COMP26120 Lab 5

Anna McCartain

January 6, 2020

1 Complexity Analysis

1.1 Iteration 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, initializing 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.

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)

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 Hypothesis

Firstly I am going to discuss the theoretical concepts displayed for linear and binary search, in an effort to decipher a prediction as to what I think the conditions may be.

Linear search is a search algorithm that sequentially checks each element of a list until the desired element is found. Once it reaches the end of the list, it terminates unsuccessfully. The basic algorithm performs 2 comparisons per iteration; 1 to check if the incrementing integer has hit the end of the list, and 1 to compare the value at each position in the list with the desired value. Worst case complexity would occur if the value is not in the list at all, hence 2n steps must be undergone, giving a complexity of O(n) according to rule 1 of the big Oh rules.

IF d(n) is Of(n) then a * d(n) is Of(n) for a greater than 0

Best case scenario would be where the desired element is first in the list, hence only 2 * 1 steps are taken, giving a complexity of O(1).

Binary search is again a search algorithm that is applied only to sorted arrays. It always compares the target element to the middle the array and chops the array above or below this point according the outcome of the comparison. This search mechanism is more complicated yet more sophisticated. Thinking about the worst case scenario would be when until the array is chopped up to a size of 1.

Example: array A of 8 elements.

MIDDLE = A[3] ... CHOP LOWER * * * X * * * *

MIDDLE = A[1] ... CHOP LOWER * X *

MIDDLE = A[0] ... CANT CHOP X

Therefore, worst case steps would be (a * log(n)) and hence complexity = O(log(n))

Best case again would be the same as linear search; O(1), for when the target element is the initial middle element of the array.

2.2 Experimental Design

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 for linear search
- Approach used for binary search. ie different sorting algorithms
- The position of the target word in the dictionary

SIZE

I am going to vary the dictionary size logarithmically between 10 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.

.

STRUCTURE

For linear search I am going to input both sorted and random dictionaries. This is see the time differences between them. I don't feel the differences in time will be that different from the sorted and random inputs, however I will add some code into my linear search to define the upper and lower boundaries of the dictionary. This will allow the algorithm to exit initially if the target output is out of bounds hence decreasing the overall average time for sorted arrays.

.

SORTING

For binary search, 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.

.

POSITION

Finally for both algorithms I will vary the position of the target word in the dictionary. Linear search predictions are pretty self explanatory, giving a 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 however. My prediction hypothesis is as follows:

This method does also use the assumption that arrays are evenly spaced in terms of sorting. ie, the range between numbers doesn't differ too much.

If dictionary has an even number of entries, as dictionary is reduced the new truncated dictionary will become odd if target is in lower half, or even if target is in upper half. If in even half, process repeats partitioning dictionary into even and odd chunks depending on where target lies. If dictionary becomes odd size, then how the array splits depends on which odd number it is. Every other odd number (5, 9, 13) will split the array into 2 even arrays, (3, 7, 11) creating 2 odd arrays. I have defined this as:

If the dictionary size can be defined as 2n + 1, where n is natural number with odd parity, then the resulting arrays will also be odd.

In an odd sized array, the division boundary is always taken as the centre element. (the size / 2 - rounded) In just you could argue that if the array is odd, and the target lies on a decision boundary of the array size, then the likelihood of it being found is higher.

In an even sized array, the boundary is always taken as half of the size, hence again if it lie on this boundary it will be found quicker.

.

EXAMPLE

ARRAY 1 2 3 4 5 6 7 8 9 10 11 12 13 14

FIND 9

MIDDLE 1 2 3 4 5 6 8 9 10 11 12 13 14

HIGHER

MIDDLE 8 9 10 12 13 14

LOWER

MIDDLE 8 10

FOUND

So I could predict that, I know the upper and lower bounds of the array and the array size is 14 - EVEN so the decision boundary will be at element 7, which will using my assumption be around the integer 7. I know I will most likely have to go higher and hence the array now will become size 7 - ODD so the decision boundary will be the centre element, the 11th element. I am assuming the integer will be around the value 11 and hence I will most likely have to go lower. The integer 7 adheres to the property I discussed above about every other odd number and so I know the array will split odd. Again, the size is ODD so the decision boundary will be the centre element, the 9th element. I could now predict using my assumption that around 3 iterations of the quick sort algorithm will be required based on my assumptions above. I understand I

am not fully there with this concept, but as it is quite complex to get my head around, I have tried my best to understand it.

