FIT2004: Assignment 1 Analysis

Task 1 – preprocess

Algorithm

The *preprocess* function begins by initialising a list of characters (punctuation and whitespace) and words (auxiliary verbs and articles) to be removed. The text file is then opened and scanned line by line, and every character of each of these lines are checked, and if the character is one of those to be omitted, it is ignored. If not, then the character is added to a string of characters, which is the current word. The algorithm also checks if the current character is a blank space (“ “), so that if this is detected, the current string of characters is added to the list of all preprocessed words if it is not in the list of words to be ignored, and then the current string of characters is initialised again to be empty, so that the previous steps can be repeated.

Pseudocode

```
1 |   set punctuation = [",", ".", "?", "!", ":", ";", '"', "\n", "\t", "\r"]
2 |   set text = open(filename)
3 |   set words = []
4 |
5 |   for line in text do
6 |         set word = “”
7 |         for character in line do
8 |               if character == “ “ then
9 |                     add word to words
10|                     word = “”
11|               elif character not in punctuation then
12|                     word += character
13|               end if
14|         end for
15|         add word to words
16|   end for
17|
19|   return words
19|   end function
```

Time Complexity

With *n* as the total number of lines in the text, and *m* as the total number of characters in the text file, by analysing each line of the pseudocode (which is very similar to what I wrote), we can determine the following:

- Lines 1-3, 6, 10, 12: O(1) because each of these tasks (setting variables, and general operations such as +, - etc.) do not depend on the input (number of words/characters)
- For loop at line 5: O(n) because the loops iterates over each line individually.
    - Nested for loop at 7: O(m) because here the code iterates through every character
    - Lines 8, 19: O(1): If and return statements are constant.
    - Lines 9, 15: O(1): Appending takes constant time, assuming the list is not one of the more complicated data structures that may require O(logn), O(n) or greater.

- Line 11: O(1) Although the built in function *in* iterates through every item, it's easy to say that it will take O(n) time, however since the input (punctuation) is known and that it is constant, this line will take constant time as well

Adding up everything above, we get: O(1) + O(n)*O(m) = O(nm) as required. Of course, we can add a constant *k* (O(knm)) to represent the time it takes for all the other constant operations on the algorithm, however Big O notation does not include constants.

Space Complexity

The algorithm above needs O(1) to store parameters such as the punctuation. However the two for loops in lines 5 and 7 require O(nm) space because they are dependent on the size of the input. Line 5 iterates through every line of of the text (which is dependent on how many words there are) and so requires more space to store the lines if the text is longer, so therefore requires O(n) space. The for loop on line 7 also iterates through every character in the line, so if the line is longer, it needs more space to store characters and hence also require O(m) space. For every line, we go through every character, hence the overall complexity of the algorithm is O(nm), leaving out the constant *k* which accounts for all the other constant operations in the algorithm.

Task 2 – wordSort

Algorithm

The *wordSort* function uses radix sort to sort the words in alphabetical order. It does this by first making every word the same length by adding as many special characters as needed at the end of each word so that every word is the same length. It then uses counting sort to sort the words in ascending order according to the last letter (where the special character is treated as though it appears before every other letter in the alphabet), then it sorts it again according to the second last letter, and so on. This works because counting sort is a stable sort, hence the order of the strings is preserved. The invariant at the end of wordSort is that the words in the list are sorted according to the nth letter, where n starts out as the length of the longest word - 1, and ends as 0. Another invariant is that the relative order of the words should be preserved.

Time Complexity

My *countSort* function runs in O(n) time because it traverses through the list linearly when it collects indices. And then *countSort* is run for every character in the longest word, hence the overall time complexity is O(nm) where n is the number of words, and m is the length of the longest word (characters).

Space Complexity

My *countSort* function requires O(n+k) time (where k is a constant, hence simplifies to O(n)) because it uses up space when it looks at an element of a list, hence requires more storage for longer list. *wordSort* however needs O(m) space so that it can store characters, hence if *countSort* is run for every character, the required space complexity is O(nm) (with all constants removed.s

Task 3 – wordCount

Algorithm

The *wordCount* function counts the frequency of each word by assuming that the list was sorted beforehand, so that if there are duplicates of the same word, they would be next to each other. The algorithm then checks consecutive words and increments counters accordingly so that the frequency of each number corresponds to how many consecutive appearances of the word there are.

