Task 1

Task 1 was completed by having the method read from text file and concatenate each word into a final_string. This process is done for n words in the file, and the max length for a word in the text file is of size m. Therefore the time complexity of this process is O(nm).

Now final_string is a large string of the text file, again the program traverses this time through the entire string, character by character, which has complexity of O(m) and compares with “punct” list (array with punctuation). Since the array with punctuation is a fixed size, the process is O(cm) = O(m). The new modified string is now called “filter_string”.

Finally, the program traverses through filter_string character by character and every time it arrives at an empty space, it appends the word to a list by maintaining two counters. Then the words in this list are compared to the array consisting of auxiliary verbs. This process iterates through words list of size n and compares with each element in auxiliary verbs list of size k. So overall complexity of process is O(c*n) = O(n)

Thus the total time complexity of the algorithm is O(nm + c*m + k*n) which simplifies to O(nm). The total space complexity is also O(nm) since total size of input is n*m and output is the same. So O(nm + nm) = O(nm).

Task 2

Task 2 was completed by implementing a radix sort algorithm in order to sort the array of words in alphabetical order. This was achieved by forming buckets for duplicate items and placing them in their respective indices within the count array. The ordering of buckets is as follows the first character in the word has the highest weight when ordering, following from that the second character and so on.

In order to generalise this for any string the program finds out the length of the largest stringing the array, then starts to order from the final position of the character, for all the strings in the array. Thus shorter strings are going to be first in the ordered list.

As such Radix sort for words typically follows an O(m*n) time complexity, as counts are formed for n words with m number of characters. The space complexity is also O(m*n) as an array of n words with m number of columns denoting the characters is inputed into the algorithm.

Task 3

The first step for Task 3 was to count the number of words present in the array. The duplicates of wordSort array are removed (“nodup”), and used as an input for the algorithm. An array is created (bucket) and n number of buckets are appended the the array as well as a count array. This process has time complexity of O(n). The length of “nodup” is inserted into bucket at index zero using built-in .insert( ) function.(It is important to note that while .insert( ) has worst case time complexity of O(n), since it is inserted at the beginning it has O(1)).

Next the function iterates through the array and comparing the ith element with the (i-1)th element. If they are the same then increment count[n] by one, if not then increment index on count array by one.

This process traverses through the array of size n giving O(n) time, and each time the ith word is compared to the (i-1)th word, which is done in O(m). Thus total time complexity is O(nm). Total space complexity is also O(nm), as array of size n with max m number of characters is inputed into the algorithm, and a same sized array is outputted of size n +1 as tuples ([‘word’, count]). So total space complexity is O(nm + (n+1)m) = O(nm)

Task 4

In order to satisfy a time complexity of O(nlog(k)) a heap sort algorithm must be implemented. The greatest challenge of this task was maintaining stability, since Heap Sort is an unstable algorithm certain adjustments were necessary.

The first step to the solution was to form a max-heap of size k, and add the first k elements from the wordSort algorithm and form a max-heap based on their frequency. This process takes O(log(k)) time. Next iterate the array obtained from word sort from range (k+1, len(array)), and compare each element to the kth element. If the ith element in the wordSort array has a greater frequency than the kth element in the heap, then the final leaf is popped and the ith wordSort tuple is inserted at the root. Now that the heap structure has been broken, Heap Sort is implemented again to form a max-heap of size k. This process is repeated n times for n number of words. Thus the total time complexity achieved would be O(n*log(k)).

Stability is maintained by implementing binary search within the algorithm. During the heapify subroutine, the parent is compared with the left and right child nodes, if the count value for either are the same, then both tuples are searched in the wordCount array from Task 3, and their positions are compared. The tuple that yields a greater Binary Search value should go lower in the max-heap. Binary Search has a complexity of O(log(n)) where n is the size of the wordCount array.

Thus the total time complexity of the entire algorithm would be O(n*log(k) + log(n)) = O(n*log(k)).
The total space complexity for the algorithm is O(km), since a heap of size k is input into the algorithm, with m number of max characters. And the output is the same.
So O(km + km) = O(km)