Task 1:

Process(filename):

We start with a "file" containing lyrics from many songs.
The function takes as input a string of a filename.
The function will sort the words of all the songs, and output these sorted words to another file.

A word is defined to be a sequence of lowercase English characters (a-z).
A word does not contain any kind of punctuation or other symbols.
Song lyrics are a sequence of words separated by spaces.
Each line of the input file will start with a non-negative integer, which is the song ID.
This number is followed by a colon, then the song lyrics on a single line process will write output to a file named "sorted_words.txt" (which it will create).
This file will consist of T lines, where T is the total number of words over all songs in the input file.

Each line will start with a word, then a colon, and then the song ID of the song that word belongs to.

The order of the lines depends on the words. If the word in line x is lexicographically less than the word in line y, then x will appear before y in sorted_words.txt. If two line have the same word, then the one with lower song ID will appear first.

Algorithm Walk-through:

The function process(filename) starts off by reading in the lines of 'file' into a list of 'tuples' of word-ID pairs.
This is done by using slice and the index of the colon(:) to split the song ID and corresponding Word.

We then build the list of tuples by appending a tuple consisting of word and corresponding ID into a list called "tuples'.
I then used the function 'maxDigitVal' which runs in complexity O(N). This function 'maxDigitVal' gets the length of the longest first element in the list of tuples by iterating over all the first elements (word) of the list of tuples and finding the longest word.

We then use the padding approach to bring all the words in the list of tuples to the length of the longest word.
This is done with time complexity O(TM) where T is the length of the longest word and M is the number of words we are iterating over.
This is done so that radix sort an be done on the list of tuples, to sort the list by the words alphabetically.

We do the padding by iterating over all the elements in the list of 'tuples', and if the word in each tuple (first element of each tuple)
is less in length compared to the longest word, that difference in length is added in zeros ('0' - padding element)to the end of that particular word by means of string concatenation.

We then have a list of 'tuples' which where each tuple contains a word (first element) that is of the length of the longest word.
We then sort the list of tuples using radix sort, to sort the tuples by the alphabetical order of the words.

How Radix Sort works:
In this variant of radix sort is least significant digit (LSD) radix sort.
In essence, LSD radix sort works by sorting an array of elements
one digit at a time, from the least significant to the most significant. Each digit is sorted in a stable manner in order to maintain the relative ordering of the previously sorted digits.

We sort the array one digit at a time, from least significant (rightmost column) to most significant (leftmost column)

For each digit: We sort using a stable sorting algorithm, this is done im Radix Pass

Radix Sort Works with complexity (best/worst case): O(KN) - where K is the length of the longest element in list and N is the length of the list

How Radix Pass Works:
Suppose we wish to sort an array that contains only elements in some fixed universe U .
The radix pass algorithm sorts the words based on the positions of digits column wise by taking their corresponding
ascii values into considerations (ascii value obtained using getDigit which is used in thr function).

Specially in the context of the function process(), let's assume that all of the words are in the range of the alphabet
Thus taking into consideration the range of ascii values that the words may occupy (ascii value of z = 122) we
provide appropriate base for radix_pass to function correctly.

We do a pass over the input to count the number of occurrences of each ascii value
In order to preserve the stability of the algorithm we then do a second pass over the input and place each element into its correct position.

Radix Pass works with complexity (best/ worst case): O(n + u) where n is the length of the longest string and u is the base

The function radix sorted returns a list of tuples where the words (first elements of tuples) are sorted alphabetically.
This 'sorted_list' now must be depadded whereby the padding element('0') must be removed from all the words to which it was added.
This is done with time complexity O(TM) where T is the length of thr longest word and M is the number of words we are iterating over.
We do the depadding by iterating over the tuples in 'sorted_list' and for each word (first element in tuple) we find the index of the 'padding element' which is '0'. Using this index (of padding element) we use slice to find the original word, and the original word added back into the tuple at the same position (padded word replaced with original word).

