Complexity Analysis

n - number of words in dictionary
m - number of words in documents
e - number of possible edit-distance 1 words
k - average length of word

Task 3

Task 3 requires us to search up the dictionary for all words present in documents, which simply iterating through it will cause a complexity of O(nm), which is not ideal. It is decided that storing words into a hashtable for processing will provide a better runtime.

Words in the dictionary is stored into a hash table. The hash table is made where it is 1.5X size of total words in dictionary, to spread out words more evenly across the table to reduce number of collisions and improve runtime. Words in the linked list *document* is then iterated, while searching for the word if its present in the hash table. Using the function *hash_table_has,* if word is present, it will be printed out otherwise it will be printed out with a "?". Searching in the hash table will have an average case of O(1). The hash table is then freed.

Method	**Average complexity case**
Create and initialize Hash table	O(n)
Fill Hash table with words in dictionary	O(n)
Iterate words in Documents	O(m)
Find words in Hash table	O(1)
Free Hash Table	O(n)

Time complexity of task 3 : O(3n + m) ≈ **O(n + m)**

Task 4

The approche for this task requires the implementations of functions in task 1 - task 3. This requires smarter methods to find corrected words traversing through the whole dictionary every time, which involves hashing & making possible edit distance 1 words.

For this task, a hashtable is initialized with size 1.5X of *n.* Another two arrays of strings is made; **[1]** one for storing all words in dictionary in sorted order as decreasing probability of occurrence; **[2]** the other for storing all probably edit distance 1 words, which will be used later on. Based on a formula (len*53 + 26), the size of an array for edit-distance 1 words can be easier determined, hence array size of 13594 is chosen for the maximum assumed word character length of 256.

Each word in document will be iterated, and will be checked in the hash table, *if found* it will be printed out and have a complexity of O(1). If word is *not found*, all possible edit-distance 1

words will be produced and stored in array **[2],** all words in it will be then searched in the hash table, where only the word with the lowest index will be chosen (dictionary sorted by most used to least). This process will have complexity of O(e) or O(k*53 + 26).

If the no edit-distance 1 words is found, edit-distance 2 and 3 words will be searched just by iterating the array of dictionary **[1].** Using the function for task 1, the first word that comes up with edit-distance 2 and 3 will be stored; the loop will break for the case for ED 2, and the words will be printed out. If no words are found after going through the whole dictionary, the original word will be printed with a '?'. Since to find the edit-distance, a 2d array of length of both words must be made, with a complexity of O(k^2) for every word in the dictionary until word is found. Hence this will have a worst case of O(nmk^2), for all words without a match or ED 1. To compensate this and improve efficiency, I've limited it to calculation for edit distance for words with difference in length more than ±3, as that would be bigger than edit distance 3 which is irrelevant. (k << m << n) hence k will not significantly impact on the complexity. The two arrays and the hashtable will be freed after all words have been searched.

Method	**Best case**	**Worst case**
Create arrays and hashtable	O(n)	O(n)
Fill up arrays and hashtable	O(n)	O(n)
Iterate words in Documents	O(m)	O(m)
Find matching words in hashtable	O(1)	O(1)
Finding Edit Distance 1 words & Edit Distance 2-3 words	O(e)	O(e) + O(nmk^2)
Free Arrays and Hash table	O(n)	O(n)

Time complexity for task 4 (best, all matches) : **O(n + m)**
Time complexity for task 4 (average, ED1) : **O(n + m*e)**
Time complexity for task 4 (worst, no match) : **O(n + m*e + nmk^2)**

Other methods

Alternatively, I initially used quicksort and binary search to find words in the dictionary. The dictionary was stored into an array, which was then sorted by ascending order of alphabets. Time complexity for this method is on average O(n log n), while binary search is O(log n). Compared to hashing (O(n) + O(1)), it was not significantly slower. But for task 4, hashing is essential for the way i searched the words. Aside from searching the exact matching word, i also produced all possible *edit-distance 1* words, and searching them in the dictionary. This was impossible without hashing, as quicksort has changed the sorted order of frequently used words, hence words with *edit-distance 1-3* can only be found by using brute force, resulting in a significantly slower run time O(nm). Hence the average case for this method if used in task 4 is (half matching, half not) : O(n log n + ½ (m(log n) + nm)) ≈ **O(n log n + m log n + nm)**