Task 1: Querying a database

In this task, two Trie data structures were formed. One for the identification numbers of each individual and another for the last names. When the query(filename,id_prefix,last_name_prefix) was called, first searchPrefix was done for the id_prefix in the Trie data structure of ID numbers. The searchPrefix(root,word) method returned a list of indexes at which the prefix was present. This function was called twice for each prefix respectively. Once these lists were obtained, the intersecting elements between the two lists were returned.

(e.g. if id_list = [0,2,3] and last_name_lst = [0,1,3] then query returned [0,3])

Time Complexity

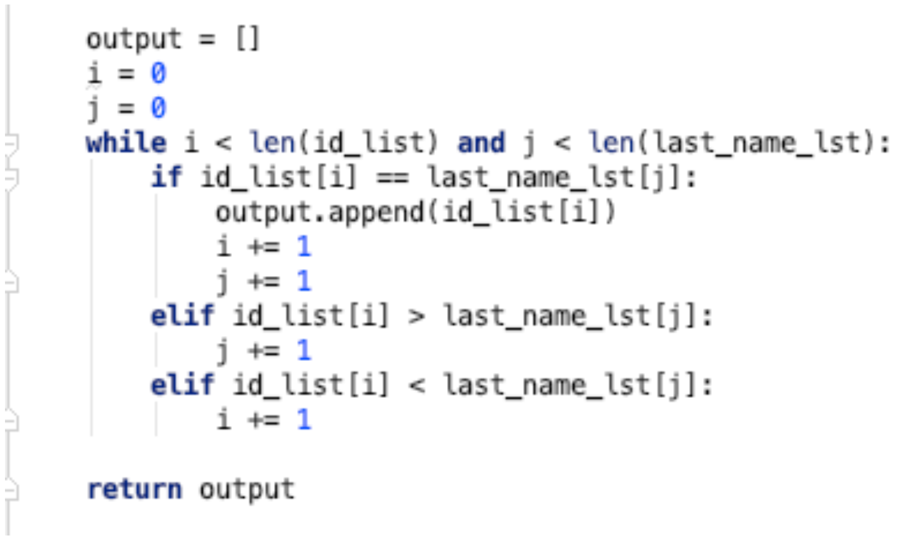
Building Trie

T1 = number of characters in all ID numbers
T2 = number of characters in all last names
To build ID Trie took O(T1) time as each character forms a node in the Trie To build last names Trie took O(T2) time for the same reason
Then total time is O(T1 + T2) = O(T)

Query
Let,
k = length of id_prefix
l = length of last_name_prefix
nk = number of records matching id_prefix
nl = number of records matching last_name_prefix

To search every character of the id_prefix within the Trie data structure takes O(k) time and O(l) time to search every character of last_name_prefix.

Since id_list and last_name_lst are already sorted, finding the intersecting elements can be done in linear time by comparing the items in id_list to items in last_name_list. If one value is smaller than the other then the smaller value is incremented until there is a match. Once there is a match that item is appended to a list of intersecting values. The code for this is shown below.



```
output = []
i = 0
j = 0
while i < len(id_list) and j < len(last_name_lst):
    if id_list[i] == last_name_lst[j]:
        output.append(id_list[i])
        i += 1
        j += 1
    elif id_list[i] > last_name_lst[j]:
        j += 1
    elif id_list[i] < last_name_lst[j]:
        i += 1

return output
```

Thus the time complexity for comparing two sorted lists is O(nk + nl) time.

Total time complexity for query is O(k + l + nk + nl)

Space Complexity

Total space complexity is O(T + NM), since in order to construct the Trie, it requires O(T) space. And to traverse through database text file, it requires O(NM) space. Thus total space required is O(T + NM)

Task 2: Reverse substring search

For Task 2, first a suffix Trie was built for the string, then the prefixes of the string were found and its characters were added on top of the Suffix Trie. If there was an intersection, that indicated that there would be a reverse substring for that character, and so it was saved in a list. After building the Suffix/Prefix Trie, the list of reverse substrings is searched using the FindPrefix method from Task 1 within the Trie in order to find the index at which the substring is present within the string. Finally the reverseSubstrings method returns the substring with the index at which it occurs.

