Task 1
The first function preprocess gets rid of all the auxiliary verbs, articles, white spaces and punctuation. The approach I took into coding this function was first split the file in its lines, then line by line have a string containing a character in the alphabet, it would then append the next character if its in the alphabet, if not it would save that string to a list, reset it to an empty string and carry on. This part of the function has a time complexity of worst case O(nm + k), with k being some amount of white spaces and punctuation, so O(nm), note that .appends were used and have O(1) time complexity. The next part of this function was to take the list made from the first part and eliminate any word that was in a list of 'banned' words, this was done by going through the list O(n), and comparing if the word is in a list. The complexity is O(n) and not higher, because the list of 'bad' words is fixed, as it the alphabet in the previous part.
In terms of space complexity no list is made larger than O(nm), so the space complexity remains O(mn).

Task 2
The list of words is sorted via radix sort, as some words are as large as the largest word, the character 'a' is used to fill in the gaps so sorting can be done, a list was made to save the word and its original length, the space complexity of this does not exceed O(nm). The worst-case time complexity of the function wordSort is O(nm), as seen in the sort where a loop goes n times in a loop in m iterations. Once the words are sorted a loop, with time complexity O(nm), shortens each word to their original length.

Task 3
wordCount goes by a similar process to finding words in preprocess, it keeps a tally when the next word is the same as the current word when iterating through a list, if the next word was different, or was the last word in the list, It would save the word and its occurrence value to a list. The space complexity of this list O(nm + k), where k in this case would be all the counts by the words, but once again K < nm, so it can be ignored, therefore giving O(nm).

Task 4
kTopWords is required to out put the k top occurring words, as the is not sorted by their value you can grab the first k elements, nor can you sort, as these will not fill the Time complexity requirements. The approach in my function uses a min heap implementation of a list, as it has O(log K) time complexity to insert, worst case would have to be going through and adding each one if they were in increasing order, giving O(n log k) worst case complexity. There is a loop at the end of the function which ensure that the list is returned is the correct order (alphabetically if occurrence is same), by popping and appending, constant complexity, which overall has a O(k) time complexity, which isn't larger than O(n log k), the space complexity of this function is, O(nk), no list exceeds the orginal input size.