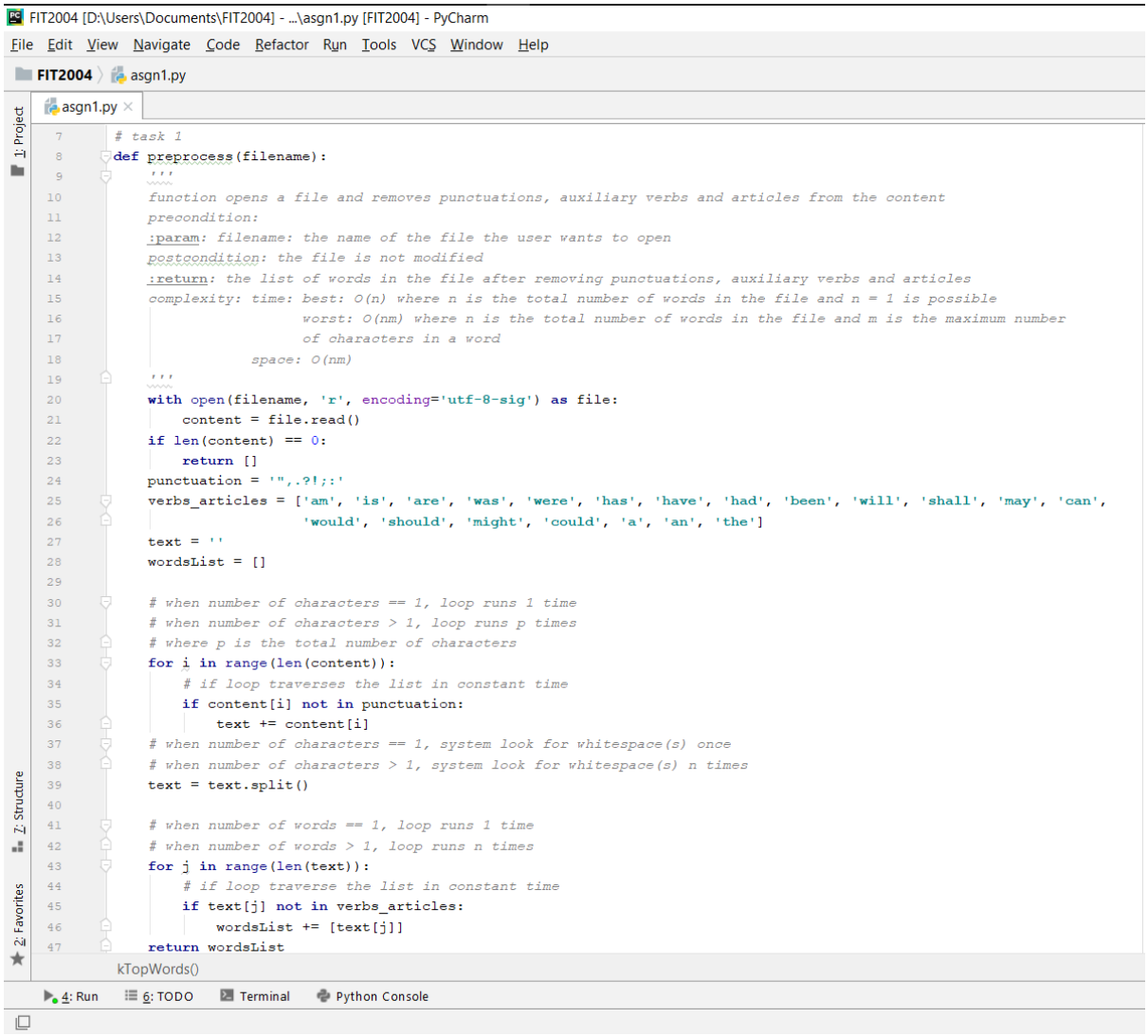
ID: 28390121

Name: Priscilla Tham Ai Ching

FIT2004 Assignment 1: Analysis

TASK 1




```python
# task 1
def preprocess(filename):
    '''
    function opens a file and removes punctuations, auxiliary verbs and articles from the content
    precondition:
    :param: filename: the name of the file the user wants to open
    postcondition: the file is not modified
    :return: the list of words in the file after removing punctuations, auxiliary verbs and articles
    complexity: time: best: O(n) where n is the total number of words in the file and n = 1 is possible
                      worst: O(nm) where n is the total number of words in the file and m is the maximum number
                      of characters in a word
                space: O(nm)
    '''
    with open(filename, 'r', encoding='utf-8-sig') as file:
        content = file.read()
    if len(content) == 0:
        return []
    punctuation = '",.?!;:'
    verbs_articles = ['am', 'is', 'are', 'was', 'were', 'has', 'have', 'had', 'been', 'will', 'shall', 'may', 'can',
                      'would', 'should', 'might', 'could', 'a', 'an', 'the']
    text = ''
    wordsList = []

    # when number of characters == 1, loop runs 1 time
    # when number of characters > 1, loop runs p times
    # where p is the total number of characters
    for i in range(len(content)):
        # if loop traverses the list in constant time
        if content[i] not in punctuation:
            text += content[i]
    # when number of characters == 1, system look for whitespace(s) once
    # when number of characters > 1, system look for whitespace(s) n times
    text = text.split()

    # when number of words == 1, loop runs 1 time
    # when number of words > 1, loop runs n times
    for j in range(len(text)):
        # if loop traverse the list in constant time
        if text[j] not in verbs_articles:
            wordsList += [text[j]]
    return wordsList
```

Task 1 requires a function *preprocess* which will take a filename of the input file and return a list of words in the file after removing auxiliary verbs, punctuation and articles. However, the file itself should not be modified.

