In this document n is used to represent number of words and m is used to represent length of the biggest word.

Task 1:

The pre-process function opens and reads the given file and returns a list of all words excluding articles and auxiliary verbs.

It first performs constant number of operations to open and read file into a string.

It then runs a loop for each character in the string. In each iteration of this loop, if the character is a separation (space, tab or newline) or a punctuation, then it saves whatever is held in tempString and assigns an empty string to it. Otherwise, it just appends the character to tempString. It performs a constant number of operations in each iteration, regardless of the input size.

Once the loop ends a list of all words is returned.

Everything outside the loop as well as inside the loop happens in constant time. The only thing that depends on input size is the loop, which runs for each character in the file, which would be maximum of m x n times, since we know that "The upper bound of the input size is O(nm)". As a result, this function has a time complexity of O(mn).

The function has a space complexity of O(mn) as ListOWords is the only variable affected by input (each word found is appended to ListOfWords). All other variables have constant space.

Moreover, function 'itemInList' technically has complexity of O(x) for x items in the list, but the function is only ever used with lstSep or lstSep, both of which have constant space, resulting in complexity of O(1) when used by preprocess.

Task 2:

Function 'cord' has time complexity of O(1) as all operations are constant regardless of input size. Space complexity is also O(1) as size of variables does not depend on the input either. The function uses char as the index to retrieve the corresponding character in lstChar.

Function 'wordSort' uses radix sort to sort all items. First 0's are added at the end of all strings to make them all the same size.

```
    # find m
    m = 0
    for i in ListOfWords:
        if len(i) > m:
            m = len(i)
    # create lst with all words of equal length (by adding zeroes at the end)
    lst = ListOfWords[:]
    for i in range(len(lst)):
        while len(lst[i]) < m:
            lst[i] = lst[i] + '0'
```

Calculating m has time complexity of O(n) as the for loop runs entire length of list and in each ieration constant number of operations operations are performed.

Creating 'lst' (adding 0's at the end of all words) has complexity of O(mn) as a for loop runs entire length of the list and a while loop inside the for loop runs a maximum of m times in each iteration of the for loop. The space complexity does not exceed O(mn).

```
    # radix sort
    lstCount = [0]*27
    lstIndex = [0]*27
    output = [0]*len(ListOfWords)

    for n in range(m-1, -1, -1):
        #lstCount
        for word in lst:
            lstCount[ cord(word[n]) ] += 1
        #lstIndex
        runningTotal = 0
        for i in range(len(lstCount)):
            lstIndex[i] = runningTotal
            runningTotal += lstCount[i]
        #output
        for word in lst:
            o = cord(word[n])
            output[lstIndex[o]] = word
            lstIndex[o] += 1
        # reset values
        lstCount = [0]*27
        lstIndex = [0]*27
        lst = output[:]
```

Performing radix sort has a time complexity of O(mn). The outer for loop runs m times. The first for loop inside the outer loop (to create lstCount) performs constant number of operations n times. The second for loop (to create lstIndex) performs constant number of operations len(lstCount) times, which is a constant number of operations since length of lstCount is 27 regardless of the input. The third for loop (to create output) performs constant number of operations n times (for each item in lst). There are also other constant operations inside the for loop. Since the outer loop runs m times, and two inner loops run n times while one of them performs constant number of operations, time complexity for performing radix sort is O(mn). Once these are created, it uses these lists to create output. As output is updated, the corresponding index in lstIndex is incremented to represent the new position. Finally, lst becomes output and output is reset.

The outer loop works backwards from m. In each iteration, it creates lstCount (which holds frequency of each character) and lstIndex (which holds the index for each character).

```
    # remove all 0s once you're done
    for i in range(len(output)):
        while output[i][-1] == '0':
```

```
            output[i] = output[i][:-1]
```

Finally, all 0's are removed at the end. The outer loop runs n times (length of output). Inside the outer loop is a while loop that performs constant number of operations a maximum of m times for each i. Hence, time complexity for removing 0's is O(mn).

Space complexity for the entire function would be O(mn) as largest input dependant variables are lst and output which contain n words, each with a maximum length of n.

Task 3:

Function wordCount finds the number of each item in the given list. There are some constant number of operations throughout the function.

```
    for i in range(1, len(sortedList)):
        if sortedList[i] == tmp:    # next item is same as before
            count += 1
        else:   # next item is different from previous item
            returnList.append( [tmp, count] )
            count = 1
            tmp = sortedList[i]
```

The function has a for loop that performs constant number of operations n times, hence has a time complexity of O(n). (for each item in sortedList). Since the list is sorted, it runs through the list linearly and increments count each time it finds the same word. When it finds a different word, it updates returnList and resets count to 1.

This function has a space complexity of O(mn). The biggest variable that depends on the input is returnList, each item in which contains a word with maximum m letters, as well as an integer for each word. Hence space complexity comes down to O(mn).

Task 4:

```
    # add index values to wordcount
    for i in range(len(wordCount)):
        wordCount[i].append(i)
```

Index values are added to wordCount, where the loop runs n times.

```
# build minheap with first k elements in array
    minheap = []
    for index in range(k):
        minheap.append(wordCount[index])
        i = len(minheap)-1 # i = last index of nlst
        while i != 0 and minheap[i][1] <= minheap[(i-1)//2][1]: # while node
<= parent: swap (this way new nodes end up on top)
            minheap[i], minheap[(i-1)//2] = minheap[(i-1)//2], minheap[i]
            i = (i-1)//2
```

A minheap is created with first k elements in wordCount, where the outer loop runs k times and inner while loop runs log(k) times.

```
    # fit remaining elemts in minheap
    for i in range(k, len(wordCount)):
        if wordCount[i][1] > minheap[0][1]:   # if word belongs in minheap
            minheap[0] = wordCount[i]   # replace root with next item in
wordCount
            RearrangeRoot() # find appropriate position
```

Remaining items are added to the root if they meet the criteria and then RearrangeRoot is used to find their position. Outer loop runs n-k times. In each iteration RearrangeRoot is called.

Rearrange root performs constant operations logk times.

Finally, a loop runs k times.

```
    # now that heap of size k is formed, pop and add to returnlist in reverse
order
    returnlist = [None] * k
    for i in range(k-1, -1, -1):
        returnlist[i] = minheap[0]
        minheap[0] = minheap[-1]
        minheap = minheap[:-1]
        RearrangeRoot()
```

Hence, time complexity is O(nlogk) and space complexity is O(k) (since heap is the largest input dependant variable).