2.3 Experimental Results

SIZE & SORTING

Size	Linear	Binary_Insertion	Binary_Quick
10	0.00	0.00	0.0
100	0.00	0.00	0.0
1000	0.01	0.02	NA
10,000	0.06	0.66	NA

.

STRUCTURE

linear - size 1000

Structure	Time
Sorted	0.00
Random	0.01

.

POSITION

Size 1000

Position	Linear	Binary_Insertion	Binary_Quick
Front	0.00	0.02	0.01
Quarter	0.00	0.02	0.01
Middle	0.00	0.02	0.01
Back	0.00	0.02	0.01

Binary was given a sorted array here, hence the positions will be relatively the same as Linear Search.

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?

For binary tree and hash maps, I will be comparing their finding efficiency's in terms of size of dictionary, giving them a variety of search words, too see which one performs faster. I predict that hash will perform faster as hash key can usually be looked up in 1 iteration. I would except the time of the hash map to barely increase, whereas the binary tree lookup to be increasing. The best case for both structures lookup would be O(1) of course, as the element could be the exactly the key in the hash map, or the top element in the tree. Worst case in terms of the tree would be O(h) where h is the height of the tree. When building my hash map and testing it with the one letter dictionary, my key involved the modulus of the ascii code and position, hence for 1 letter values, they were all coming out with a key of 0, and had to use linear probing. The worst case complexity here would become O(n) as the maximum positions the value could be put away from the key is the number of elements in the dictionary. O(n) will always be greater than O(h) and so although the worst case is very unlikely, the binary tree would here be faster.

SIZE

Size	Binary_tree	Hash_Map
10	0.00	0.00
100	0.00	0.00
1000	0.00	0.00
10,000	0.07	0.07

4 Conclusions

4.1 Part 2

SIZE and SORTING

My result's were not quite as I expected. For the linear search, we can see an increase from 0.0s to 0.06s, showing a relative to the logarithmic scale, a reasonable time scale. For Binary insertion however, we can see the time frame increase from 0.0s to 0.66s - a much larger increase than the linear search; this is due to the fact however of sorting the large array of 10,000 entries and so reflects on the efficiency of the sort rather than the search. The binary quick sort showed some issues however; running seamlessly for the smaller dictionaries, but being unable to finish the sorting process for dictionaries over 1000 entries. I was not expecting this behaviour and went back and rechecked my code for errors. I could not spot a discrepancy and hence am concluding that the algorithm is just to inefficient to complete the sorting process.

.

STRUCTURE

Inputting a random vs sorted dictionary into the linear search, only increases the time frame by 0.01s. This is a much smaller value than I expected, and hence

would conclude that sorting the algorithm is not an efficient use of time, if the values position you are searching for is not roughly known.

.

POSITION

I ran this experiment for linear search multiple times, always seeming to get the value 0.0s regardless of the position. This is again not what I was expecting, but could have more to do with my perhaps poor choice of keeping the size constant at 1000 rather than 10,000 - where bigger changes in values can be observed.

Binary insertion and quick sort, I used a sorted array to keep the target positions constant. Again here, I was expecting the binary search to perform better, this could be due to again the sorting of the sorted arrays adding a fraction of time. For the middle value, linear was still faster than binary - which I know in theoretical concepts is incorrect.

4.2 Part 3

Again, my experiment churned out some unexpected results. I was predicting that hash map would perform quicker due to the sheer efficiency of the data structure, however they seem to have identical results for each dictionary size. Again this could be due to the data chosen; being closer to binary searches best case scenario.

I think overall my experiment was relatively successful, however I have gained a lot of knowledge on experimental design and feel I would approach the problem differently if asked to repeat this coursework.