Name: Khoo Teng Ian

1 - Sorting and Collating

First of all I need to read the file in order to get access to its elements, this is a complexity of O(TM) , where T is the total number of words and M is the length of the longest word, that is because we have to look at all the elements in the file and during this process I also check and update variables which hold the length of the longest word and the length of the longest number/ song_id for later use. I also append all the elements from the file into a list of lists. E.g. [[song_id_1,word_1],[song_id_1,word_2]…….]

There is two different implementation of Radix sort in my algorithm, one for song_ids (by value) and one for word (by lexicographic order). The need for two different implementations is because for numbers you know that a number with more digits should be sorted below one with less digits, you know the relative order of the song_id with relation to one another without the need to look through all the digits. However for words in lexicographic order, we don't know where the word will be sorted until we sort through all the letters of the word.

So with that in mind, to sort my song_ids, I first loop through all my elements and check the length of the number, I then append each element to another list of lists, but I append them based on the length. E.g. the ID 23 will be appended to the [1] list in my list of lists (23 has 2 digits and index starts with 0).

Now that every element is sorted by their correct lengths, I will perform radix sort separately on each 'bucket' and then combine them all together back in the end (starting from the shortest number)
The sorting itself is O(TN) where T is the total number of words and N the number of the longest digit, however if we assume our song_id is bounded by some constant we can view it as O(T).

Now that my list is sorted by song_id, I will sort by lexicographic order of the words, the reason we sorted by song_id first is that because my implemented radix sort is stable, relative order among elements is preserved and the same word with lower song_id value will appear higher up in the file.

For the words, I loop through all the words in the file starting M (length of longest word) -1, and I check the length of the word each iteration, if the word length is >= the current digit increment my count array I am looking at, then we include it in my radix sort.
I also at every loop keep track of how many elements I need to include in my Radix sort, this is because in usual radix sort your position array[0] is set to 1, but I set mine to T – num_involved . this is so when I am constructing my output, I also check for the length of the words, and if the length >= current_digit and only if the element fulfills this check do I shift it to its correct position .

The time complexity is O(TM), because I have to look through all the letters of all my words in order to properly sort them.

2 - Collating

For my collate function, I simply linearly look through all my elements, and if the word is different or the number is different appended to a list first. To make sure the same elements but different numbers are appended together I have a counter that increments only if the word is different.
E.g. [[‘a’,0,1,2,3],[‘b’,1]…..]

This has a complexity of O(TM) as I have to look through all my elements(T) and for each compare their words (comparison bounded by M).

3 – Lookup

Firstly I read my collated_songs and my query file and store them in two separate lists.
For my lookup function I used binary search to find if the query appears in any song or not, there are q, number of queries and for each query I must compare the strings to see if the mid is > or <query, and my mid is bounded by M, the length of the longest word. At each iteration of my binary search I cut by search space if half of my collated_songs file.
If I don’t find the word I add Not Found to my output file.
Hence my lookup is O(q * Mlog(U)).
But if my query does appear in the collated_songs file. I will need to loop through my collated_songs list and for each element of the found query add it to my output file.

Hence my overall lookup is O(q * Mlog(U) + P). Where P is the total number of IDs in the output