COMP26120 Lab 3

Anna McCartain

December 3, 2020

1 Initial Expectations

1.1 Insertion Sort

Insertion sort is an algorithm that sorts elements of an array one element at a time. It works as so; taking each element of the array, comparing it with each element before it, and swapping them if they are not in order. In my insertion sort algorithm, I first created a new array, initialising it with the first element of array to be sorted. Then for each element in the array, we add it to the new array, compare it with each element previous in the array, and swap addresses in the array accordingly. For my own reference in explanation I will add an example of how my code works.

Initially: array - CBA
Add to newArray element 0: newArray - C

Loop 1:

newArray - CB

Loop 1a:

COMPARE B with C

SWAP &B and &C if B less than C

newArray - BC

Loop 2:

newArray - BCA

Loop 2a:

COMPARE A with C

SWAP &A and &C if A less than C

newArray - BAC

Loop 2b:

COMPARE A with B

SWAP &A and &B if A less than B

newArray - ABC

As I have researched online the complexity of insertion sort - I have seen how other people have written this algorithm, taking the outer loop as iterating up the array, and the inner loop to compare this element with each before it. Their inner loop condition states: $While(array[j-1]/ > array[j] and j/ > 0)$ then swap element. This algorithm is much better than mine as it allows for this inner loop to not be executed if elements are in order. When dealing with complexity, this allows the best case to be O(n) - as the outer loop only iterates when the elements are already in order.

However, my production of the insertion algorithm is a little less efficient; as I have 2 nested for loops, both will always execute. The outside loop will execute n - 1 times, whereas the inner for will execute 1 + 2 + ... (n - 1) times . So (n - i) for each iteration of the outer loop, where i is in the range 1 to n - 1.

Example: see above. n = 3

outside loop iterates n - 1: 2 times

inside loop iterates I - used the summation of the natural numbers formula, minus n:

$n(n-1)/2$

3(2)/2 : 3 times

I also have another for loop to put new sorted array back into old array. This will run n times however is not nested. Change array loop iterates n times: 3 times

My swap function inside my inner loop is the dependable line; ie this is the line that changes how many lines are executed. If the If/COMPARE statement is not satisfied, the line will not execute. This will only change the number of steps by 1 value, eg. the inner loop would go from $2(n(n-1)/2)$ to $3(n(n-1)/2)$.

This change in the constant is insignificant when we start addressing large values of n.

Overall the steps required for the algorithm I have wrote are:

Best Case:

$= 5(n-1) + 2(n(n-1)/2) + 5n + 3$

$= n^2 + 9n - 2$

Worst Case:

$= 5(n-1) + 3(n(n-1)/2) + 5n + 3$

$= 3n^2 + 17n - 4$

Using rule 5 of of Big O notation:

If f(n) is of degree d, then f(n) is $O(n^d)$

We can decipher that the best and worst case complexity for my algorithm are actually the same - n^2.

In this case, due to the inefficiency of my algorithm, the average case is also n^2.

If I am looking at the best implementation of insertion sort (not mine) the average case I feel would be closer to the worst case n^2 as even if we're moving elements halfway through the array, that's still going to be closer to n^2 than n. If I look at my implementation however I can see that the only difference between best and worst here is constants - making the average case also O(n^2)

1.2 Quick Sort

Quick sort is a simple divide and conquer algorithm; we chose element at index 0 as the initial pivot, sort the other values around this pivot, and then recursively repeat this process with the 2 partitioned arrays either side of the pivot until the front and back index's of the arrays cross over, hence the partitioned arrays are all now of size 1. We can picture this algorithm working down through a tree structure as the array decreases in size. If we denote the time taken at each node to be v, then this value will be proportional to the length of the array.

The complexity of the sorting algorithm itself is O(n) as we have 1 while loop that iterates over the size of the array. When we perform quick sort, an array of size m

Gets broken into an array of size n1, n2, and x the pivot.

Making the time complexity of each layer $T(n) = T(n1) + T(n2) + c * n$
Complexity: n

The worst cases of quick sort occur when the pivot always ends up in the same location - the front of the array. This will occur with a sorted array, or an array of equal elements. In this case, the size of either n1 or n2, will be 0 and the latter (n - 1). When subbing this into the recurrence equation, it is that T(n2) = 0, T(n1) is always decreasing in size, along with the Complexity.
Level 1: T(n) = T(n - 1) + 0 + Complexity: n

Level 2: T(n) = T(n - 2) + 0 + Complexity: n - 1

...

