ID: 28390121

Name: Priscilla Tham Ai Ching

FIT2004 Assignment 3: Analysis

```
class TrieNode:

    def __init__(self, size=62):
        self.array = []
        # time and space complexity is constant
        self.child = [None]*size


class Trie:

    def __init__(self, size=62):
        # always points to the root/head
        self.head = TrieNode(size)
        # points to the current position
        self.node = self.head

    def _get_index(self, char):
        '''
        function calculates the position of the node for the character
        precondition: size of trie node must be 52 in case of inserting alphabets
        :param: char: character to be inserted
        postcondition:
        :return: integer in the range of 0 to the size of trie node as the position of the node for the
character
        complexity: time: best and worst: O(p) where p = 1 is possible
                    space:
        error handling:
        '''
        try:
            return int(char)
        # should it gives an error, the character is of type alphabets, therefore calculation is done by
using
        # ASCII value
        # an addition of 26 (letters) if the character is lowercase taking into consideration of uppercase
letters
        except ValueError:
            if char.islower():
                return ord(char) - ord('a') + 26
            if char.isupper():
                return ord(char) - ord('A')

    def append(self, data):
        '''
        function appends the item to the array of the node
        precondition:
        :param: data: item to be appended
        postcondition:
        :return:
        complexity: time: best and worst: O(p) where p = 1 is possible
                    space:
        error handling:
        '''
        node = self.node
        node.array += [data]

    def insert(self, char):
        '''
        function 'inserts' the item to the child node of the parent node
        precondition:
        :param: char: the item to be inserted
        postcondition: item is not stored inside the trie
        :return:
        complexity: time: best and worst: O(p) where p = 1 is possible
                    space: O(p) where p = 1 is possible as the creation of nodes is of constant size
        error handling:
        '''
        node = self.node
        index = self._get_index(char)
        if not node.child[index]:
            # runs in constant time
            node.child[index] = TrieNode()
        self.node = node.child[index]

    def search(self, key):
        '''
        function searches through tree for the item
```

```
        precondition:
        :param: key: item to be searched
        postcondition:
        :return: the array of the last node associated to the last character in the key
        complexity: time: best and worst: O(m) where m is the maximum number of characters of the key
                    space:
        error handling:
        '''
        node = self.head
        try:
            # when number of characters == 1, loop runs 1 time
            # when number of characters > 1, loop runs m times
            for i in range(len(key)):
                index = self._get_index(key[i])
                if not node.child[index]:
                    return []
                node = node.child[index]
            return node.array
        # should it gives an error, the item is of type integer
        except TypeError:
            index = self._get_index(key)
            if not node.child[index]:
                return []
            node = node.child[index]
        return node.array
```

Above are the classes used to create the trie.