The in-built function *open()* reads a file as it takes a filename with its extension which we can assume it as an operation that cost constant time complexity represented in big O notation as O(1).

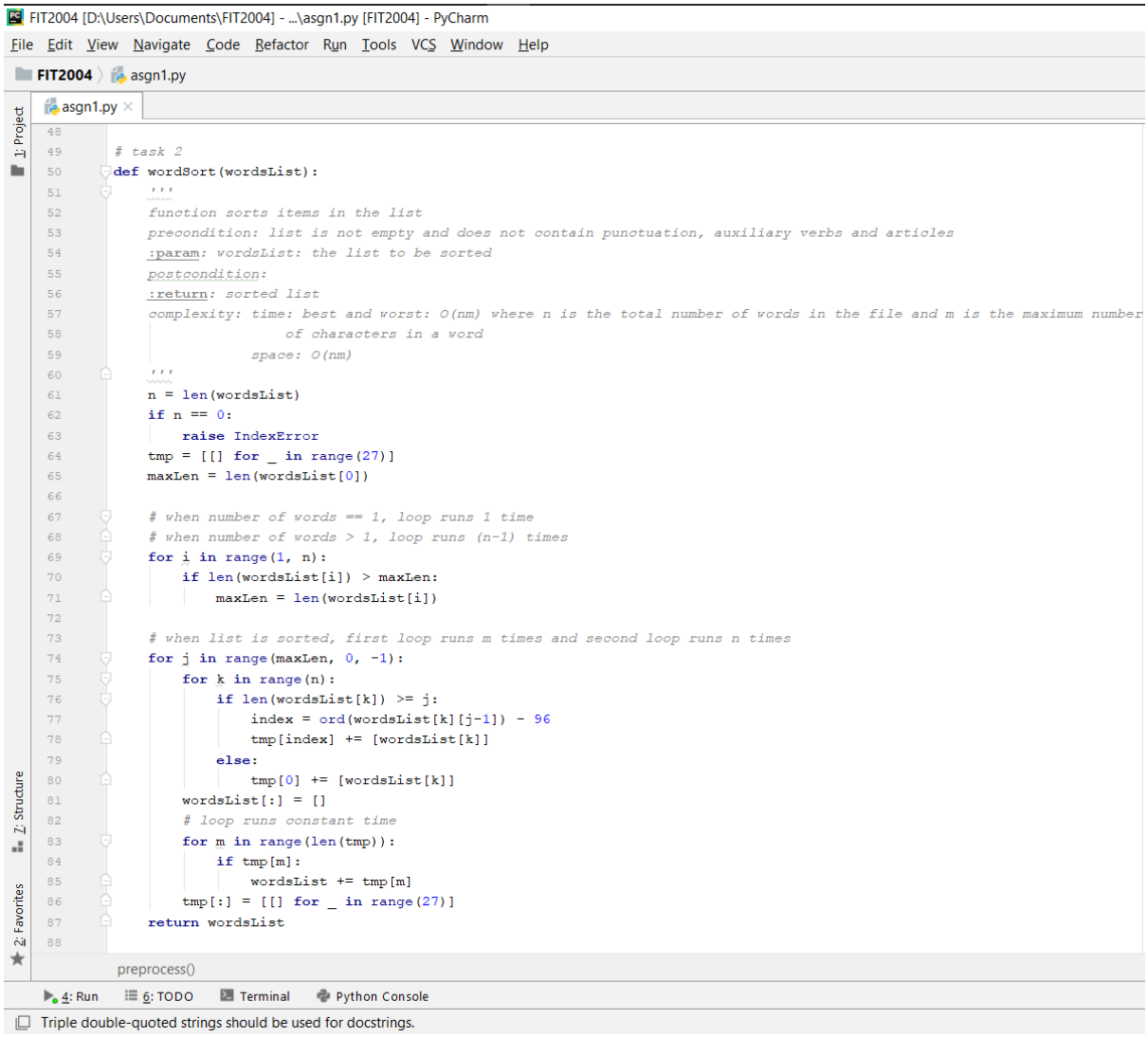
The first *for* loops through the characters in the file denoted as p. Then, the *if-statement* is a ‘loop’ to compare strings in the file with items in the list containing punctuation. This comparison is constant as both the string and the item are always of length = 1.

This concatenated characters are then separated by whitespace(s) using the in-built function split(). The function looks for whitespace(s) between the words (end of each word). So, consider n to be the number of words then, the function looks for it n times. The total time complexity so far is p + n + 7.

The second *for* loops through the words in the file, again, denoted as n. Then, the *if-statement* is a 'loop' to compare strings in the file with items in the list containing auxiliary verbs and articles. So, consider m to be the maximum number of characters in a word then, the comparison cost is m where m equals the maximum 'loop'. Therefore, the total time complexity is (nm) + n + p + 8, ignoring the constant we get, (nm) + n + p. We know that nm > p > n, hence, O(nm) is the worst time complexity.

