FIT2004 S1/2019

Assignment 1: Analysis

Nicholas Chong (29808146)

Introduction

This document will give a brief outline of my solution along with the worst-case space and time complexity of my solution. Each section will be about a single task in the assignment.

For reference, n is the total number of words in the writing and m is the maximum number of characters in a word. Furthermore, assume that I am discussing the **worst-case complexity for both time and space complexity** for any operations.

Task 1: Preprocessing

The function uses Python file handling methods in order to open the file and extract the contents into a single string variable. We initially start with a list for banned words, a list for banned punctuations, a temporary list used to hold words, and an empty string to hold the current word. We then create a variable to hold our current word and begin to iterate over each letter in the content string, if the letter is a letter in the alphabet, we add it to our word variable, building it up. When we encounter a character found in the list of banned punctuations, we append our word into the temporary list and reset the word to an empty string. The temporary array will now hold a list of words without any banned punctuations. To finish, we iterate over the list and append any non-banned words into our output list, returning the output list.

Worst-case Time Complexity

Reading the file's contents takes O(nm) time, having to read each letter in the file. Iterating over each letter (O(nm)) has a nested operation of checking if the word is in the banned punctuation list, which is always constant regardless of the input, so the nested operation is O(1). The second iteration takes O(n) as the temporary list can only hold up to n elements, with the nested operation once again checking if the word is in a list of constant size, hence being O(1). All of the other operations in the function are O(1) each, totalling to O(k).

Overall, the worst-case time complexity is…

O(nm) + O(nm) + O(n) + O(k) = **O(nm).** (where k is a constant)

Worst-case Space Complexity

The list of banned words/punctuations will always be created regardless of the input, being O(1) in space. Extracting the file contents to a variable will be O(nm) as the variable's size is dependent on the characters in the file. A temporary list will be created that can hold up to all the words in the file, so it is O(nm). The output list will be O(nm) as the list is an immediate subset of the temporary list. All other variables are O(1) each, totalling to O(k).

Overall, the worst-case space complexity is…

O(nm) + O(nm) + O(nm) + O(k) = **O(nm).** (where k is a constant)

Task 2: Sorting the words

The function will begin by making all the words in the list the same size, filling each character with a backtick (`) that represents a special character lower in value than the 'a' (i.e. 'a' > '`' == True) so that radix sort can work for words with different lengths. After all words have been padded into the

same size, radix sort is then applied on the list, applying counting sort with the key being the letter at a given index, iterating right-to-left for each word. Because every letter is a lower-case alphabet, we can define the offset such that ord('a') will be 0, so that 'a' will be at the beginning of the list. We then adjust the offset by 1 so that ord(``) will be 0 and therefore be of lower value than 'a'. The counting sort function will return the same list but rearranged and will then be reassigned to the list so that the changes take effect. Once all the words have been sorted, the backticks are removed.

Worst-case Time Complexity

Finding the largest element will be O(nm) as we would have to iterate over each word in the list and apply string comparison (O(m)). We then find the length of the word which will take O(m), having to iterate over each letter of the word. Filling each word with the backtick will cost O(nm) having to iterate over each word (O(n)) and filling each word (O(m)). Radix sort is then applied based on the length of the largest word (O(m)), with each iteration applying a counting sort that takes O(n + d) where d is the constant range for the ord values. Removing the backticks will then cost O(nm) with a similar reason as filling each word. All of the other operations are O(1) each, totalling to O(k).

Overall, the worst-case time complexity is…

$O(nm) + O(m) + O(nm) + O(m(n+d)) + O(nm) + O(k) =$ **O(nm).** (where d and k are constants)

Worst-case Space Complexity

The size of pre-processed words is assumed to be O(nm). Filling each word with a special character creates variables based on the number of words in the input list and then adding more characters based on the size of the word, so it is O(nm). In radix sort, counting sort is called, creating variables based on the number of words and the range of the ord values, being O(n+d). Radix sort runs for O(m) times, hence being O(m(n+d)). Removing the special characters will then create variables based on the number of words and letters, similar to adding the backticks, so it is O(nm). All of the other variables are O(1) each, totalling to O(k).

