ID: 28390121

Name: Priscilla Tham Ai Ching

FIT2004 Assignment 2: Analysis

TASK 1

```
def messageFind(self, filename='encrypted_1.txt'):
    '''
    function will find out the initial deciphered message (longest subsequence of common alphabets) from
two
    encrypted text
    precondition: the file is not empty and each text is in each line
    :param: filename: the name of the file which contains the two encrypted text
    postcondition: the file is not modified
    :return:
    complexity: time: best and worst: O(nm) where n is the size of the first text and m is the size of the
second text
                    or vice versa
                space: O(nm)
    error handling: return current content of the message class variable if file does not exist
    '''
    try:
        with open(filename, 'r', encoding='utf-8-sig') as file:
            content = file.read()

        content_len = len(content)
        # when number of characters == 1, for loop runs 1 time
        # when number of characters > 1, for loop runs n times
        for i in range(content_len):
            if content[i] == '\n':
                break
            self.fstSequence += content[i]

        # when number of characters == 1, for loop runs 1 time
        # when number of characters > 1, for loop runs m times
        for j in range(i+1, content_len):
            if content[j] == '\n':
                break
            self.sndSequence += content[j]

        fstSeq_len = len(self.fstSequence) + 1
        sndSeq_len = len(self.sndSequence) + 1
        dp = self.createDP(sndSeq_len, fstSeq_len)
        # loop runs n times
        for k in range(1, fstSeq_len):
            # loop runs m times
            for m in range(1, sndSeq_len):
                if self.fstSequence[k-1] != self.sndSequence[m-1]:
                    dp[k][m] = max(dp[k-1][m], dp[k][m-1])
                else:
                    dp[k][m] = 1 + dp[k-1][m-1]

        # loop depends on whether k (n) and m (m) is greater
        while k > 0 and m > 0:
            if self.fstSequence[k-1] == self.sndSequence[m-1]:
                self.message += self.fstSequence[k-1]
                k -= 1
                m -= 1
            elif dp[k-1][m] >= dp[k][m-1]:
                k -= 1
            else:
                m -= 1
        self.message = self.message[::-1]
        return
    except FileNotFoundError:
        return
```

Task 1 requires a function *messageFind* which will take a filename of the input file that contains the two encrypted texts. This function implements dynamic programming to find out the initial deciphered message from the two texts. 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 usage of in-built function *len()* throughout the function can be assumed 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 to newline, thus stop once met indicating the end of the first text. This comparison is constant as both the string and newline are always of length = 1. The second for loops does the same thing to get the second text.

```
def createDP(self, column_size, row_size, value = 0):
    '''
    function creates a dynamic table array of size row_size * column_size to implement dynamic programming
    precondition:
    :param: column_size: number of columns
            row_size: number of rows
            value: initialisation value; set to default 0
    postcondition:
    :return: dynamic table array created
    complexity: time: best and worst: O(nm) where n is the number of rows and m is the number of columns
                    or vice versa
                space: O(nm)
    error handling:
    '''
    return [[value for _ in range(column_size+1)] for _ in range(row_size+1)]
```

Next, a dynamic table array is created using the function *createDP*. The table arrays are initialised with the value 0 by looping through the rows and columns. So, the total time complexity so far is O(p) + O(nm) where n is the number of rows and m is the number of columns or vice versa.

The process to fill up the dynamic table array starts from the third for loop with another within. The outer loops through the rows where the number of rows is equal to the size of the first text, n whereas the inner loops through the columns where the number of columns is equal to the size of the second text, m. The total time complexity so far is O(p) + O(nm) + O(nm).

The algorithm is such that if the current string from the first text (row) is equal to the current string from the second text (column), then the length of the subsequence increases by one. The length of the subsequence found so far is stored in the table array located diagonally to the current cross point between the two strings. Otherwise, we will choose the longest common subsequence found so far thus the need to compare the table array located above (excluding the second string of the current row) and left (excluding the first string of the current column) of the current cross point between the two strings.

Now, the while loop backtracks from the final position we are in the dynamic table array – which tells us the length of the longest common subsequence found – to extract the strings used that forms the subsequence. Since the loop only runs when both k (row) and m (column) is valid, the worst time complexity depends on the variable which runs the longest. We get either O(p) + 2O(nm) + O(n) or O(p) + O(nm) + O(nm) + O(m).

The algorithm is such that if the current string in the first text is equal to the current string in the second text, the string is included in the subsequence and move to the top left where the string(s) has yet to be included. Otherwise, move to the longest common subsequence found either above (excluding the second string of the current row) or left (excluding the first string of the current column) of the current cross point.

Therefore, the total time complexity is either O(p) + 2O(nm) + O(n) or O(p) + 2O(nm) + O(m). We know that O(nm) > O(p) > O(n), hence, O(nm) is the worst time complexity. The two encrypted texts take up space of size n and m. Plus, initialising the dynamic table array cost n because there are n rows and m because there are m columns or vice versa. The space taken is the size of the dynamic table array nm (because nm > m > n or nm > n > m), thus, the worst space complexity is O(nm).