Ignoring the space taken by the list of punctuation and the list of auxiliary verbs and articles for it is constant, the list of characters and the list of words are affected by the file input. Appending to the list of characters cost p and appending to the list of words cost n so total space complexity is p + n. However, since strings are list itself, it cost m when appending to the list of words where m (maximum number of characters in a word) equals to the maximum size of list inside the list of words. Hence, O(nm) is the worst space complexity since nm > p > n.

TASK 2



Task 2 requires a function *wordSort* which will take a list of words and return an alphabetically sorted list of words. The algorithm of radix sort is universal so, we would skip on further elaboration but the implementation to unequal length of string here, is done by comparing the length rather than adding space or special characters behind to make them all equal length. For strings that we cannot compare alphabetically (length of the string is < the length of the longest string), those strings are placed in the first position of a temporary list to be compared later when the point of comparison is not the last alphabet of the longest string.

The first *for* loops through the words in the list denoted as n. Then, the *if-statement* compares the length of one word to another with the assumption that the first word in the list is the longest. This uses the in-built function *len()* which we can assume it as an operation that cost constant time complexity represented in big O notation as O(1). The total time complexity so far is (n-1) + 5.

The second *for* loops length-of-longest-word times. So, consider m to be the maximum number of characters in a word then, the loop runs m times. There are two *for* loops within. The former loops through the words in the list, again, denoted as n. Then, the *if-statement* compares the length of a word to the counter of the outer loop which starts from the length of

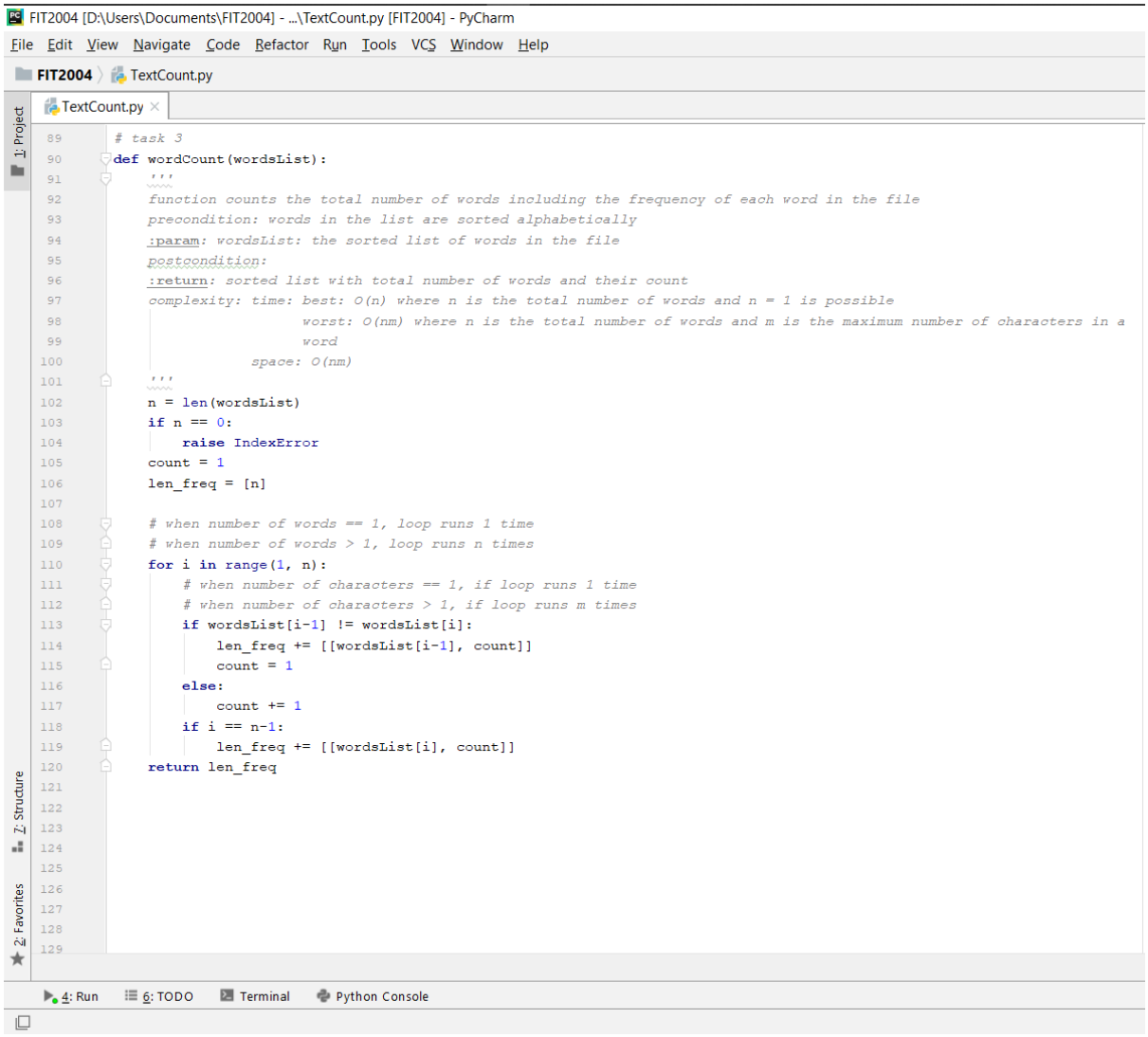
the longest word. This uses the in-built function *len()* which we can assume it as an operation that cost constant time complexity represented in big O notation as O(1). The condition within calculates the index of the alphabet (a = 1 and so on) to fill it into the temporary list. This uses the in-built function *ord()* which we can assume it as an operation that cost constant time complexity represented in big O notation as O(1).

