Report

1. **Function Outline:** The first was broke into 2 main parts, construct a prefix trie and use the provided prefixes to filter the data. So after converting the data into numbers, loop through every character of the id in every record to insert all ids into the trie. Since there would be at most 10 identical numbers for each character in the id, so each node should have 10 spots plus one spot to indicate if the id ends or not and one spot for the index list. Each node in the trie would have an index list to store the indexes that have the prefix which contains all the characters from the beginning till this node. After generating the trie, another function would take the query prefix and look it up in the trie to find out whether it exists or not. If it exists, simply returns the index list that stores at the last position in the last node of this query prefix. Then use the filtered data as the original list to perform the constructing and finding actions again to find all the records that contain both id prefix and last name prefix.

Worst-case complexity: The worst time complexity is O(NM) for reading in the file, O(T) for constructing the prefix trie twice and O(k+l+n_k+n_l) for searching the prefix trie with the query prefix and getting all the matched records. The worst space complexity is O(NM) for reading the data, O(T+NM) for constructing the prefix trie and O(k+l+n_k+n_l) for searching and matching. This happens when all the characters in the raw data are identical so the trie cannot reuse the node and constructing would take the space of all the characters in the ids and last names. The constructing would take T space even after storing all the qualified indexes at each node is because, for a record, it would only take the space of its length to store its index at every node it has. So storing all the indexes would take T spaces and the constructing process would take O(2T) space which is O(T) space.

2. **Function Outline:** For the second task, the strategy is to build a suffix trie of the reversed original string first and then using all the chars in the original string as starting points to find all the possible results. So after converting the letters into numbers using ASCII, use a simple loop to reverse it and generate all the reverse suffixes. Then use each character in the original string as a start point to examine if the suffix trie has the same substring or not. If so, keep examine and when the length of the substring is greater than 2, append it with the index of the current character in the original string to the result list and go deeper. Keep going deeper until the original list reaches the end or the branch of the suffix trie cannot match anymore.

Worst-case complexity: The worst time complexity for this task is O(K^2+P) as K^2 stands for the time taken to go through each character in the string and compare them with the suffixes in the suffix trie. P stands for the time to generate the result list and remove the duplicate ones. The worst case time complexity happens when the string contains only one kind of letters and keep repeating this letter so the find function would take O(K^2+P) time to request and get the answer from the suffix trie. The worst case space complexity would be O(K^2+P) as K^2 is for the suffix trie and P for the result list. The worst case space complexity would happen when all the letters in the string are identical so when generating the suffix trie, the previous node cannot be reused and the suffix trie will use O(K^2) space.