TASK 2

```
def wordBreak(self, filename='dictionary_1.txt'):
    '''
    function will break the deciphered message stored in the message class variable into multiple words
using the
    vocabulary from a list of words
    precondition: there must an initial deciphered message stored in the message class variable
    :param: filename: the file which contains the vocabulary
    postcondition: the file is not modified
    :return:
    complexity: time: best: O(m) where m is the maximum number of characters in a word
                      worst: O(m*km) where k is the size of the initial deciphered message and m is the
maximum
                      number of characters in a word
                space: O(km+nm)
    error handling: return current content of the message class variable if file does not exist
    '''
    try:
        with open(filename, 'r', encoding='utf-8-sig') as dictFile:
            dictWords = dictFile.read()

        dictWords = dictWords.split()
        dict_len = len(dictWords)
        if dict_len != 0:
            max = self.findMax(dictWords, dict_len)

            msg_len = len(self.message)
            dp = self.createDP(msg_len, max, -1)

            if msg_len < max:
                max = msg_len
            j = 0
            start = 0
            end = max
            # loop runs m times
            # best case when there is not matching words
            while max > 0:
                if self.message[start:end] != dictWords[j]:
                    j += 1
                    if j == dict_len:
                        start += 1
                        end += 1
                        j = 0
                else:
                    # loop runs m times
                    for k in range(start + 1, end + 1):
                        if dp[max][end] >= 0:
                            break
                        if dp[max][k] < 0:
                            dp[max][k] = j
                        elif dp[max][k] >= j:
                            dp[max][end-k] = -1
                            dp[max][k] = j
                        else:
                            break
                    start += 1
                    end += 1
                    # start = end
                    # end += max
                    j = 0
                if end > msg_len:
                    dp[max - 1] = dp[max][:]
                    max -= 1
                    start = 0
                    end = max
            # loop runs km times
            for m in range(msg_len + 1):
                if dp[1][m] > -1:
                    self.message = self.retrieveMsg(dp, dictWords, msg_len)
                    break
        return
    except FileNotFoundError:
        return
```

Task 2 requires a function *wordBreak* which will take a filename of the second input file that contains the vocabulary in each line. This function also implements dynamic programming to find out the words in the initial deciphered message. 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 usage of in-built function *len()* throughout the function can be assumed as an operation that cost constant time complexity represented in big O notation as O(1).

This vocabulary is 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. So, the total time complexity so far is O(n).

```
def findMax(self, wordList, length):
    '''
    function finds the maximum length of characters among words in a list
    precondition: list is not empty
    :param: wordList: list of words
            length: size of the list
    postcondition:
    :return: maximum length of characters
    complexity: time: best and worst: O(n) where n is the number of words in the list
    error handling:
    '''
    max = len(wordList[0])
    # loop runs n times
    for i in range(1, length):
        n = len(wordList[i])
        if n > max:
            max = n
    return max
```

Then, the function *findMax* loops through the list of words (vocabulary) to find the maximum number of characters in a word. So, consider n to be the number of words then, the function looks for it n times. 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). So, the total time complexity so far is O(n) + O(n).

Next, a dynamic table array is created using the function *createDP*. The table arrays are initialised with the value -1 by looping through the rows and columns. So, the total time complexity so far is 2O(n) + O(nm) where n is the number of rows and m is the number of columns or vice versa.

The process to fill up the dynamic table array starts from the while loop with for loop within. The outer loops through the rows where the number of rows is equal to the maximum number of characters, m whereas the inner loops through the columns according to the number of matching characters between the initial deciphered message and the word in the list of words, km. The total time complexity so far is 2O(n) + O(nm) + O(m*km).

The algorithm is such that it will start from the last row of the dynamic table array as we prioritise the longer matching words. It takes a subsequence of size row number and compare with words in the list. Once matches, the dynamic table array is stored with the position of the word in the list from the starting character to the ending character. We will then continue from the character after the ending character, again with the subsequence of size row number. Should there be no matches and the loop has not run until the end of the initial deciphered message yet, we continue with the subsequence from the next character, otherwise, we continue with the subsequence with a size smaller (row before the current row). Caution taken when considering subsequence of smaller sizes if and only if there is no longer subsequence containing a subset of the current subsequence.

```
def retrieveMsg(self, dp_table, wordList, length):
    '''
    function extracts the stored index in the dynamic table array to fetch matching word(s) from a list,
from
    initial deciphered message
    precondition: there are matching words and its index is stored in the dynamic table array
    :param: dp_table: dynamic table array to extract the indexes from
            wordList: list of words
            length: length of the initial deciphered message
    postcondition:
    :return: extracted message
    complexity: time: best and worst: O(n) where n is the number of columns
                space: O(p) where p is the total number of characters in the message
    error handling:
    '''
    msg = ''
    prediction = ''
    p = 1
    # loop runs n times
    while p < length + 1:
        if dp_table[1][p] > -1:
            if prediction != '':
                msg += prediction + ' ' + wordList[dp_table[1][p]]
                prediction = ''
            else:
                msg += wordList[dp_table[1][p]]
            p += len(wordList[dp_table[1][p]])
            if p < length:
                msg += ' '
        else:
            prediction += self.message[p - 1]
            p += 1
    if prediction != '':
        msg += prediction + ' '
    return msg
```

Finally, the second row will now contain the position of the words in the list and the words are then fetched using the function *retrieveMsg*. However, this is only done if there are matching words and it is a loop itself looping through the column. So, consider km to be the size of the initial deciphered message, the loop runs km times. If there are no matching words, the loop will also run km times.

Therefore, the total time complexity is 2O(n) + O(nm) + O(m*km) + O(km). We know that O(m*km) > O(nm) > O(km) > O(n), hence, O(m*km) is the worst time complexity which satisfy the assignment requirement. Splitting the vocabulary in the file will create a list of words of size n where n is the number of words. Plus, initialising the dynamic table array cost m because there are m rows and km because there are km columns or vice versa. Forming the message will takes up space of km where k is the size of the input string, but space included. The space taken is the size of the dynamic table array nm and message km (because nm + km > > n), thus, the worst space complexity is O(km + nm).