Once the former loop is over, the list of words will then be sliced to remove the items in the list. This operation is in constant time as a complete slice is performed where list traversal is not done. The latter loops through the temporary list of size 27. This is also an operation of constant time as the size is constant referring to the English alphabets. Once the latter loop is over, the temporary list is also sliced to remove the items in the list and the time complexity is also as explained for previous slicing. The total time complexity of both loops inside is n + 27 + 7 = n + 34.

Therefore, the total time complexity is (n-1) + 5 + m(n + 34) = nm + n + 34m + 4, ignoring the constant we get, nm + n + 34m. We know that nm > m > n, hence, O(nm) is the worst time complexity.

Ignoring the space taken by the temporary list for it is constant, the list of words is affected by the file input. Since list is mutable, the list of words passed as an argument here is a reference to the returned list from task 1. Hence, as explained from task 1, the worst space complexity is O(nm).

TASK 3



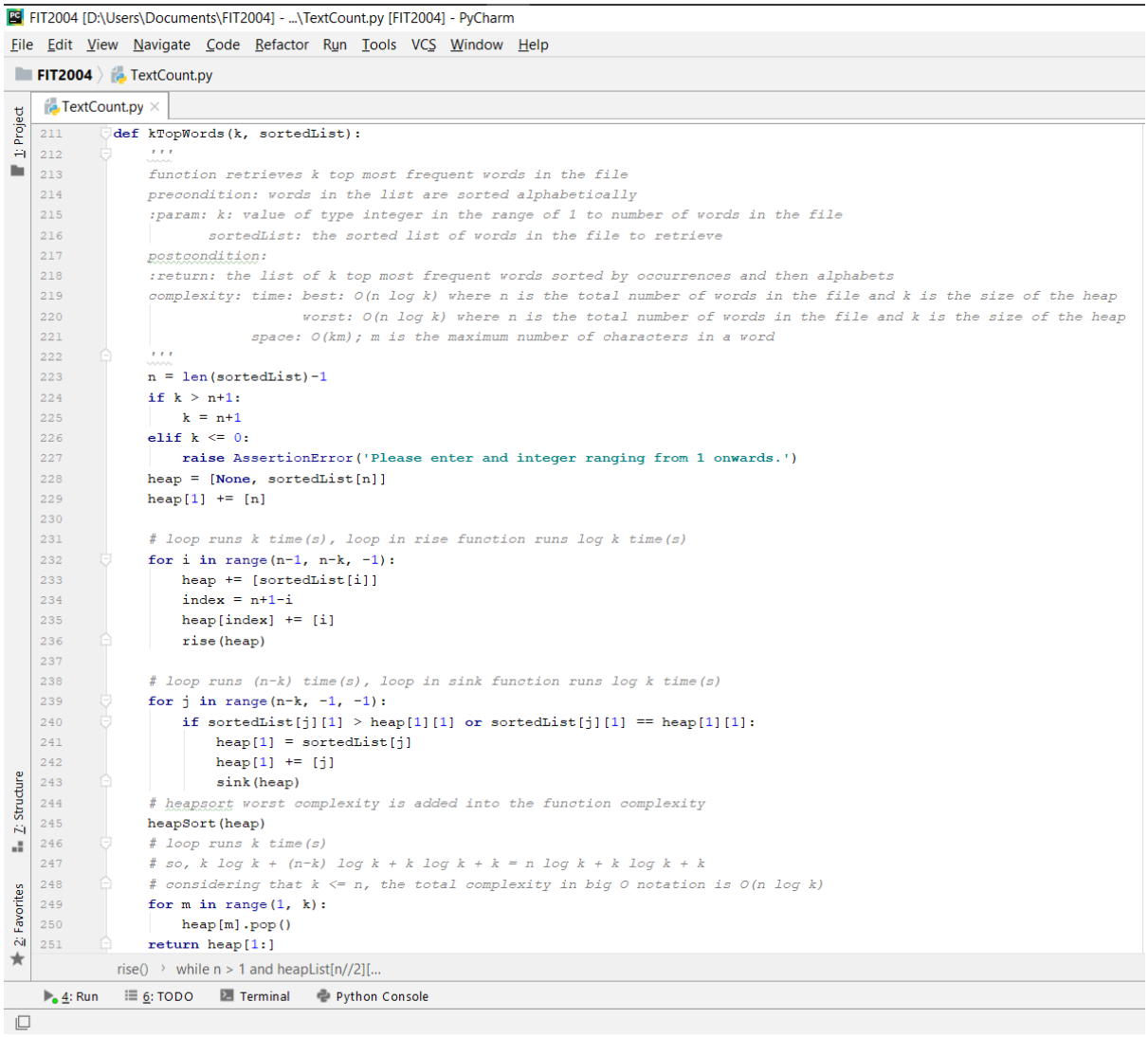

```python
# task 3
def wordCount(wordsList):
    '''
    function counts the total number of words including the frequency of each word in the file
    precondition: words in the list are sorted alphabetically
    :param: wordsList: the sorted list of words in the file
    postcondition:
    :return: sorted list with total number of words and their count
    complexity: time: best: O(n) where n is the total number of words and n = 1 is possible
                      worst: O(nm) where n is the total number of words and m is the maximum number of characters in a
                      word
                space: O(nm)
    '''
    n = len(wordsList)
    if n == 0:
        raise IndexError
    count = 1
    len_freq = [n]

    # when number of words == 1, loop runs 1 time
    # when number of words > 1, loop runs n times
    for i in range(1, n):
        # when number of characters == 1, if loop runs 1 time
        # when number of characters > 1, if loop runs m times
        if wordsList[i-1] != wordsList[i]:
            len_freq += [[wordsList[i-1], count]]
            count = 1
        else:
            count += 1
        if i == n-1:
            len_freq += [[wordsList[i], count]]
    return len_freq
```