Pseudocode

```
1 |   words += [‘’]
2 |   set frequency = [len(words)]
3 |   set i = 0
4 |   set j = 0
5 |
6 |   while j < len(words) do
7 |         if words[i] == words[j] then
8 |               j += 1
8 |         else:
10|               add tally to frequency
11|               i += j
12|               j += i
13|         end if
14|   end while
15|
16|   return frequency
17|   end function
```

Time Complexity

With *n* as the total number of words, by analysing each line of the pseudocode (which is very similar to what I wrote), we can determine the following:

- Lines 1-4, 7-8, 10-12, 16: O(1) because each of these tasks (setting variables, and general operations such as +, - etc., conditional and return statements) do not depend on the input (number of words/characters).
- While loop at line 6: O(n) because the algorithm traverses through the entire list linearly.

Adding up everything above, we get: O(1) + O(n) = O(n) as required. Of course, we can add a constant *m* (O(nm)) to represent the time it takes for all the other constant operations on the algorithm, however Big O notation does not include constants.

Space Complexity

The algorithm above needs O(1) to store general values used for incrementing. However, the while loop in line 6 requires O(n) space because they are dependent on the size of the input. The algorithm traverses through the entire list linearly and needs space to store the values it is comparing and hence needs.

Task 4 – kTopWords

Algorithm

The *kTopWords* function begins by finding the maximum frequency of the list, then it traverses through the list linearly (because the list is assumed to not be sorted by frequency) to find words that have the same frequency as the maximum. Everytime a match is found, it checks if there are enough matches (e.g. if k = 4, if the top 4 most frequent are already given) then the entire loop exits and returns the answer. If there are not enough matches, then the current maximum is decremented by 1 until enough matches are found. Exceptions are also raised if the list is empty, or f the list contains less elements than k.

Pseudocode

```
1 |   set topWords = []
2 |   set currMax = 0
3 |   for frequency in tally do
4 |         if frequency > currMax then
5 |               currMax = frequency
6 |         end if
7 |   end for
8 |
8 |   while len(topWords) < k do
10|         for words in tally do
11|               if words[frequency] == currMax then
12|                     add words to topWords
13|                     if len(topWords) == k then
14|                           exit for loop
15|                     end if
16|               end if
17|         end for
18|         currMax -= 1
19|   end while
20|
21|   return topWords
22|   end function
```

Time Complexity

With *n* as the total number of words, by analysing each line of the pseudocode (which is very similar to what I wrote), we can determine the following:

- Lines 1-2, 4-5, 11-14, 18, 21: O(1) because each of these tasks (setting variables, and general operations such as +, - etc., conditional and return statements) do not depend on the input (number of words/characters).
- For loop in line 3: O(n) because the list is traversed linearly
- While loop in line 8: O(log(k)) because the while loop runs depending on how frequent words are. For example, if there are a lot of words with the same frequency, which also happens to be the highest frequency, then topWords would fill up quicker than if the list contained words with varying frequencies. And the way we account for this is through $\log_{m,}$ where m is a constant.
    - Nested for loop in line 10: O(n) because this traverses through the list linearly.

Adding up everything above, we get: O(1) + O(n) + O(logk)*O(n). Since O(logk)*O(n) is the significant term, we ignore every other term and consider this one our Big O value. This term simplifies to: O(nlogk).

Space Complexity

The algorithm above needs O(1) to store general values used for incrementing, etc.. However, topWords will need to store a varying number of elements, hence this requires O(k) space. Nested inside this while loop, is another for loop which requires O(n) space to store the values of the frequency table it is traversing, hence the overall algorithm requires O(kn) space. The for loop used to find the frequency also requires O(n) space, however since the more significant term is O(kn), O(kn) is the Big O value for space complexity.