FIT2004 – ASSIGNMENT 3 – ANALYSIS

James Schubach – 29743338

Task 1

My implementation for this task was to build 2 basic Trie data structures and fill them with keys for either ID or last names. From here I simple created a search function that would do a similar method to my insert function in which it would get the first letter of the key and convert it to a number and then place it in the trie accordingly. The search did this but instead would just follow the path until it either found a character that didn't match up or got to the end of the key. From here I recursively make my way down the trie until I reach the end, in which I return the given key, and then go down any other paths until all have been filled. Then it simply returns a list full of the records according to the queries.

Time Complexity: `O(k + l + nk + nl)This is due to the nature of the Trie Structure that allows us to search in O(n) time where n is the length of the key. In our case the key is k and l and then the number of records for both k and l respectively are recursively pulled in O(n) time. This allows the whole function to operate in the given time frame. Not accommodating for the creation of the Trie which is O(T) time, where T is the number of ID and last name characters.`

Space Complexity: `O(T + NM), the NM is from the results that have been returned, which will always be NM on the basis that the algorithm is correct. The O(T) represents the size of the trie which is filled with all the characters from ID and last names.`

Task 2

This function is a bit different to the first part, in this case we build a suffix tree with the reversed string. From here we generate all substrings of the normal string and then simply do a search with each substring as the key. The building of the Trie is a bit different as well. In this case we would simply start with the first character add it to the first spot in the array and then loop through the word till we got to the end, this would create one path. Then we would go to the next character, check if it can be attached to the previous character, if not we create a new spot for it. We repeat this for the rest of the word. From here all we have to do is do a simple search algorithm that finds the keys, exactly like part 1

Time Complexity: `O(K^2 + P)- The reason why this algorithm is K^2 + P comes from the fact that P (being the total length of all valid substrings) in which we search the structure from the smallest to largest at a given index and then once we reach a point where the substring doesn't fit we move to the next index. The K^2 part comes from the adding of the word into the suffix trie. We not only have to go over the word, we also have to make sure it cant fit into any other children, this brings the adding task up to k^2. As we have to go over a word twice basically.`

Space Complexity: `O(K^2 + P)`, This is because we only have in our trie all suffix's of the word and then we only store the valid substrings that fit into the reverse of the string. This allows the structure to make the space complexity