Task 3 requires a function *wordCount* which will take an alphabetically sorted list of words and return a list with two values: (a) the total number of words and (b) a list of words with their count. However, the list of words should remain sorted.

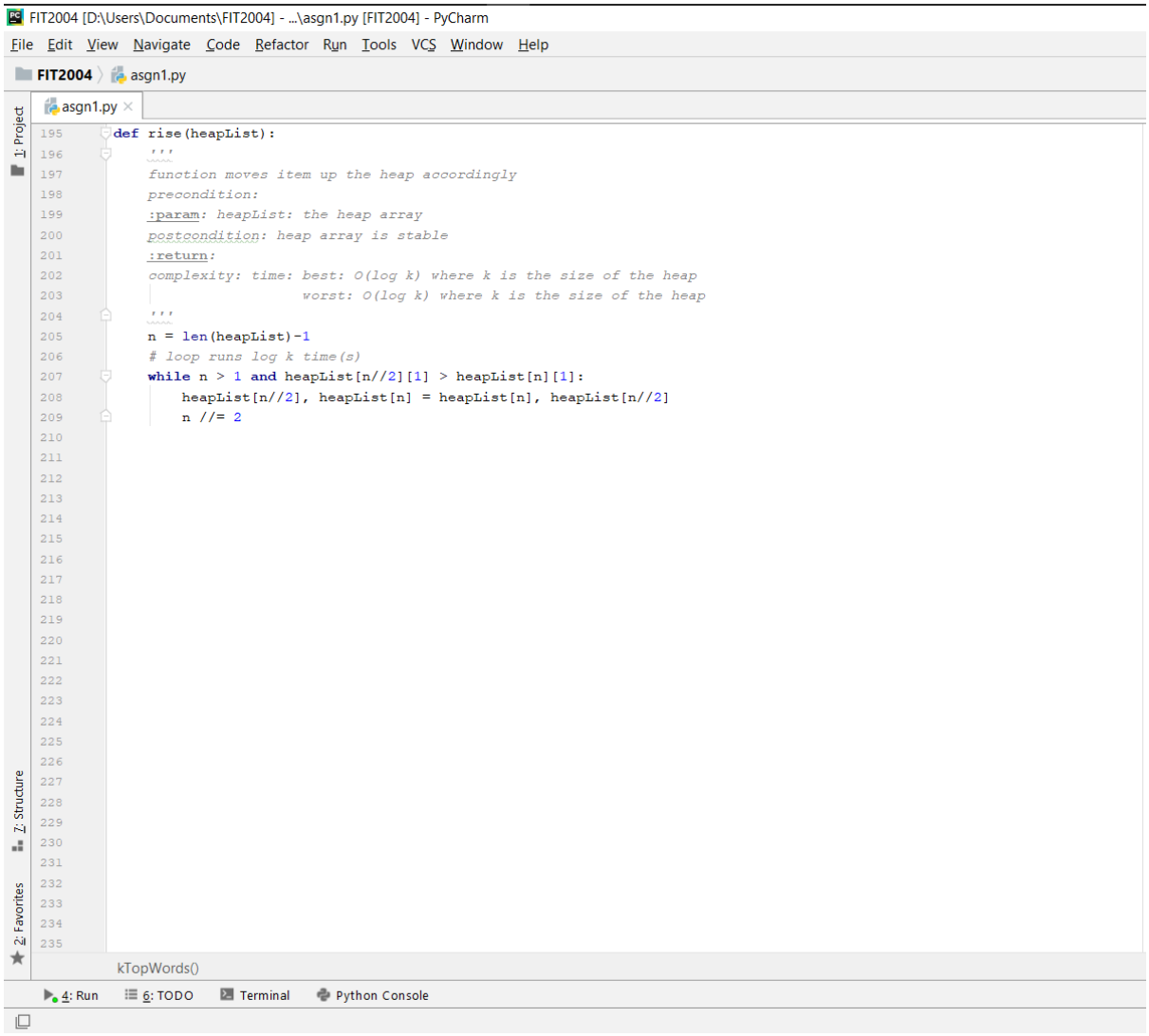
The *for* loops through the words in the list denoted as n. Then, the *if-statement* is a ‘loop’ to compare previous word to the next word in the list. So, consider m to be the maximum number of characters in a word then, the comparison cost is m where m equals the maximum ‘loop’. Therefore, total time complexity is nm + 6, ignoring the constant we get, nm, hence, the worst time complexity is O(nm).

