Story

Nathan and his friends have always enjoyed playing games.

Well, that might not be completely true. Nathan and his friends have always enjoyed winning games. The glory of victory, the thrill of achievement – these are the important things. These are what really matter.

For some people, the point of a game isn't to win, but rather, to have a good time. Good company, good fun, and a relaxed atmosphere. These people do not get invited to play games with Nathan and his friends.

Every so often, Nathan and his friends have a game night. These are cutthroat, vicious affairs, with every player doing their best to win (short of cheating). Professional coaches, obscure rules, loopholes, you name it.

Nathan has never won one of these game nights. But this is about to change, because this year, Nathan has you. With the power of algorithms, you will help him destroy his competition and finally claim the victory that he has craved for so long.

The first game night: The Song Game

Before the first game night of semester, Nathan finds out through devious and underhanded means, that the game is going to be the **Song Game**. The game is played as follows:

1. A word is chosen. This word is called the **target word**.
2. Each player must think of as many songs which contain the **target word** as possible.
3. Players then taking turns singing a line from a song which contains the **target word** and has not been sung by another player.
4. Players that cannot do so are out, with the last player remaining being the winner.

Nathan has asked for your help. You have agreed to write an algorithm for him which allows the user to quickly look up songs which contain specific words. A detailed breakdown of what you need to do is given in the tasks below.

Algorithm Descriptions (2 mark)

For each of the three tasks, you must also write a **brief** description of your algorithm. The total length should be no more than 800 words.
These three descriptions will be submitted in the pdf file `descriptions.pdf` mentioned in the "Submission Requirement" section. These descriptions should explain the steps your algorithm takes to solve the problem, and the complexity of each step. Please try to keep these descriptions

at a fairly high level, talk about your data structures and algorithms, not individual variables or lines of code.

1 Sorting and Collating (12 marks)

You start with a file containing lyrics from many songs. This is not in a useful format for quickly looking up individual words, so you will need to preprocess it first. You must write a function `process(filename)` which takes as input a string of a filename (the format of this file is specified in 1.1). The function will sort the words of all the songs, and output these sorted words to another file.

1.1 Input

For this task, we define a **word** 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. Since song lyrics can be quite long, each line of the file may also be quite long.

Example:

```
1234:an example song
222222:a second song
0:a example an example
```

1.2 Output

`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 **word**s 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.

Example:

```
a:0
a:222222
an:0
an:1234
example:0
example:0
example:1234
second:222222
```

```
song:1234
song:222222
```

1.3 Complexity

`process` must run in $O(TM)$ time, where

- T is the total number of words over all songs in the input file
- M is the length of the longest word

2 Collating (8 marks)

Now that You have the words in sorted order, the next task is to collect information about each **word**. When you look up a particular **word**, you want the **song ID**s of the songs which contain it to be easily accessible. To do this, you will write a function `collate(filename)` which takes as input a string of a file name (the format of this file is specified in 1.2). The function will collect the **song ID**s of each **word**, and output the unique **word**s together with their **song ID**s to another file.

2.1 Input

The input to this task is a file produced by the `process` function from task 1.

2.2 Output

`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 **word**s in the input file. Each line will start with a **word**, then a colon, and then all the **song ID**s of the songs which contain that **word** (separated by spaces). The **song ID**s 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`.

Example:

```
a:0 222222
an:0 1234
example:0 1234
second:222222
song:1234 222222
```

2.3 Complexity

`collate` must run in $O(TM)$ time, where

- T is the total number of words over all songs in the input file
- M is the length of the longest word

3 Lookup (8 marks)

You are finally ready to play the song game. You will be given a number of queries, each one consisting of a single **word**. For each query, you need to determine which songs contain that **word**. You will write a function `lookup(collated_file, query_file)` which takes as input two filenames, and for each word in `query_file` determines the songs which contain that **word**.

3.1 Input

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 **word**s, each one on a separate line. Note that these **word**s may not be in sorted order.

3.2 Output

`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 ID**s (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".

Example:
If the `query_file` was

```
a
example
the
```

`"song_ids.txt"` should contain

```
0 222222
0 1234
Not found
```

3.3 Complexity

`lookup` must run in $O(q \times Mlog(U) + P)$, where

- q is the number of **word**s in `query_file`
- M is the length of the longest **word** in any song
- U is the number of lines in `collated_file`
- P is the total number of IDs in the output

You may assume that the time complexity of reading the input file is small compared to the complexity of `lookup` as a whole. In other words, do not consider the time to read the input when determining the complexity of `lookup`

Warning

For all assignments in this unit, you may **not** use python dictionaries or sets. This is because the complexity requirements are all deterministic worst case requirements, and dictionaries/sets are based on hash tables, for which it is difficult to determine the deterministic worst case behaviour.

Please ensure that you carefully check the complexity of each inbuilt python function and data structure that you use, as many of them make the complexities of your algorithms worse. Common examples which cause students to lose marks are list slicing, inserting or deleting elements in a list in the middle or front of a list (linear time), using the `in` keyword to check for membership of an iterable (linear time), or building a string using repeated concatenation of characters.

These are just a few examples, so be careful. Remember, you are responsible for the complexity of every line of code you write!