Level X: T(n) = T(n - (n - 1)) + 0 + Complexity: n - (n - 2)

Overall, the summation of the complexity is the sum of natural numbers up to n, minus 1.

Complexity: $\frac{n(n+1)}{2} - 1$

Hence Complexity: $O(n^2)$

The best case of the quick sort algorithm, is where the pivot of each layer roughly ends up in the centre of each array. This allows for the comparisons at each level to be divided equally.
Hence

T(n) = 2T(n / 2) + Complexity: n

Each partition takes O(n) complexity, and we have on average log(n) partitions.

Hence in the best case complexity is O(nlogn)

The average case for quick sort I believe should be closer to the best case as if we are assuming that the numbers are reasonably uniform , then the pivot should in general partition the array into sections that are closer to n/2 in size than n - 1 in size.

1.3 Hash Set

In my hash set implementation, I have decided to implement linear probing only due to time constraints. Hash sets work by creating a value based on the data given to us. The key is then calculated from the modulo of the value with the character count of this particular data value. In my hash set, I have implemented 2 keys. The first just sums the ascii value for this data value whilst the latter raises each ascii value to the power of its position in data value.

Instead of counting up the primitive steps as I did in my analysis of insertion sort, I will give a more general thinking approach to the hash set analysis as to practice different techniques.

Inserting and finding in the best case analysis I believe would be the same O(1) as in this case. A the value would be passed to the hash key function, where a value would be calculated, then as this place in the hash set the required value would be found. For both hash keys it would be O(1) as even though hash key B uses 2 methods to raise ascii to a power, for an arbitrary large input n , this constant becomes insignificant.

The worst case for hashing I believe could be where:

* you have implemented an awful hash function that for some reason calculates each key to be equal (for example key = 0) and we have already inserted n items into hash set of size n. As there has already been n insertions, you hash set will now have to resize, doubling and copying all the elements over to new hash set. Your program now will go to element with key 0, and then using linear probing will iterate up through hash set trying to find next available space at n + 1.

I calculate this to be 2n + C – where the 2 is from resizing and finding next available space, and c from additional operations in program (new hash set initialisation) Making the worst case complexity O(n)

The average case complexity I believe for the hash set would also be very close to the best case complexity, as in most cases we have ensured a hash function and set size that computes the least collisions possible. Resizing the hash set also is a function that would rarely be used. Average = O(1)

1.4 Binary Tree

Binary search tree is the last data structure up for analysis. A binary search tree has 3 main operations; insert, delete and search. The best case analysis would occur for insert and search when we are referring to the root node; both operations should take O(1) time complexity. For deletion, the best case would occur when we are removing a node from the last depth layer; no additional structure changes have to be made, giving a complexity of also O(1) /para

Worst case analysis for inserting and searching would occur when the element we are searching for appears at the largest depth of our tree. (the height) At

worst the height will could be (n - 1) where n = size of the tree, s this is a very unbalanced tree. For deletion also this would occur when we are deleting the root node. We would have to in turn reset the pointers to restructure the tree. O(n) time.

The average case for binary search tree I feel would be somewhere in the middle. In general we are always trying to keep our tree as balanced as possible to give the minimum depth for a certain size n, as this would improve any functions time complexity on the tree. This would mean achieving a height as close to O(logn) The average case then I feel would be around this value, taking into account if the was balanced well, the the time complexity would vary from O(1) to O(logn) and if it was balanced poorly the range would become O(1) to O(n). On average this then would be approximately O(logn)

2 Experimental Design

2.1 What are the best and worst cases and what are their theoretical complexities

complexity O()	Worst	Best
Insertion Sort	n^2	n
Quick Sort	n^2	nlogn
Hash Set	n	1
Binary Tree	n	1

2.2 What average case will you consider, in what way can it be considered average

I will try to implement a uniform random shuffle by producing random strings.