The list created to return is affected by the file input and is of size (n+1). Appending to the list will cost n so total space complexity is n+1 (one space for the total number of words). However, since strings are list itself, it cost m when appending to the list where m (maximum number of characters in a word) equals to the maximum size of list inside the list of words. Hence, O(nm) is the worst space complexity.

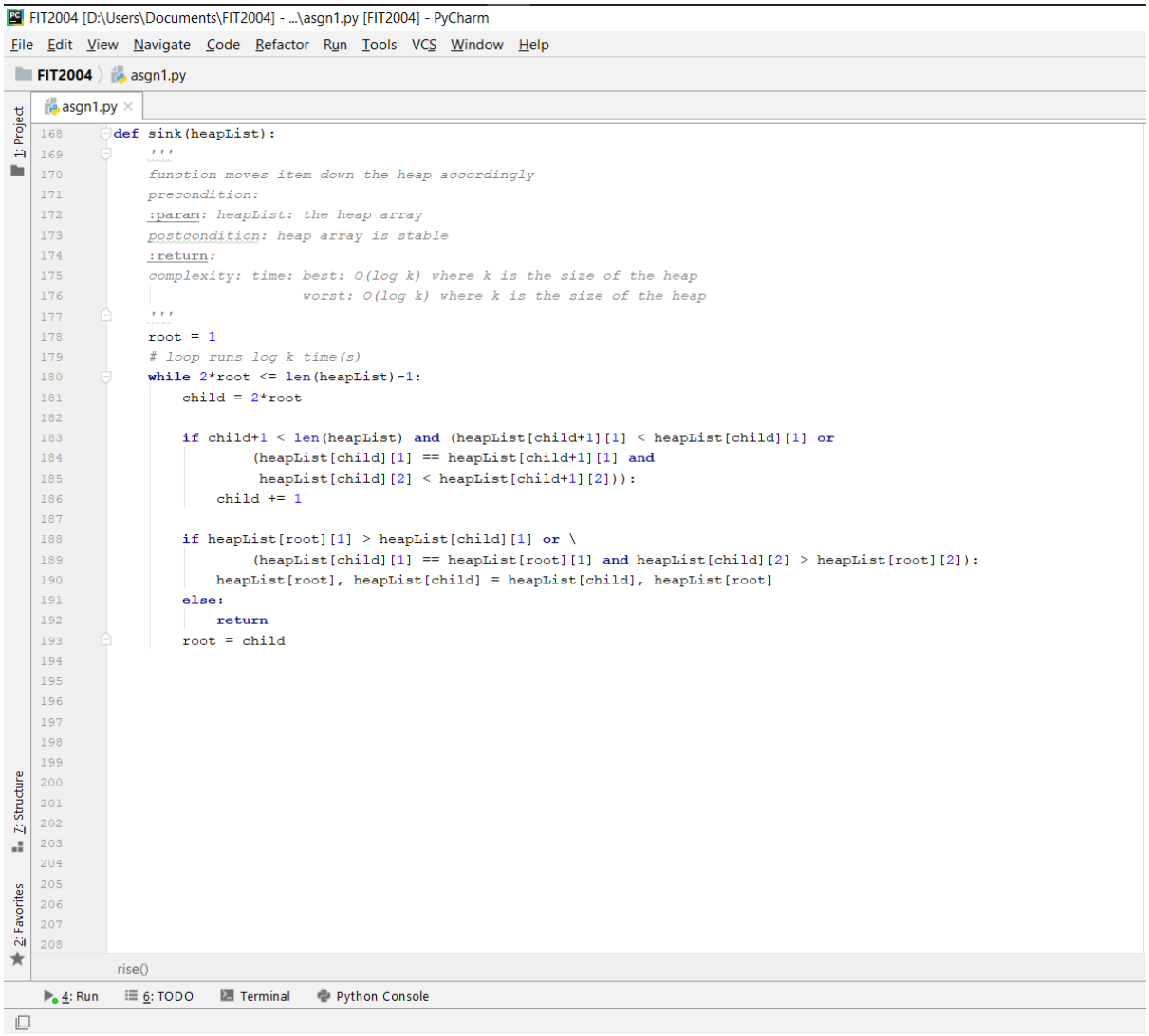
TASK 4



Task 4 requires a function *kTopWords* which will take a value k and an alphabetically sorted list of words with their frequencies and return a list that contains the k number of top-most frequent words. However, if the count of two words are the same, the word comes earlier in the sorted list will have higher priority over another. The algorithm to create a min heap and sort the heap is universal so, we would skip further elaboration but the implementation to stabilise the heap before and after sorting here, is done with the concept of priority queue. Position of the words in the sorted list is appended to it whenever the word at the root of the heap is replaced. This position value is what determines the priority of words to let remain/be replaced in the heap. Also, creating the min heap is done from the end of the list and sorting the heap is done from the bottom of the heap. This is to simplify replacement of words of higher priority by words of higher occurrences and the replacement of words of lower priority by those of higher priority when their occurrences are the same.