TASK 1

```
def query(filename, id_prefix, last_name_prefix):
    '''
    function finds pairs of data containing the id prefix and last name prefix the user input
    precondition: the file is not empty and each data is in a new line;
                  data ids consists of integers only and data last names consists of alphabets only
    :param: filename: the name of the file containing the data
            id_prefix: positive integers
            last_name_prefix: letters
    postcondition: the file is not modified
    :return: the indices associated to the data containing the id prefix and last name prefix
    complexity: time: best: O(T) + O(l)/O(T) + O(k) where T is the number of characters in all ids
                      and all last names, k is the length of id_prefix and l is the length of
last_name_prefix
                      worst: O(T) + O(k + l + nk + nl) where nk is the number of records matching id_prefix
and
                      nl is the number of records matching last_name prefix
                space: O(T + nm) where T is the number of characters in all ids and all last names, n is
the
                number of data associated to all ids and all last names and m is the maximum number of
                characters (either id or last name) per data
    error handling: return empty list if file does not exist
    '''
    try:
        with open(filename, 'r', encoding='UTF-8') as file:
            context = file.read()

        n = len(context)
        if n == 0:
            return []

        data = ''
        sepCount = 0
        # 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(n):
            if context[i] != ' ' and (sepCount == 1 or sepCount == 3):
                data += context[i]
            if context[i] == ' ':
                sepCount += 1
                if sepCount == 2:
                    data += ' '
            if context[i] == '\n':
                data += '\n'
                sepCount = 0

        # further nodes/child nodes are only created during insertion
        idTrie = Trie(10)
        lnTrie = Trie(52)

        index = int(context[0])
        endFlag = 0
        # when number of characters == 1, loop runs 1 time
        # when number of characters > 1, loop runs T times
        # where T is the number of characters in all id and all last names
        # O(T)
        for j in range(len(data)):
            if data[j] == ' ':
                # points the node to the root/head for each new data
                idTrie.node = idTrie.head
                endFlag = 1
            elif data[j] == '\n':
                # points the node to the root/head for each new data
                lnTrie.node = lnTrie.head
                endFlag = 0
                index += 1
            elif endFlag != 1:
                # creates the node for the character
                # O(1)
                idTrie.insert(data[j])
                # appends the index associated to the data to the array in the node created
                # O(1)
                idTrie.append(index)
            else:
                # creates the node for the character
                # O(1)
                lnTrie.insert(data[j])
                # appends the index associated to the data to the array in the node created
                # O(1)
                lnTrie.append(index)

        # if both prefix is empty, return all indices of the database
```

```
        if len(id_prefix) == 0 and len(last_name_prefix) == 0:
            return list(range(0, index + 1))
        else:
            # search loop runs according to the length of id_prefix, k
            # O(k)
            matchedID = idTrie.search(id_prefix)
            # search loop runs according to the length of last_name_prefix, l
            # O(l)
            matchedLn = lnTrie.search(last_name_prefix)

            # if either is empty, return the matching indices for the non-empty prefix
            if len(id_prefix) == 0:
                return matchedLn
            elif len(last_name_prefix) == 0:
                return matchedID
            else:
                # further nodes/child nodes are only created during insertion
                pairTrie = Trie(index+1)

                pairs = []
                # when number of matching ids == 1, loop runs 1 time
                # when number of matching ids > 1, loop runs nk times
                # O(nk)
                # best case when there is no matching ids and matching last names
                for k in range(len(matchedID)):
                    # creates the node for the character
                    # O(1)
                    pairTrie.insert(matchedID[k])
                    # appends the index associated to the data to the array in the node created
                    # O(1)
                    pairTrie.append(matchedID[k])
                    # points the node to the root/head for each new data
                    pairTrie.node = pairTrie.head

                # when number of matching last names == 1, loop runs 1 time
                # when number of matching last names > 1, loop runs nl times
                # O(nl)
                # best case when there is no matching ids and matching last names
                for m in range(len(matchedLn)):
                    # search loop runs in constant time as items in matchedLn are all integers
                    # O(1)
                    pairs += pairTrie.search(matchedLn[m])
                return pairs
    except FileNotFoundError:
        return []
```

Task 1 requires a function *query* which will take a filename of the input file that contains the data, the id_prefix and last_name_prefix to search through the data for matching records. The data is in the format of

Record_index Identification_no First_name Last_name Phone_number Email_address

in each line. This function implements construction trie. 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. This loop is to extract ids and last names to ensure construction of trie is within O(T). It searches through the characters to append the characters of ids and last names into a whole new string by skipping through whitespaces and using them to identify the starting and ending position of the necessary characters.

Next, two trie; one for ids and the other for last names are created. Creation of trie runs in constant time as the size is constant. The second for loops through the characters in the new string which contains all the characters in all ids and all last names, T. Using whitespaces to

identify the starting and ending position of id characters and last name characters to insert into the correct trie. The total time complexity so far is O(T) excluding the time taken to read the file.