2.3 Which data will you use to compute f(n) for best, worst, and average cases

In my experimental design, I want to apply the theoretical concepts I know about the algorithms to produce a range of results for both algorithms. The things I have decided to vary are as follows.

- The size of the dictionary
- The structure of dictionary e.g sorted or random
- The search algorithm for both sorting algorithms; linear or binary
- The position of the target word in the dictionary

2.4 SIZE

I am going to vary the dictionary size between 1000 and 10,000 to see how the time relatively changes for each search algorithm. I am predicting that the time gap between the 2 searches will increase as the size increases. This is due to the relationship between complexity n and complexity log(n). However I do have to take into account the fact that binary requires a sorted array and will additionally add this time on as well to get both results. I would also predict that for binary search tree will be of similar complexity to binary search algorithm. Hash set should stay pretty consistent independent of set size

2.5 STRUCTURE

I will repeat the experiment using both insertion sort and quick sort, to see which algorithm combination gives the best results. In terms of predictions for the sorting algorithms I have written, I believe quick sort will always come out on top. This is partly due to inefficiency of my insertion sort algorithm discussed above, giving a matching best and worst case. However, I feel that even if I had written the insertion sort to be of maximum efficiency, ie a best complexity of O(n), the likelihood of the insertion being closer to the best case is less than the likelihood of quick sort being closer to the best case complexity: O(nlog(n)). I think quick sort will be a lot more reliable as such in keeping a similar time rather than the insertion sort.

Hash set and Binary search trees have their sorting mechanism's already implemented in insertion phase. Instead if I have time I will try to produce some inputs more menacing for these data structures to sort - perhaps a dictionary set with words around 20 character mark that only differ by 1 character each.

2.6 POSITION

Finally I will vary the position of the target word in the dictionary. Linear search and hash set predictions are pretty self explanatory, giving a (mostly) linear relationship between position and time taken. I am going to have to do some testing to understand how the time changes for binary search trees and sorting algorithms by varying the target word.

Note - I just changed my quick sort implementation to the model one as I thought I has errors, turns out it was just my print statements and not infinite recursion. I have kept the new implementation, see old git submits for my own.

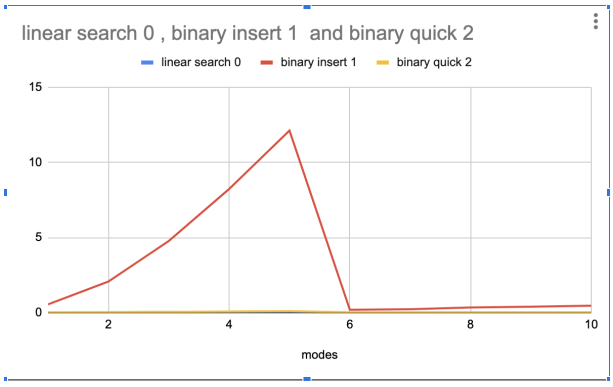
I will then use the data I have gathered and produce some graphs giving me the best, worst, and average cases for all the tests ran. I will then validate my findings using some larger inputs (100k) and seeing if the trends and patterns I have found, hold.

3 Results and Analysis

3.1 SIZE

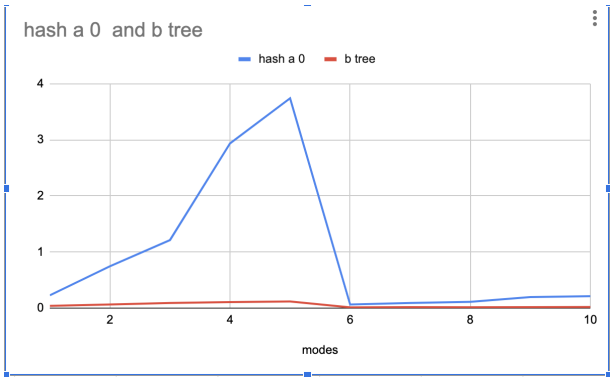
The first experiment demonstrating size, I ran a random produced string document of varying size through my program:
$python3 random_s trings.py > random$
$echo"x" > x$
$a = (10000200003000040000500006000700080009000 10000)$
$foriin"a[@]"; do tail -irandom > t; time./speller_d array - dt - m0x; done$line
Producing this table:

Modes	Linear Search 0	Binary Insertion 1	Binary Quick 2	Hash keyA 0	BS Tree
1000	0.033	0.555	0.034	0.227	0.039
2000	0.03	2.076	0.041	0.745	0.065
3000	0.047	4.772	0.057	1.212	0.091
4000	0.062	8.288	0.077	2.936	0.107
5000	0.081	12.112	0.11	3.739	0.117
6000	0.009	0.198	0.028	0.064	0.013
7000	0.012	0.236	0.03	0.09	0.018
8000	0.013	0.352	0.018	0.111	0.015
9000	0.013	0.4	0.021	0.196	0.017
10000	0.017	0.468	0.021	0.212	0.019



As we can see from these results here, linear search and quick sort with binary search are staying reasonably even as the number of dictionary entries increases - suggesting an O(1) relationship for these two when changing size only. On further anaylsis - binary quick from 1000 entries to 5000, has a increase of 1.0, 1.2, 1.67, 2.26, 3.2 respectively. This is saying as n increases by 2, 3, 4, 5 - the time taken is this much larger than the original time for 1000 entries. As quick sort is supposed to have a best case of nlogn. nlogn of 2, 3, 4, 5 is 0.6, 1.43, 2.4, 3.49. As you can see my results here are trending along a nlogn relationship which is means our quick sort is running at optimum. Insertion

sort from entries 1000 to 5000 is increasing at rate: 1.00n, 3.74n, 8.598n, 14.93n, 21.8n. As the entries are increasing by 1 each time, the insertion sort is showing an n^2 relationship; increasing roughly by 1, 4, 8, 16, 32 each time. This is showing worst case for insertion sort.



As entries increase from 1000 to 5000, the time scale for hash set increases by 1.00, 3.28, 5.34, 12.93, 16.47. This is showing roughly a 3n relationship meaning O(n). This is the worst case for hash set. Meaning the random string generator must have been producing multiple anagrams. My Hash code was independent on positioning however I can see potentially from the random string generator how multiple collisions have been caused. Binary search tree from entries 1000 to 5000 increase at factor 1.00, 1.67, 2.3, 2.74, 3.00. This is showing overall relationship (0.6n) meaning time compeity is O(n) - again this being the worst case for binary.

I wanted to also comment here on the fact that after 5000 entries all the times dropped right back down; I feel this is due to the way the random string generator works, maybe looping back around the string generation formula every 5000 entries? However I am not entirely sure.

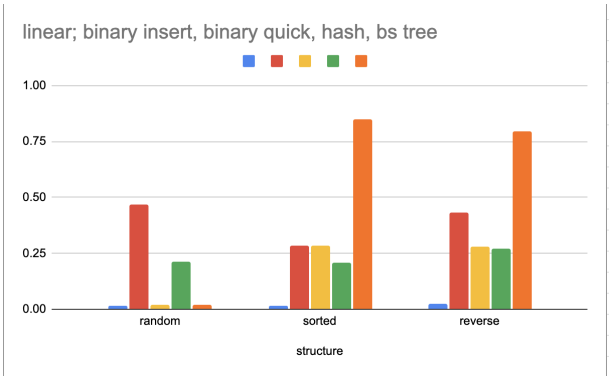
3.2 STRUCTURE

The second experiment I conducted demonstrated structure as I ran a 10000 word random strings dictionary in various ways:
$sort - drandom > sorted$
$sort - rdrandom > reverse$
for i in "a[@]"; $dotail$−i sorted ¿ t; time ./speller$_d array - dt - m0x; done$

Producing this table:

I do not have time to comment on everything I tested in experimental phase however I will try to cover the main points. Moving from a random to sorted dictionary in insertion sort, almost halves the time this is due to the sorted dictionary involving less comparison primitive operations. This is the ideal

