average time of 5 runs.

Binary search turned out to be faster than sequential search for lists. The first and second implementations of sequential search returned 51.2569 and 53.4384. Sequential searches have a time complexity of O(n) because in the average case, we will find the item we are looking for about halfway through the list, making the size of our comparison list n/2. In the best case we will find what we are looking for in the first instance, and in the worst case in the last. On the other hand, binary search requires a sorted list, which would have taken additional time, had we not had a sorted list to begin with. In our case, however,the result for binary search was significantly faster than the sequential searches operating on lists, returning a result of 0.1354. Binary search is, in general, O(log n), because the maximum possible number of comparisons it has to make to arrive at the final result will grow logarithmically with respect to the size of the list.

The results for the two implementations of sequential search for deques varied significantly. The first implementation made use of indexing and returned a result of 335.8778. In contrast, the second implementation of sequential search was much faster, as it made use of the enumerate function instead, returning a result of 55.9173, which was much closer to the results of the two implementations on lists. This is due to the fact that the first function has to iterate through the list until it finds what it is set to look for, and once it has found it, the function will go back to the beginning to execute the same process again. The algorithm that uses enumerate can access the information available at a given index without having to run through the entire list. It is worth noting that time complexity will change depending on where the code is running. For example, running it on Google Colab will be much slower because it's online, as opposed to Spyder, for example. Interestingly, though the relations with respect to each other still hold true, the actual numerical results vary depending on the implementation of the test function.A test function that tests each function individually in general returns faster results, but we chose to use a test function that runs through every function five times to return an average time.

Dictionary search was the fastest because the key with its value can be accessed directly (O(1)).

2 Sorting

Time complexity for bubble sort is O(n^2) in the average case. The implementation is adaptive, given that the algorithm stops sorting if there was no swap during a given iteration, taking advantage of the existing order of elements so that the process will be faster for sequences with less disorder. Despite this advantage, bubble sort has the highest complexity in terms of number of swaps, although not necessarily in terms of comparisons in every case, as the list size increases. Bubble sort is the worst performing in the reverse method, as the number of swaps is considerably higher as the size of the list increases. For bubble sort, around 1225 swaps were made for list size 50, whereas for quick sort and selections it was 49 for each.

We implemented quick sort algorithm for our second sorting algorithm. Theoretically it is better to choose pivot randomly, however, in order to keep the implementation more straightforward, we chose the first element as a pivot. As per our lecture notes, we can see that quick sort is not necessarily adaptive. Quick sort will always run through the whole function, but will not do unnecessary swaps if they are not needed, whereas bubble sort will stop if it sees that no swaps were made in the last pass. That particular function of an algorithm could be said to "adapt" to its input. However, if we define adaptive as we did in our lecture and say that it necessarily means the algorithm will perform its task faster depending on the input it receives– in our case a presorted list instead of an unsorted one– then quick sort is not adaptive. Bubble sort, according to this definition is also adaptive. As a test, we tried running the example code from the textbook to see if quick sort would perform its task faster if given a presorted list. The result turned out to be slower than the result with a non-sorted list, even when the pivot value was the first term. When we did the same experiment with bubble sort (the adaptive variation), it was quite clearly much faster when using a presorted list. Given these two conditions and the way that quick sort performs (when the first element is used as the pivot value) we can conclude that quick sort is not adaptive. Time complexity for quick sort is O(n log n) in the average case. In the worst case, where the split points are skewed very much to either the left or the right, and not near the middle, it could be $O(n^2)$.