COMP26120 Lab 5

Michael Krasa

December 9, 2019

1 Complexity Analysis

1.1 Insertion Sort

The insertion sort takes the maximum amount of time if elements are sorted in reverse order. In this case the inner loop will have to shift all the sorted section of the array before inserting with each iteration and when looking for its correct position the for loop also has to iterate through the entire dictionary each time. This makes it very slow in cases where the dictionary is longer as an algorithm with quadratic time becomes slow very quickly. It is a stable sorting algorithm as it keeps the order of the elements it was given and doesn't reorder them during sorting. Since there is a while loop inside a for loop, this algorithm's worst case scenario complexity is $\Theta(n^2)$.

It takes linear time Θ(n) elements are already sorted. During each iteration the element is only compared with the high element and move onto the next one and finishes without any other nested loops which would increase the time taken.

Even though commenting on the average case is beyond this course, I argue that given a random dictionary and queries the time should be close to $\Theta(n^2)$. It is as the worst case scenario, since the algorithm has to do a lot of time-consuming comparing, swapping and most importantly the index changing for all elements with every single insertion, the only difference being that instead of searching all the way through the dictionary it might only need to go half way through it given that its an average case. For that very reason insertion sort is very useful when input array is almost sorted, only few elements are misplaced in complete big array.

1.2 Quick Sort

- Must define the recurrence equation and then solve it using something from the lecture

Quick sort is a divide and conquer algorithm, same as merge sort which we discussed in the lectures. It divides an array into two smaller sub-arrays. It can then sort the sub-arrays using recursion.

We can write the time for the quick sort as following:

$$T(n) = T(k) + T(n-k-1) + \Theta(n) \tag{1}$$

"T(k)" and "T(n-k-1)" are for two recursive calls and the last term is for the partition process. k represents the number of elements which are smaller than pivot in that recursive call. The time taken by quick sort depends on the input array (obviously) but also on partition strategy.

The best case occurs when the partition process always picks the median element as pivot. That way we would have two (near) equal length sub-arrays where the next pivot is easily determined. However, if a pivot isn't the median it doesn't mean that it will be that much slower, if at all. The time for the best case is Θ(n log n) which we can determine from the following equation:

$$T(n) = 2T({}^{n}/_{2}) + \theta(n) \tag{2}$$

The worst case occurs when the partition process picks the greatest or smallest element as pivot. If that were to happen the program would waste time by doing a an n number of passes before moving the pivot again. The time for that is $\Theta(n^2)$.

$$T(n) = T(0) + T(n-1) + \theta(n) \tag{3}$$

2 Experimental Analysis

In this section we consider the question:

> Under what conditions is it better to perform linear search rather than binary search?

2.1 Experimental Design

Before we design our experiment we have to keep several things in mind. First being the core difference between these two algorithms which is that binary search requires a sorted dictionary to perform as well as it does. Given this information we can safely assume that most other conditions will not have such an impact on our experiment's conclusion.

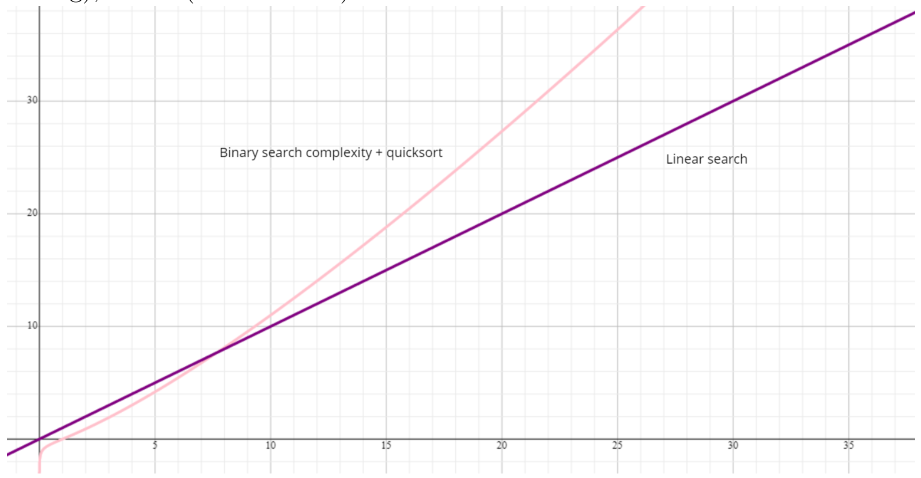
Knowing this, designing an experiment where this condition is true is quite simple. All we have to do is have an unsorted dictionary as this will have the most significant impact on binary searches performance. This means that a binary search algorithm will have to sort the dictionary first which is time consuming, especially in larger dictionaries which take longer to sort. We also have to decide on what sorting algorithm we'll use for the data set. For this application lets use quick sort because its best case scenario isn't necessarily an improbable outlier and the algorithm can perform very close to it in most cases. This yields us two simple equations showing us their relative average time complexities.

In this experiment we make the assumption that all the queries are in the dictionary, just to make the two algorithms a bit more closely matched so they don't have to necessarily look for items that aren't there. It wouldn't make much difference though since just the sorting alone of the dictionary would take longer than checking whether a query is contained within the dictionary in linear search. However, if we were to test for a very large number of queries it might be the case that linear search wouldn't be the fastest approach. This is because binary search only needs to sort its dictionary once and after that checking a large number of queries really doesn't take any time at all whereas linear search would be endlessly looping through the unsorted dictionary with every query.

2.2 Experimental Results

All we have to do now is put our two equations in a graphic calculator and see for ourselves. Since linear search's time complexity alone is linear we do not need to include a sorting time complexity in its function, however, we have to do just that for the binary search, I am using the average case scenario which is usually very close to the worst case one.
Linear search complexity = Θ(n)
Binary search complexity + quick sort = Θ(n log n) + Θ(log n)

Figure 1: A comparison of the functions n log n + log n (binary search + sorting), and n (linear search).



Looking at Figure 1 we can determine that given that the input is unsorted, it is better to use linear search for a dictionary that holds at least 8 elements. Since simply sorting the entire input alone would take longer than linear search for a larger dictionary, a linear search wins all the way in this scenario. However, this graph and functions alone do not define this experiment fully. Since we designed our experiment to have a small dictionary and a small number of queries it is quite constrained, under different conditions the end result would be more closely matched. That condition could be what I alluded to in experiment's design where if the number of queries is substantial enough, the functions we would need to compare would be different. Lets say we would have 100 queries. In a worst case scenario it would take linear search 100 n to check the dictionary for all the queries. Binary search with quick sort however, would sort the dictionary once and then do the binary search a hundred times. This wouldn't be a scenario where linear search wins which just shows how important it is to set up our initial conditions of the experiment correctly.

3 Extending Experiment to Data Structures

We now extend our previously analysis to consider the question

> Under what conditions are different implementations of the dictionary data structure preferable?

4 Conclusions