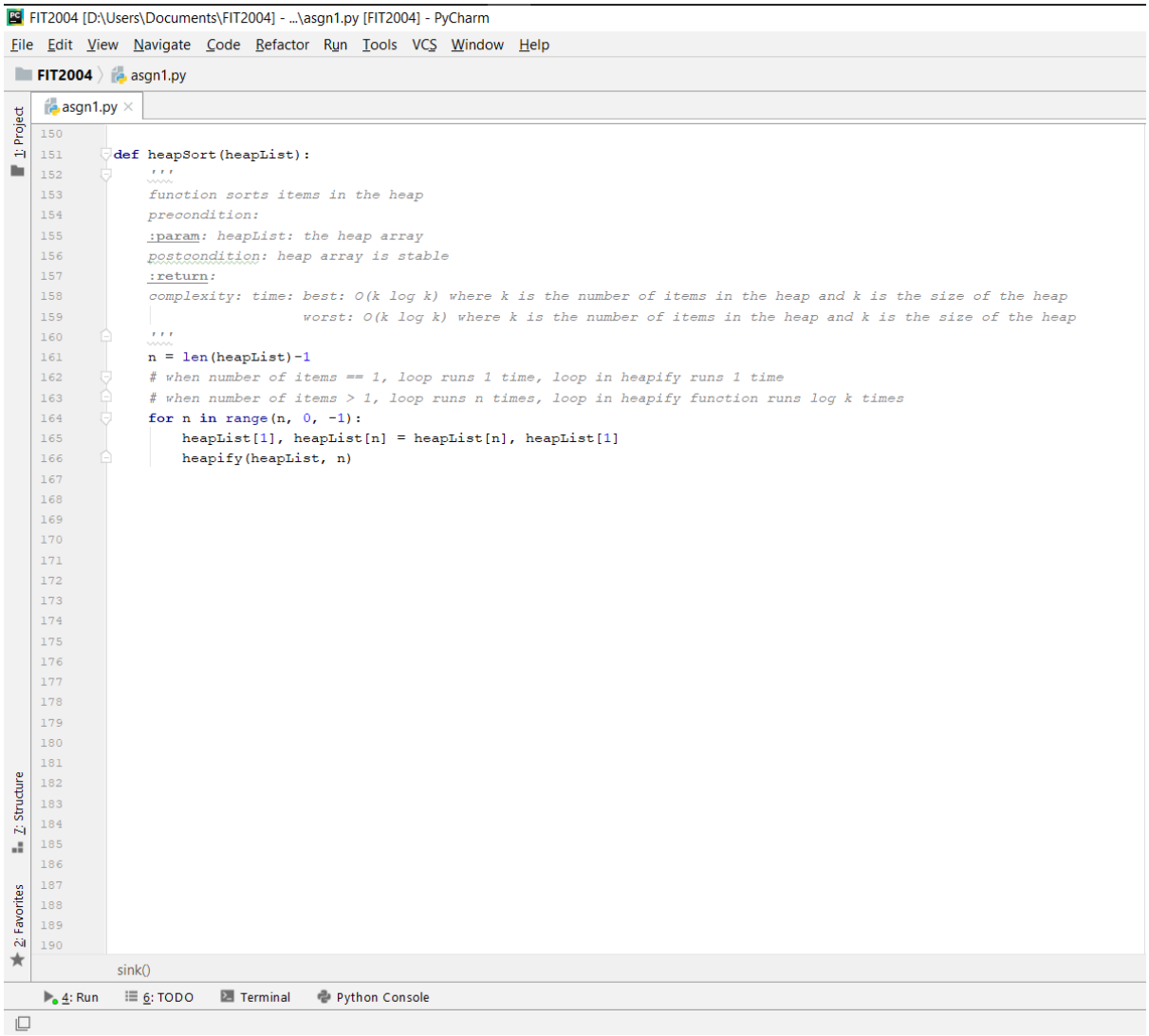


The first *for* loops through the k-1 words in the sorted list to append the k-1 items into the heap. The function *rise* is then called to ensure the heap is a min heap. In the function *rise*, the *while* loop compares item in the heap at a certain position(left/right child) to another item at position//2 (parent node)(integer division). The time complexity k log k so far is the total after both the loop and the *rise* function.

