Spelling Corrector Analysis:

Task 3:

The initial inputs for Task 3 were two linked lists; one was a dictionary (of size D) containing all the possible valid words, the other was a document of words (of size L) that were not necessarily valid according to the dictionary.
The task required, for each word in the document, that the dictionary be scanned to verify whether that word was valid or not. However if the dictionary was used as a linked list, this would mean that for each word in document, the dictionary would have to be scanned from the head to (on average) halfway through the dictionary - having an asymptotic time complexity of $O(D * \frac{L}{2})$ or $O(D * L)$. This would be too slow for dictionaries and/or documents of large scale.
A better alternative would be to implement a hash-table, which would have a much quicker look-up time. Although there is some initial layout in terms of putting everything that is in the dictionary linked list into the hash table, requiring $O(D)$ time. However once this pre-processing is done, for each word in the document list, only $O(1)$ time is required to look-up whether that word is in the dictionary (assuming hashing is done effectively). Thus the total asymptotic time complexity is $O(D + L)$, which is much better than $O(D * L)$, however this is at the expense of memory usage, with additional memory needed in order to create and store the hash-table.
For Task 3 these were the only two approaches considered, and it is clear from the asymptotic time complexity which is the best option.

Task 4:

Again, the inputs for Task 4 were two linked lists; one being a dictionary (of size D), and the other was a document of words (of size L). I also converted the dictionary list into a hash-table $O(D)$, so as to improve look-up time complexity. In order to determine whether a word was spelt correctly, I would check whether the word was in the dictionary hash-table - requiring $O(1)$ time for each word. If the word was not in the dictionary, the task was then to find the best possible correction (if one existed).
Naturally, the first check was to calculate all strings one edit away from the original word, of which there were $O(A * N)$, where N represents the average word length and A is the number of characters in the alphabet. I then put these inside a new hash-table called the edits-table, which contained all the edits that had been seen such that repeat computation was avoided. I then checked if each edit was in the dictionary hash-table, if it was I added it to a linked-list of valid words. Once all edits had been checked, I ran through the valid words list and printed the valid word with the position closest to the head of the original dictionary list. If there were no valid words, I then moved onto the second edit distance.
In order to do this, for each edit with an edit-distance of 1, I calculated all the edits 1 edit distance away from that edit, and inserted into a linked-list (assuming the new edits had not already been seen). For each of these edits, (of which there were $O(A^2N^2)$), I then followed the same process of checking whether they were in the dictionary, and printing the best-matched valid word. If there was still no valid word, the third edit distance was looked at.

In order to find the third edit distance, I used the naive approach of running through the dictionary list and calculating the edit distance from each valid word to the original misspelt word, taking on average $O(D * N * M)$ time (where M is the length of the word). If the edit distance equalled 3, then this word could be printed as the correction. Otherwise there was no valid correction within the dictionary.
For the third edit distance I also considered using the approach that was used for the first and second edit distances, however the asymptotic time complexity began to get very large ($O(A^3N^3)$), and for an 11-character word, more than 200,000,000 computations would be required. Thus I decided that for the third edit distance, the best approach was the naive one.
Another approach that was discovered whilst researching spelling correctors online, was using BK-trees, however this would require using a concept and data structure that I'm not entirely comfortable with, so my implementation would likely not have been optimal (or possibly even correct).
So, using my approach, in the worst-case, the asymptotic time complexity of Task 4 was $O(L) * O(AN + A^2N^2 + D * N * M)$, however it can be assumed that both N and M will be the same, as they are both average lengths of words, so the time complexity can be simplified to $O(L * D * N^2)$ in the worst case (assuming good hash-table behaviour).