**The 3 following tasks are built on the same Trie and Node class.

***I used the recursive implementation got from lecture coding and modify it to save song id at each node (character).**

Trie: a tree that each node has a maximum number of M children (here M is 27, , for lower-case alphabet a-z and '$')

Data saved in a Node:

Node.data: the list of saved data

Node.child: the list for possible children of this node (here it has a size of 27, for lower-case alphabet a-z and '$')

Node.occur: the number of occurrences (number of songs it exists in)

Task 1: Radix sort Revenge (lookup(data_file,query_file))

Step 1 -> construct Trie

For each of the line in the input file, I will pre-process it and insert each word into my Trie along with the corresponding song id as data saved at each node. If the data is not seen in the list before, I will append it, otherwise nothing will be done. The program will add the node for each character into the Trie until the end (last character) and insert the data into the data list if it does not exist yet and compute the new number of occurrences at the ending node ('$' or Node.child[26]) before recurse the number of occurrences back to the parent. If the occur of child is larger than the parent, the occur of parent will be updated. This process will end when it reaches the root.

The recursive insertion method I used is optimised for task 1 and task 2, therefore it saves the occurrence of word (number of songs it exists in) at each node. The time complexity here is $O(C_I)$, C_I is the number of characters in data_file.

Step 2 -> search query

After that, I will search for each of the queries in the query_file, by using the "whole" mode for search to tell the program that it is finding for a complete match for the input word.

*I have added a parameter "mode" in to search function so that it can differentiate when it is finding a whole word for task 1 or finding a prefix for task 2.

For task 1, we are required to search for one whole word, therefore the search must end with a '$' to indicate that the word is found. In this task, the word is returned when a complete match is found otherwise "Not Found" will be return when the search cannot traverse deeper in to my Trie (query word does not exist in Trie). The complexity for searching is $O(C_Q)$, where C_Q is the number of characters in query_file.

At last, we write each result return by the search into a file. The time complexity for this is $O(C_P)$, where C_P is the number of characters in song_ids.txt (output file).

The resulting time complexity is $O(C_I + C_Q + C_P)$.

Task 2: Most Common Lyrics (most_common(data_file,query_file))

Step 1 -> construct Trie

Same implementation as Task 1 Step 1 (construct Trie). The time complexity here is $O(C_I)$, C_I is the number of characters in data_file.

Step 2 -> search query

I will search for each prefix given in the query_file and feed it to the search function for the word parameter. First, I will traverse down my Trie to determine if the input prefix exist in my Trie. The complexity for searching is $O(C_Q)$, where C_Q is the number of characters in query_file.

*the mode for search used in task 2 is "prefix" which "tells" the program that it is matching a prefix rather than one complete word as in task 1.

If the prefix exists (can be find in my Trie), there must have a word with that prefix in the Trie. Therefore, I will find the first child with the same occur value (Node.occur) with the last character of prefix and traverse deeper until I get to a '$' and return the word I traversed. The complexity for searching is $O(C_L)$, where C_L is the number of characters in the longest word.

At last, we write each result return by the search into a file. The time complexity for this is $O(C_M)$, where C_M is the number of characters in most_common_lyrics.txt (output file).

The resulting time complexity is $O(C_I + C_Q + C_M)$. C_L is dropped because it is less significant and does not contribute much to the time complexity.

Task 3: Palindrome finding (palindromic_substrings(S))

*For this task, a different insert and search function is used.

Step 1 -> construct Suffix Trie

First, I constructed a Suffix Trie, a Trie that holds the suffixes of a string. The way of inserting the individual strings into the Trie is somewhat similar to that of task 1 and task 2, the difference is the way of data that was saved at each node. At each node, the index of that specific character in the string is saved.

Step 2 -> Search for palindromic substring

For each iteration, I will input an ending point and the string into the search function for it to search for palindromic substrings in it. "i" indicate the maximum position in the string that I want to check in this iteration. Maximum number of iterations will be the length of the S, when the "I" is equal to length of S. Therefore, the time complexity here is O(N), where N is the length of S.

In the search function, there are 2 things that is stored, the result list where it saves the index of the character from S which found a match when traversing through the Trie; the compare list where it stores the data list pulled out at each matching node, the compare list

First, we will check starting from ending point(back of string) and towards the beginning of the string against the Trie(original string). If a character of the input word is found in the Trie, the result and compare list is updated. The maximum depth it has to traverse will be N (the original word). Therefore, the time complexity here is O(N), where N is the length of string S.

For example one of the suffix is "abab",

abab$ -- S[0:4] -> palindromic substring should be "bab" ,S[1:4]

result = [3, 2, 1] – the index of S that found a match when traverse through Trie

compare = [0, [3], [2, 6], [1, 3, 5]] → i = 1

After that, we compare the data saved in compare and result. If ith item in compare is not 0 it indicates that a match was found in that position of string S.

First, check the to see if an ending of palindromic substring can be found. We check and find the any matching item in compare that matches the first matching character (result[0]), If there is, the starting index of the palindromic substring will be the index that was found matching to result[0] in compare. For each iteration to search, the search will return all palindromic substring that has the same ending point.

The maximum depth it has to traverse will be length of S (to get the whole word, when the whole word is a palindrome). Therefore, the time complexity here is O(N), where N is the length of string S.

The resulting total time complexity is $O(N^2)$, where at each iteration, there will be at most N traversal, and there are at most N iteration.