Modes	Linear Search 0	Binary Insertion 1	Binary Quick 2	Hash keyA 0	BS Tree
Random	0.017	0.468	0.021	0.212	0.019
Sorted	0.015	0.284	0.283	0.208	0.851
Reverse	0.024	0.431	0.278	0.272	0.793



input for an insertion sort algorithms. Quick sort has slightly increased on both orderings of the input. This is interesting to me as the model solution takes the pivot from the centre meaning I would feel no elements would have to be exchanged either side and should take less time. If the pivot was taken from either end of the array this would make sense that sorted and reversorted takes more time as each end is already in position hence array size decreases by 1 each time. Hash key is staying pretty consistent for each structure as hash keys are independent of input structure. Binary search tree increases by a substantial amount for both ordered dictionaries. I feel this is due to the way bs tree is structured: if you had a set of data that was in order for a binary tree then the insertion would be similar to a linked list as each insertion would require no previous pointers to be changed. As a binary tree works best starting from middle element and branching down to create minimum depth perhaps having in a random structure is best case for this. Having an ordered data set is going to create a very unbalanced deep tree.

3.3 POSITION

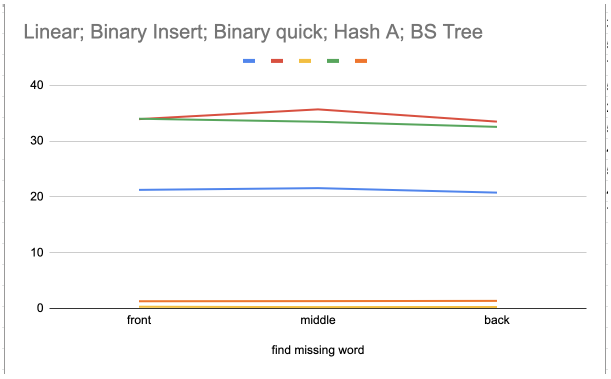
The final experiment I conducted was based of position of the word. I might add here that I attempted to use my other hash key function b, however during all experiments above it produced a segmentation fault. I am sure that this is due to the nature of the long strings being raised to the power of their position it was causing giant numbers to break the program. I decided to use a dictionary from usr/share/words – the Spanish one (this is sorted but makes the experiment closer to real world applications), adding the word beefymanriddims, into portions at front, middle and back.

Producing this table:

Modes	Linear Search 0	Binary Insertion 1	Binary Quick 2	Hash keyA 0	BS Tree
Front	21.243	33.938	0.305	33.995	1.28
Middle	21.57	35.682	0.243	33.456	1.303
Back	20.769	33.496	0.261	32.551	1.348

As another note, my additional hash function produced set fault once again on the Spanish dictionary. However I do know that it works (see below) I just think due to the power raising function, it does not work on large dictionary sets, hence was probably a bad hash function!
time ./speller$_h ashset - dsample - dictionary - m4 - vv./sample - file$
$Usingdictionary`sample - dictionary'$
$Checkingtextfile`./sample - file'$
$Usingmode4$
$Readingdictionary$

Dictionary read
Spellchecking:
1: twelve
Usage statistics:
PRINT STATS
Total number of cells is 509
Total number of used cells is 6
Total number of collisions on insertion are 3
0.002



These results are also quite interesting: Insertion sort seemed to increase when finding the word from the centre of the dictionary - this seems as incorrect behvaiour as due to the dictionaries binary search implemenation you would beleive it to be faster. However, I did not insert it directly in the middle and so it would have been more iterations to reach this point than an end. In general quick sort handled this data the most efficiently due the fast implementation of sorting with a middle pivot on a sorted set of words. The hash map did not

work well at all, I feel being due to the hashkey; my hashkey uses ascii codes but they do not rely on position in the word, meaning that anagrams in the spanish language would cause collisions, leading to alot of linear probing involved. My other hash key would have worked much better however I did not take into account the memory allocations for these big numbers. Again the time is actually slightly decreasing each time as word go further into dictionary but the position I dont feel affects the hashmap. Binary search tree is last slightly increasing in time each time, however again I feel this is perhaps just a coincidence in me running the experiment once as the values are all pretty close in value.

3.4 Conclusion

I am just going to sum up my points here at the end: for bs tree: Its not where the word to find is positioned that is affecting bs tree but really how the order of how the data structure is inputed. hence for a best case would be where the data is in a pre order traveral order hence allowing each data element to be inserted without reassigning pointers. For a hash table nothing is related to the order of value but infact the way you hash to reduce collisions. If I repeated this experiment, I would fix my second hash function (where position was factored in) to cope with large data sets and values (random strings) and begin to find how to reduce collisions as far as possible.