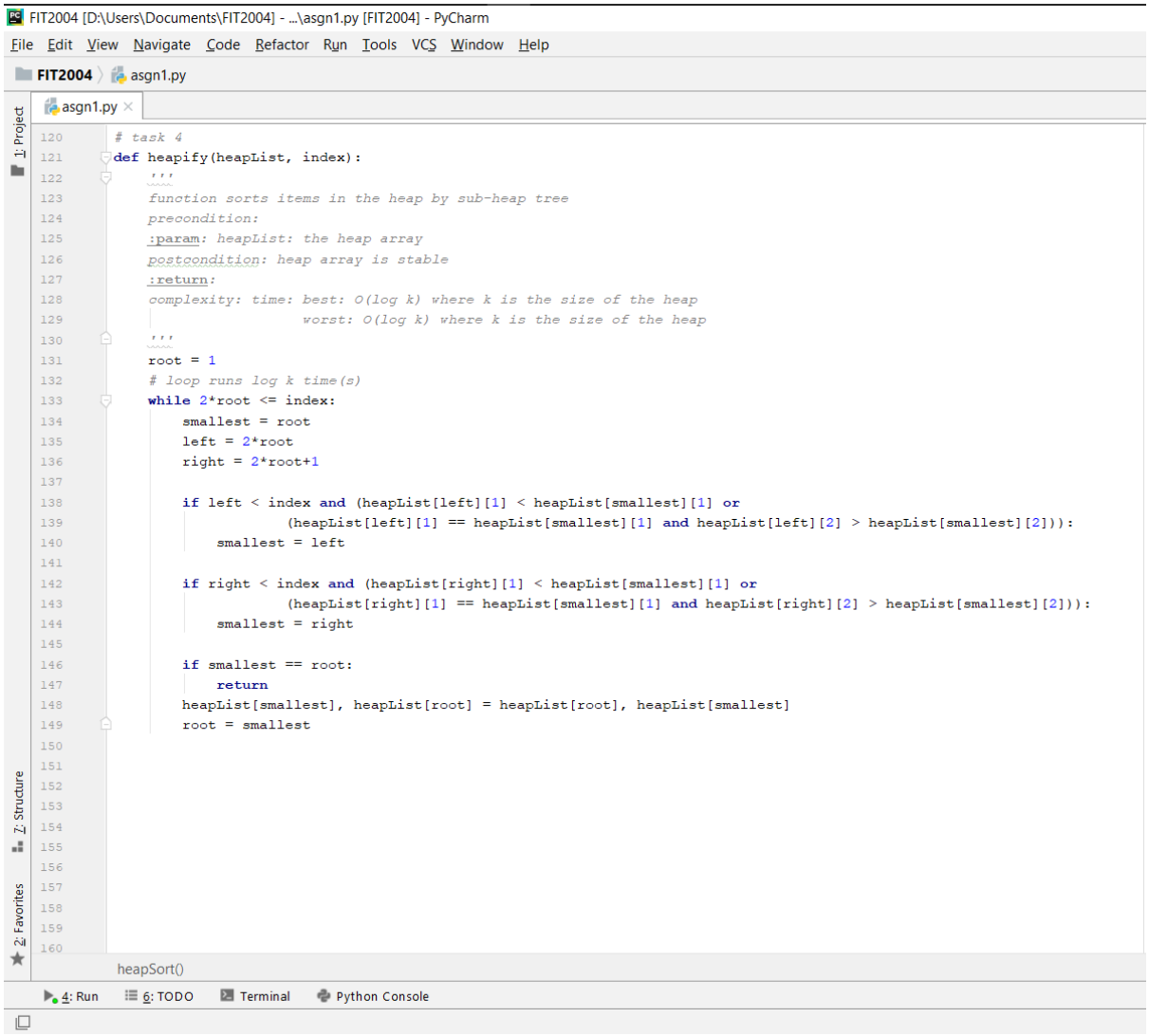


The second *for* loops through the remaining words in the sorted list. So, consider n to be the total number of words then, the loop runs n-k times. Then, the *if-statement* compares the occurrence of the word to the occurrence of the word at the root of the heap. This comparison is constant as both is integer. The function *sink* is then called to ensure the heap is a min heap. In the function sink, the *while* loop compares item in the heap at a certain position(parent node) to another item at position*2 and/or position*2+1(left and right child respectively). The time complexity k log k + (n-k) log k so far is the total after both previous loop and this loop.

Adding to the implementation, there are extra conditions added to the function compared to a normal *sink* function. The choice of child when both left and right occurrences of words is equal depends on the position of the words in the sorted list. The choice of moving the root item down the heap by swapping the child is also determined the same way. This is to ensure the lower prioritised words are first replaced.



That said, the heap now has all the necessary words and their frequency. It is pass as reference to the function *heapSort*. In the function *heapSort*, the *for* loops through the words in the heap, again, denoted as k and pass it as reference to the function *heapify* to swap items around until it is sorted. The k mentioned is the same as the k in previous function(s).



In the function *heapify*, the *while* loop compares item in the heap at a certain position(parent node) to another item at position*2 and/or position*2+1(left and right child respectively). The time complexity k log k + (n-k) log k + k log k = n log k + k log k so far is the total.

Adding to the implementation, there are extra conditions added to the function compared to a normal *heapify* function. The choice of child when both left and right occurrences of words is equal depends on the position of the words in the sorted list. The choice of swapping the root item with the child item is also determined the same way. This is to ensure the higher prioritised words are placed in the front of the lower prioritised words if their occurrences are equal.

Once the heap is sorted, the third for loops through the items in the heap, k, to remove unnecessary items/details before returning. This uses the in-built function *pop()* which we can assume it as an operation that cost constant time complexity represented in big O notation as O(1) as the item is in the last position; no list traversal is needed. Therefore, the total time complexity is n log k + k log k + k = (n+k) log k + k. We know that k <= n so, when k = n, 2n log k > n, hence, the worst time complexity is O(n log k).

The list (heap) created to return is affected by the k value of the user input. Appending to the list will cost k so total space complexity is k. However, since strings are list itself, it cost m

when appending to the list where m (maximum number of characters in a word) equals to the maximum size of list inside the list/heap. Hence, O(nm) is the worst space complexity.