CS260 - Data Structure
Programming Assignment 2: Autocomplete
Assignment Report

Fitsum Alebachew

January 21, 2022

Question 1 : 4 points

Data Structure: Answer the question below regarding the data structure you used in your implementation:

(a) [2 points] How did you store the contents of the file in memory?

Solution: I used an array of pointers to point to allocated structs that hold the word and weight info of each line in the file.

(b) [2 points] In terms of the size of the input file being defined as n, what was the **space complexity** (i.e., amount of memory used) of your application? Explain. (Note: use asymptotic notation)

Solution: The array of pointers is n bins long. There are n allocated structs holding the contents of each line in the file. The total memory used will be n*sizeof(pointer) + n*sizeof(struct Word). This is O(n).

Question 2 : 6 points

Sorting: Answer the question below regarding the sorting algorithm you used in your implementation:

(a) [2 points] Which sorting algorithm did you use in your program? Briefly explain how it works.

Solution: I used a quick sort algorithm to sort the array of pointers by using the word component of the structs they point to as the compare value. The algorithm breaks down the array into 2 with the last element as the pivot in each recursion, until the array can no longer be split.

(b) [2 points] In terms of the size of the input file being defined as n, what was the **runtime complexity** of your sort algorithm? Explain.

Solution: The worst-case scenario of the quicksort algorithm is $O(n^2)$. However this is for when the pivot picked is always the highest or smallest value. The array is randomly sorted(according to the words) so the runtime will be closer to a merge sort algorithm which is $O(n * log(n))$.

(c) [2 points] In terms of the size of the input file being defined as n, what was the **space complexity** of your sort algorithm? Explain.

Solution: Quicksort uses an inplace sorting so no additional space is required to store array elements except for temporary variables needed to store and swap elements. This is $O(1)$.

Question 3 : 6 points

Searching: Answer the question below regarding the searching algorithm you used in your implementation:

(a) [2 points] How did you search for an interval of words that all start with the same substring? Briefly explain how your algorithm works.

Solution: I used a binary search algorithm twice in a single function: once to find the starting index of the matches, and once to find the ending index. Binary search divides the search space in half with every iteration by comparing the middle value to the query given that the array is sorted.

(b) [2 points] In terms of the size of the input file being defined as n, what was the **runtime complexity** of your search algorithm? Explain.

Solution: Binary search has a worst case complexity of $O(log(n))$. This algorithm uses this twice, still making the runtime complexity $O(log(n))$.

(c) [2 points] In terms of the size of the input file being defined as n, what was the **space complexity** of your search algorithm? Explain.

Solution: Binary search does not need to allocate space to store elements in as it will only be comparing values by derefrencing a pointer. Thus the space complexity is $O(1)$.

Question 4 : 4 points

Overall:

(a) [2 points] Do you think your algorithm could have been improved? If so, how? What would improve: runtime, memory usage, both? Explain.

Solution: Using binary search is an efficient way of searching. Merge sort might be faster than quicksort but the space complexity would increase so quicksort is also a good option.

(b) [2 points] Assume that you have an input file of size 50 thousand entries. In terms of runtime, how long will your program take:

- if it searches for a single query word and then terminates?

Solution: The binary search will have a runtime of $O(log(n))$, the sorting algorithm will have a worst case of $O(n^2)$. The total runtime will also be $O(n^2)$ in the worst case since n^2 diverges much faster than $log(n)$ as n increases.

- if it searches for 1 million query words and then terminates?

Solution: Although the search will take a million times longer than searching for a single query, the complexity will not change in asymptotic notation: $O(n^2)$

Would you suggest a faster algorithm than yours for the first (or the second) case to improve runtime? Briefly explain.

Solution: When searching for multiple query words, it might be better to look for all the matches in a single run of the binary search algorithm, so that it doesn't have to transverse the array a million times. Instead of returning a single set of start and end indexes, it could return all by comparing the mid values to every query word with each recursion.