Overall, the worst-case space complexity is…

$O(nm) + O(m(n+d)) + O(nm) + O(k) =$ **O(nm)**. (where d and k are constants)

<u>Task 3: Showing the number of total words including the frequency of each word</u>

Iterating over the sorted words, we increase the total word count, accessed through the first element in the output list, and maintain the creation/updating of the words and their frequency. If the last element in the output list does not have the same word as the current word, the list is initiated with the word and a frequency of 1. If the last element in the list does match the current word, the frequency is increased.

Worst-case Time Complexity

The function iterates through every word in the sorted list (O(n)). Nested inside the loop, it applies string comparison between two words, being O(m) having to compare letter to letter to evaluate the comparison. Assume the print() operation takes O(1). All of the other operations are O(1) each, totalling to O(k).

Overall, the worst-case time complexity is…

$O(nm) + O(k) =$ **O(nm).** (where k is a constant)

Worst-case Space Complexity

The function creates and adds to the list based on the number of words in the list, respective of each word's number of letters, so the variable's size is O(nm). The input into the list is also O(nm). All of the other variables are O(1) each, totalling to O(k).

Overall, the worst-case space complexity is…

O(nm) + O(nm) + O(k) = **O(nm).** (where k is a constant)

Task 4: Showing the k top most words in the writing

The function begins by finding the largest frequency in the list and creating a table of empty lists based on the value found. This table will effectively contain the alphabetical words corresponding to how the heap progresses. A min-heap data structure is created and filled with the first k frequencies from the input list. With every incoming element from the iteration, the word is pushed into the words table at a list indexed by the frequency (e.g. an element with the word 'apple' at frequency 4 will have the word 'apple' added to the list at table[4]). We then iterate over the rest of the input list and replace the root frequency if the frequency is higher than the current root's frequency. With the old root, we also pop the last word in the list at that old root's frequency and adding the new word corresponding to the new root. By using a min-heap, we can ensure that every element below the root has a higher frequency than the root and by limiting the heap to size k, we can maintain the highest k frequencies. We then acquire all of the frequencies by repeatedly popping the root at the heap. Iterating over the frequencies, we pop the last word in the list at the specified frequency, appending it to a results list. Doing all of us will give the desired result.

Worst-case Time Complexity

Finding the largest frequency in the list requires having to iterate over every element in the input list which costs O(n). We perform k insertions into the heap, where each insertion is O(log k) because having to rise up the heap until the root at worst. Hence, it is O(k log k) to insert the first k frequency into the heap. Looping over the rest of the list will be O(n-k), potentially having to do a sink operation (O(log k)) for every element which will yield a total of O((n-k)log k). Iterating over k (O(k)) and extracting the minimum element (O(log k)) will cost O(k log k) in total. Getting the key list will iterate over all values of k so it is O(k). Iterating over the key list (O(k)) and popping the word at that specified index O(1) will take O(k). All of the other operations are O(1) each, totalling to O(p) .

Noting that **n >= k**, the worst-case time complexity is…

O(n) + O(k log k) + O((n-k)log k) + O(k log k) + O(k) + O(p) = **O(n log k).** (where p is a constant)

Worst-case Space Complexity

Passing the input list as a parameter will simply link to the original reference, hence no additional space is being taken by the input and hence it can be assumed to be O(1). Updating the table of words will take O(km) where there are always k words of max size m. A min-heap data structure is created to maintain the k frequencies. The underlying data type for the min-heap used is an array, hence an array is being created to contain k frequencies, so it is O(k). When iterating over the remainder of the list, the table is being popped and then appended, keeping the space constant. The key list will contain k elements hence being O(k). The results list then contains the k elements, each containing the words of max length m, and their frequencies. Hence the results list is O(km). All of the other variables are O(1) each, totalling to O(p).

Overall, the worst-case space complexity is…

O(1) + O(km) + O(k) + O(k) + O(km) + O(p) = **O(km).** (where p is a constant)