Task 1)
lookup(data_file, query_file)

In order to do task 1, lookup(data_file, query_file) in the given time complexity, I have implemented a prefixTrie.

This function 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, data_file will contain in each line the song ID and words of the song, separated by a colon.
The 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 "data_file" 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'.
A instance of a PrefixTrie() object is created
We then iterate over the ‘tuples’ of word-id pairs and each word and ID is inserted into the prefix trie by calling PrefixTrie.insert()
This is done in O(n) where n is the length of the data file.

.insert(word, SongID):

*This function adds the 'word' which is a string, character by character into the PrefixTrie,
this is done by iterating over each character in the 'key'and if that character is not present in the PrefixTrie a new TrieNode is created and the char_appear_songID list is updated.*

*The 'CurrentRoot' is updated after each iteration, when a character is added to the trie.
This operation is done in O(1)*

As we iterate over the characters in the 'key', if that character (in that character combination) is already present in the Prefix Trie, the currentRoot is moved to that node in the Trie O(1) and the SongID is appended to the char_appear_songID list [done in O(1)] of the currentRoot (provided the songID it is not already present in the list).

At the completion of the addition of the 'word' to the prefix trie, when the leaf node is reached, the songID is appended to the songID list of the leaf Node and the boolean 'endOfWord' is set to True

This function executes with Best/Worst Case time-complexity: O(k) where k is the length of input into the prefix trie

We then open a new file "song_ids.txt"

We then iterate over the elements (‘words’) of the query_array and for each ‘word’ in query array we call a PrefixTrie.search(word) which returns the List of SongID’s in which the word appears in.
This list is then written into a new line in the file "song_ids.txt"
This is done in O(n) where n is the length of the query file.

.search(word):
This function checks for the presence of a word in a prefixTrie and returns the SongIDs of the songs in which the word can be found in.
The function iterates over the characters of the word and at each iteration checks for the presence of the character in the prefix trie in that order, if the character is found (in the correct order) in the prefix trie the currentRoot is moved to that Node. This is done in O(1)

When the end of the word is reached, we return the list of the SongIDs at that leaf node.

This function executes with Best/Worst Case time-complexity: *O(k) where k is the length of input into the prefix trie*

Task 2:

most_common(data_file, query_file):

The objective of task 2 is to determine which word is present in the most songs and begins with that prefix

most_common will write to a file “most_common_lyrics.txt”.
This file will contain the same number of lines as query_file.
If the string on line i of query_file is the prefix of a word in any song, then line i of “most_common_lyrics.txt” will contain the word which ...

a) Is present in the most songs
b) Has the string on line i of query_file as a prefix

If the string on line i of query_file is not the prefix of any word in any song, then the corresponding output line should be the string "*Not found*"*.*

Algorithm WalkThrough:

The function starts off by reading in the lines of "data_file" 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'.
A instance of a PrefixTrie() object is created
We then iterate over the ‘tuples’ of word-id pairs and each word and ID is inserted into the prefix trie by calling PrefixTrie.insert()
This is done in O(n) where n is the length of the data file.

Please refer to the .insert() function overview shown above

We then iterate over the elements (‘words’) of the query_array and for each ‘word’ in query array we call a PrefixTrie.most_common_search(prefix) which returns the word (that starts with the prefix) which appears in most songs. This list is then written into a new line in the file "song_ids.txt"

This is done in O(n) where n is the length of the query file.

.most_common_search(prefix):

This function finds and returns the word that occurs in most songs, which is derived from the prefix.

The function works by iterating over all the characters in the word and for each character we check if the character is present in the prefix trie (in that relative order). If any character of the prefix is not present, “Not Found” is returned.
If the character is found the currentRoot is updated to that relevant node. (Done in O(n).)

This entire operation to find the final node of the prefix in the prefix trie is done in O(n)

After currentRoot is set to the final character of the prefix in the Prefix Trie
The function recursive_find_most_common(currentRoot,string) is called.

best-case time complexity: O(m) where m is the length of the word that occurs in most songs derived from prefix
worst-case time complexity: O(m) where m is the length of the word that occurs in most songs derived from prefix

recursive_find_most_common(currentRoot,string):

This function iterates recursively to find the most common word given a currentRoot which is the end node of a prefix in the prefix Tree.
This function is called by an external function (most_common_search(prefix)) which determines the end node of the prefix in the prefix tree, then proceeds to call this fucntion (recursive_find_most_common) providing the end node of the prefix as 'currentNode'.

This function will then continuously recurse to find a word derived from that prefix which occurs in the most songs.
This is done by finding the child node with most unique occurrences in songs and moving the currentRoot to this node and recursively calling recursive_find_most_common(currentRoot,string):

the ‘string’ is updated at each level of recursion by appending the character with most song occurances at that level, to eventually create the word with most song occurrences. When the endOfWord is reached, the word, (derived from the prefix, which occurs in the most songs) is returned.

best-case time complexity: 0(m) where m is the length of the string

returned
worst-case time complexity: 0(m) where m is the length of the string returned

Task 3:

In task 3, the task is to find all palindromic subtrings of a string given as input.

To do this we iterate over each character in the string and at each character we run another loop to check if the right and left characters are the same and thus form a palindrome.

If they are the same we append the index of the palindromic substring into a list and check for the next-left and next-right character to see if they match and thus form a palindrome, if they match the indexes are appended into the list.
And the process repeats, if there is no match in successive left and right characters the inner loop is broken out of and the outer loop increments and we move to the next character.

This whole operation executes in O(N^2)