Learning Outcomes

This assignment achieves the Learning Outcomes of:

- Analyse general problem-solving strategies and algorithmic paradigms, and apply them to solving new problems
- Prove correctness of programs, analyse their space and time complexities
- Compare and contrast various abstract data types and use them appropriately
- Develop and implement algorithms to solve computational problems.

In addition, you will develop the following employability skills:

- Text comprehension
- Designing test cases
- Ability to follow specifications precisely

Warning

For all assignments in this unit, you **cannot** use python dictionaries or sets. For all assignments in this unit, please ensure that you carefully check the complexity of each python function that you use. Common examples which cause students to lose marks are list slicing, inserting or deleting elements in a list, using the `in` keyword to check for membership of an iterable, 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!

Assignment timeline

In order to be successful in this assessment, the following steps are provided as a **suggestion**. This is an approach which will be useful to you both in future units, and in industry.

Planning

1. Read the assignment specification as soon as possible and write out a list of questions you have about it.
2. Clarify these questions; You can go to a consultation, talk to your tutor, discuss the tasks with friends or ask in the forums.
3. As soon as possible, start thinking about the problems in the assignment.
    - It is strongly recommended that you **do not** write code until you have a solid feeling for how the problem works and how you will solve it.
4. Writing down small examples and solving them by hand is an excellent tool for coming to a better understanding of the problem.

    - As you are doing this, you will also get a feel for the kinds of edge cases your code will have to deal with.

5. Write down a high level description of the algorithm you will use.

6. Determine the complexity of your algorithm idea, ensuring it meets the requirements.

7. Write your `description.pdf`.

    - You should be able to start working on this before you write your code.
    - If you cannot, perhaps your it is worth thinking a little more about how exactly your code will work.

Implementing

1. Think of test cases that you can use to check if your algorithm works.

    - Use the edge cases you found during the previous phase to inspire your test cases.
    - It is also a good idea to generate large random test cases.
    - Sharing test cases **is** allowed, as it is not helping solve the assignment.

2. Code up your algorithm, (remember decomposition and comments) and test it on the tests you have thought of.

3. Try to break your code. Think of what kinds of inputs you could be presented with which your code might not be able to handle.

    - Large inputs
    - Small inputs
    - Inputs with strange properties
    - What if everything is the same?
    - What if everything is different?
    - etc...

Before submission

- Make sure that the input/output format of your code matches the specification.

- Make sure your filenames match the specification.

- Make sure your functions are named correctly and take the correct inputs.

- Make sure you zip your files correctly

Story

After winning last week's games night (due entirely to your help), Nathan was allowed to choose the game for the following week. Still enraged after his loss in the Song Game, he has challenged the gaming group to a rematch.

This time, armed with more efficient data structures, you will smash the previous complexity bounds and allow Nathan to claim his rightful place as the Song Game Master.

On your advice, Nathan has also added a second round to the game. You mentioned that with the new data structures that you have learned, even more complex queries are now possible. As a result, players will also need to think of the most common word which starts with some given prefix.

This week's warm up game is going to be "find the palindrome", where players are given a long string of characters, and need to find all of the palindromes hidden inside the string as fast as possible. Whoever finds the most wins!

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 1200 words.
These three descriptions will be submitted in the pdf file `description.pdf` mentioned in the "Submission Requirement" section. These description 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 Radix sort Revenge

Given a file of song lyrics and a file of queries, (identical to the ones from assignment 1), you again need to find out which songs contain the words in the query file. To do this, you will write a function `lookup(data_file, query_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 `data_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.

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

Example:

`data_file`

```
1234:an example song excellent excellent excellent excellent lyrics
222222:a second example song
0:a example an example excellent excellent excellent extraordinary
```

`query_file`

```
example
an
the
```

1.2 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 IDs** (**in any order**) of songs which contain that word. If the `word` does not appear in any song, then the corresponding output line should be the string "Not found".

Example:

`"song_ids.txt"` for the examples given in 1.1.

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

Note that the song IDs in the first two lines could be in any order.

1.3 Complexity

`lookup` must run in $O(C_I + C_Q + C_P)$, where

- C_I is the number of characters in `data_file`
- C_Q is the number of characters in `query_file`
- C_P is the number of characters in `song_ids.txt`

2 Most common lyric

In this task, you will be given a prefix, and you need to determine which word is present in the most songs **and** begins with that prefix. To do this, you will write a function `most_common(data_file, query_file)`. These files have the same formats as `data_file` and `query_file` specified in section 1.1

2.1 Input

The formats of `data_file` and `query_file` are specified in section 1.1.

2.2 Output

`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".

If two or more words are tied for appearing in the most songs, any of them is acceptable.

Note: A word appearing multiple times in one song does not contribute to its frequency. We **only** care about how many songs a word appears in, **not** how many times is appears in `data_file`.

Example:
Given the example `data_file`

```
1234:an example song excellent excellent excellent excellent lyrics
222222:a second example song
0:a example an example excellent excellent excellent extraordinary
```

and the following `query_file`:

```
ext
ex
a
```

"`most_common_lyrics.txt`" should contain

```
extraordinary
example
an
```

Explanation:

- The only word in any song which begins with "ext" is extraordinary
- "example", "excellent", "extraordinary" all begin with "ex", but "example" appears in all three songs
- The last line could also be "a", since both "a" and "an" appear in two songs

2.3 Complexity

`most_common` must run in $O(C_I + C_Q + C_M)$, where

- C_I is the number of characters in `data_file`
- C_Q is the number of characters in `query_file`
- C_M is the number of characters in `most_common_lyrics.txt`

3 Palindrome finding

In this task, you are given a string `S[0..n-1]`, and you need to find all palindromic substrings of S, with length at least 2. A palindromic substring is some `S[i..j]` such that `S[j..i] = S[i..j]`. To do this, you will write a function `palindromic_substrings(S)`

3.1 Input

A string S, containing only lowercase a-z.

3.2 Output

A list of tuples, where each tuple represents the start and end index of a palindromic substring. The order of the tuples is not important.

Example:
Calling `palindromic_substrings("ababcbaxx")` would return
`[(0,2), (1,3), (3,5), (2, 6), (7,8)]`, or some permutation of that list.

- (0,2) - "aba"
- (1,3) - "bab"
- (3,5) - "bcb"
- (2,6) - "abcba"
- (7,8) - "xx"

3.3 Complexity

`palindromic_substrings` must run in $O(N^2)$ where N is the length of `S`.