The algorithm is such that each character inserted into the trie leads to creation of nodes unless the character has already existed in the trie. The next character will then be inserted in the child node of the node. When a whitespace is met, the pointer will be pointed back to the root of the trie as it indicates the end of one record. These characters are not stored inside the nodes of the trie but rather, used to calculate for the position of node to store the index/indices associated to the record matching these characters in the node array.

The function of this function starts from the search of matching records of the input id prefix and last name prefix in the trie. This search loop runs according to the length of id_prefix, k and last_name_prefix, l. The total time complexity now is O(T) + O(k + l).

Another trie is created to insert the array retrieved from searching the trie for id_prefix. This array consists of all the index/indices of the matching records. Hence, the loop for the trie construction runs in O(nk) where nk is the number of records matching id_prefix. The final for loops through the array retrieved from searching the trie for last_name_prefix. This loop search through the trie for each index in the retrieved array from last_name_prefix. This cost O(nl) where nl is the number of records matching last_name_prefix. The total time complexity now is O(T) + O(k + l + nk + nl).

The algorithm is such that each node in the trie is storing one index from the array of indices matching with id_prefix. So, we loop through the trie for each index from the array of indices of matching last_name_prefix. Any index/indices matching will return an array of index indicating the pair of id and last name in the data of the index/indices contains the id_prefix and last_name_prefix respectively.

Therefore, the worst time complexity is O(T) + O(k + l + nk + nl). The new string created from extracting all ids and last names take the space of O(T) where T is all the characters in all ids and all last names. Each list stored in the node can have the maximum length of n where n is the number of records in the data. This list can be stored in m nodes where m is the maximum number of characters (either id or last name) in a record. The space taken by the trie is O(nm). Thus, the worst space complexity is O(T + NM).

TASK 2

```
def reverseSubstrings(filename):
    '''
    function finds substrings whose reverse appears in a string
    precondition: the file is not empty and there is only one string
    :param: filename: the name of the file which contains the string
    postcondition: the file is not modified
    :return: the pair of substrings and reverse of the substrings in the strings with its starting position
             in the string
    complexity: time: best and worst: O(K^2 + P) where K is the total number of characters in the input
string and P
                      is the total length of all substrings whose reverse appears in the string.
                space: O(K^2 + P)
    error handling: return empty list if file does not exist
    '''
    try:
        with open(filename, 'r', encoding='UTF-8') as file:
            context = file.read()

        n = len(context)
        if n == 0:
            return []

        # takes the space of O(M)
        # where M is the total length of all substrings
        substrings = ''
        # takes the space of O(N)
        # where N is the number of substring
        indexes = []
        i = 2
        j = 0
        # loop runs K-1 times
        # where K is the total number of characters in the input string
        # O(K^2)
        while i < n+1:
            if j != n:
                # string/list slicing is O(K)
                # where K is the maximum number of characters to slice
                item = context[j:j+i]
                if len(item) >= i:
                    substrings += item
                    indexes += [j]
                    substrings += ' '
                j += 1
            else:
                j = 0
                i += 1

        # further nodes/child nodes are only created during insertion
        trie = Trie()

        n = 0
        # takes the space of O(K)
        visited = [None]*len(context)

        # when number of characters == 1, loop runs 1 time
        # when number of characters > 1, loop runs M times
        # where M is the total length of all substrings
        # O(M)
        for k in range(len(substrings)):
            if substrings[k] == ' ':
                visited[indexes[n]] = indexes[n]
                n += 1
                # points the node to the root/head for each new data
                trie.node = trie.head
            else:
                # creates the node for the character
                # O(1)
                trie.insert(substrings[k])
                if len(trie.node.array) <= 0:
                    # appends the index/position of the character in the string to the array in the node
created
                    # O(1)
                    trie.append(indexes[n])
                elif visited[indexes[n]] is None:
                    # appends the index/position of the character in the string to the array in the node
created
                    # O(1)
                    trie.append(indexes[n])

        # when number of characters == 1, system look for whitespace(s) once
        # when number of characters > 1, system look for whitespace(s) N times
        substrings = substrings.split()

        # takes the space of O(KNp)
        # where K is the maximum number of characters in a substring and Np is the number of substrings
```

```
whose reverse
        # appears in the string
        output = []
        n = 0
        # when number of characters == 1, loop runs 1 time
        # when number of characters > 1, loop runs N times
        # where N is the number of substrings
        # O(KN)
        for m in range(len(substrings)):
            # search loop runs according to the length of substring, T
            # O(K) where T = K
            array = trie.search(substrings[m][::-1])
            if array:
                output += [[substrings[m], indexes[n]]]
            n += 1
        return output
    except FileNotFoundError:
        return []
```