We then have to sort the identical words by song_ID. whereby if the words are same, the word with the smaller song_ID appears first.
This is done by running a function 'second_sort' which sorts the duplicate words by Song ID using radix_sort_integers (refer Code)
The second sort function runs with complexity O(KN) - where K is the length of ID's for each word and N is the length of thr list of tuples containing unique words and corresponding ID's

We open a file "sorted_words.txt"
Finally we iterate over each tuple in the list of sorted tuples and for each tuple the word (first element) and song ID (second element)is written into the file.
The file will contain only one word and song ID per line.

Task 2

Collated(filename):

The function takes as input a string of a "filename"
The function will collect the song IDs of each word, and output the unique words together with their song IDs to another file. collate will write output to a file named "collated_ids.txt" (which it will create).

This file will consist of U lines, where U is the number of unique words in the input file. Each line will start with a word, then a colon, and then all the song IDs of the songs which contain that word (separated by spaces). The song IDs in each line will appear in ascending order. The order of the lines depends on the words. If the word in line x is lexicographically less than the word in line y, then x will appear before y in collated_ids.txt.

Algorithm Walk-through:

The function starts off by reading in the lines of file into a list of "tuples" of word-ID pairs.
I then removed all the duplicate tuples from the list using function remove_duplicates, and the remaining list of unique tuples will contain occurrences whereby certain unique tuples may have the first element of the tuple - (word) in common but different song ID.
We then iterate over the list of unique tuples and for each tuple looked at we run a function check_duplicates_v2 which takes as input the first element of the tuple (word) and the list of unique tuples.

The function check_duplicates_v2 then returns all the ID's associated to that word. and appends the word and list of associated ID's into a list "final_list" .
Note at this point final_list contains a list of tuples where the first element of each tuple is the word and the second element of the tuples is a list of associated song ID's.

However final_list will contain duplicate tuples, because initially when building this final_list we took as input a list of uniqe tuples but there were instances where certain tuples contained the same word but differnt song ID's.
We solve this problem by running the function remove_duplicates a second time, giving as input the "final_list".

After running remove_duplicates we have a new list - (name: "final_sorted") of unique tuples, each tuple containing a word as its first element and
a list of associated song ID's as its second element.

We then open a new file collated_ids.txt and then iterate over the "final_sorted" list and for each tuple in the list, we iterate over the list of ID's, for each word (first element of tuple) we write to the file the word and the corresponding IDs.
This repeates for each ID in the list.
The file is then closed.

Task 3

Lookup(collated_ids, query)

which takes as input two filenames, and for each word in query_file determines the songs which contain that word.
The input to this task consists of two files. The first, collated_file, will be the same format as a file produced by the collate function from task 2. The second file, query_file contains words, each one on a separate line.
lookup will write output to a file named "song_ids.txt".
This file will contain the same number of lines as query_file. If the word on line i of query_file appears in at least one song, then line i of "song_ids.txt" will contain the song IDs (in ascending order, separated by spaces) that contain that word. If the word does not appear in any song, then the corresponding output line should be the string "Not found".

Algorithm Walk-through:

The function starts off by reading in the lines of "collated_ids" file into a list of 'tuples' of word-ID_List pairs. The function then proceeds to read the word in each line of "query" file into list 'query_array'.
I then iterate over the length of the query_array and for each word in query array we do a binary search (time complexity - log(N))
by giving as input the word and the list of 'tuples'.

This binary search returns the 'index' of that word in the list of 'tuples' and we fetch the list of ID's by accessing the second element of the tuples at the 'index'. These ID's are the appended to a list called 'ids'.
If the binary seach returns 'None' we append the string "Not Found". This string is then appended to the 'ids' list

We then open a new file "song_ids.txt". and then iterate over the length of 'ids' array and for each element in 'ids' we write into a new line in the file "song_ids.txt"
The file 'song_ids' is then closed.