Figure 1:

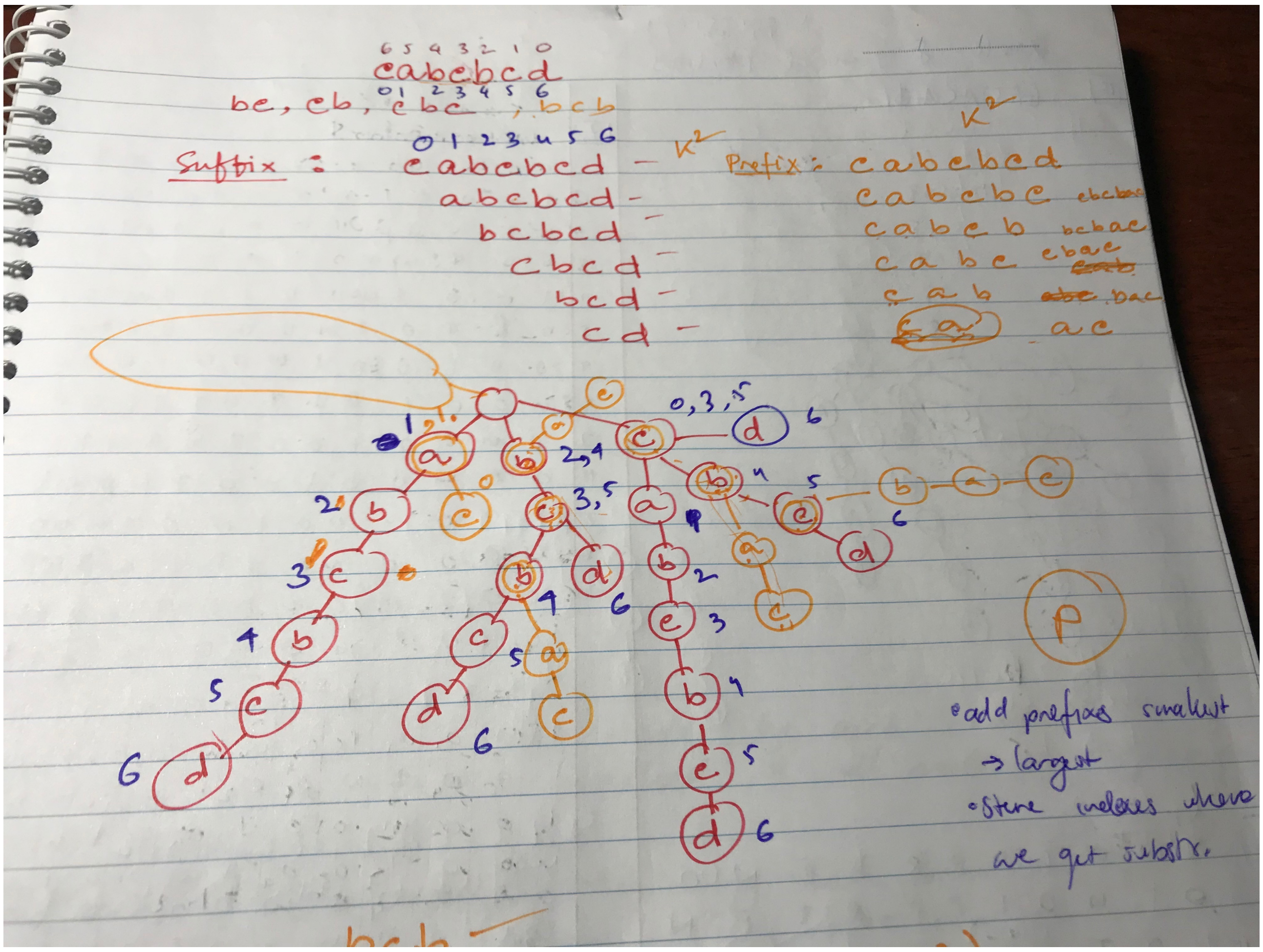


Figure 1 illustrates the algorithm used. The prefixes are added over the suffix tree. The nodes that are both orange and red indicate the reverse substrings. These are saved to a list and then searched in the findPrefix method.

Time Complexity

To Build Suffix Trie takes O(K^2) time
To add prefixes to Suffix Trie takes O(K^2) time Overall time O(K^2 + K^2) = O(K^2)

To traverse over intersecting nodes is O(P) time

Total time = O(K^2 + P)

Space Complexity

Space required for Suffix/Prefix Trie data structure would be O(K^2) due to construction of Suffix/ Prefix Trie.