Task 2 requires a function *reverseSubstring* which will take a filename of the input file that contains a string. This function implements construction suffix trie. 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 while loop runs K-1 times where K is the total number of characters in the input string. It forms a substring of length 2 onwards and concatenate them to form a whole new string. In the loop, an operation of string/list[starting position : ending position + 1] has taken place which loops according the length of output string/list. The time complexity so far is O(K^2) taking the maximum length to slice is equal to the total number of characters in the input string.

As mentioned above, the creation of trie runs in constant time as the size is constant. The second for loops through the characters of the newly created string to insert into the trie. The construction of the trie thus cost M. The time complexity so far is K^2 + M where M is the total length of all substrings.

The algorithm is such that each character inserted into the trie leads to creation of nodes unless the character has already existed in the trie. The next character will then be inserted in the child node of the node. When a whitespace is met, the pointer will be pointed back to the root of the trie as it indicates the end of one substring. These characters are not stored inside the nodes of the trie but rather, used to calculate for the position of node to store the starting position of the substring in the node array.

There is a list which keeps track of index/indices to be stored in the node array. This list is to ensure the same node does not store previously stored index/indices into the node array. When the character node is visited once, the visited[index/position of the character in the string] will update None to the index/position of the character in the string, as the node array would have stored the index/position of the character in the string. However, a new node may be created which hasn't store the index/position of the character in the string into the node array and yet visited[index/position of the character in the string] is not None. [first if] Hence empty node array (indicating creation of new node) as well as [second if] node arrays that hasn't store the current character's index/position in the string will be appended.

The concatenated substrings 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 substrings then, the function looks for it N times.

The third for loops through the list of substrings and searches the reverse of each substring in the trie. This search loop runs according to the length of substring, T. The time complexity so far is K^2 + M+ KN taking T to be the maximum number of characters in a substring which is equal to the the total number of characters in the input string, K.

The algorithm is such that should the reverse of the substring found in the trie, we know that the reverse of the substring will be met later in the loop. Hence, first output the substring itself knowing that since the list of substrings were used to create the trie, the substring exists in the trie; whatever exists in the trie must exist in the string as well. But instead of taking the array of index/indices retrieved from the search as it may contain more than one index of the starting position with the same letter, we use a list of indexes/indices. This was created earlier when we were extracting the substrings from the string. The increase as we move to another substring where list of indexes/indices at position counter would store the starting position of the current substring in the string.

Therefore, the total time complexity so far is K^2 + M + KN. We know that O(M) where M is the total length of all substrings and that O(K) where K is the maximum number of characters in a substring which is equal to the total number of characters in the input string and that O(N) is the number of substrings < O(K), O(M) < O(KN) <= O(K^2) or O(KN) < O(M) <= O(K^2), hence, O(K^2) is the worst time complexity. On the other hand, the space taken for the list of indexes/indices is O(N) where N is the number of substrings and the list of visited index is O(K). Each list stored in the node can have the maximum length of K-1 and this list can be stored in K nodes. The space taken by the trie is O(K^2). The output array at the end takes the space of KNp where Np is the number of substrings whose reverse appear in the input string. Thus, the worst space complexity is O(K^2).