FIT2004 S2/2022: Assignment 1

DEADLINE: Friday 19^{th} August 2022 16:30:00 AEST.

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 calendar days late will receive 0. The number of days late is rounded up, e.g. 5 hours late means 1 day late, 27 hours late is 2 days late. For special consideration, please visit this page: `https://forms.monash.edu/special-consideration` and fill out the appropriate form.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using the **Python programming language** (version should not be earlier than 3.5). This practical work will be marked on the time complexity, space complexity and functionality of your program, and your documentation.

Your program will be tested using automated test scripts. It is therefore critically important that you name your files and functions as specified in this document. If you do not, it will make your submission difficult to mark, and you will be penalised.

SUBMISSION REQUIREMENT: You will submit a single python file, `assignment1.py`.

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. In previous semesters, many students were detected by the plagiarism detector and almost all got zero mark for the assignment and, as a result, many failed the unit. Helping others to solve the assignment is NOT ACCEPTED. Please do not share your solutions partially or completely to others. If someone asks you for help, ask them to visit a consultation for help.

Learning Outcomes

This assignment achieves the Learning Outcomes of:

- 1) Analyse general problem solving strategies and algorithmic paradigms, and apply them to solving new problems;
- 2) Prove correctness of programs, analyse their space and time complexities;
- 3) Compare and contrast various abstract data types and use them appropriately;
- 4) 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.

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.

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 (if required).

Documentation

For this assignment (and all assignments in this unit) you are required to document and comment your code appropriately. Part of the marks of each question are for documentation. This documentation/commenting must consist of (but is not limited to):

- For each function, high-level description of that function. This should be a two or three sentence explanation of what this function does and the approach undertaken within the function.
- For each function, specify what the input to the function is, and what output the function produces or returns (if appropriate).
- For each function, the appropriate Big-O or Big-Θ time and space complexity of that function, in terms of the input size. Make sure you specify what the variables involved in your complexity refer to. Remember that the complexity of a function includes the complexity of any function calls it makes.
- Within functions, comments where appropriate. Generally speaking, you would comment complicated lines of code (which you should try to minimise) or a large block of code which performs a clear and distinct task (often blocks like this are good candidates to be their own functions!).

A suggested function documentation layout would be as follows:

```
def my_function(argv1, argv2):
    """
    High level description about the functiona and the approach you
    have undertaken.
    :Input:
        argv1:
        argv2:
    :Output, return or postcondition:
    :Time complexity:
    :Aux space complexity:
    """
    # Write your codes here.
```

1 The Perfect Matchups (10 marks)

You and your friends are playing this new mobile video game called Diabro Immoral Gacha (DIG). The game revolves around building a team of M characters to compete with other teams in a tournament. You aimed to be the very best, like no one ever was in the game.

Thus, you have done your research into past tournaments in preparation for the biggest tournament. You aim to find the best performing team composition against the other teams. You are however dealing with a large amount of tournament results. Unlike other competitors, you are able to process this data **efficiently using sorting algorithms (counting and radix)** you have learnt from FIT2004.

For this task, you are to write a Python function called `analyze(results, roster, score)` which will perform an analysis on the tournament results. The input, output and complexity for this function is to be discussed in their individual sections Section 1.1, Section 1.2 and Section 1.3 respectively.

1.1 Input

The past tournament data `results` is presented as a list of lists [1]. The inner list can be described as `[team1, team2, score]` where:

- `team1` and `team2` are uppercase strings.
  - Both `team1` and `team2` are of the same length, denoted by the positive integer M.
  - Both `team1` and `team2` are from the same character set, denoted by the positive integer `roster`. For example, `roster=5` indicates a character set of `{A, B, C, D, E}`. It is possible that not all characters from the roster set would necessarily appear in the team composition [2].
  - Teams can have multiple instances of the same character, for example, team `ABA` with two `A`s.
  - The order of characters in `team1` and `team2` do not matter. A team of `ABC` is regarded to be the same as a team of some permutation of `ABC`, such as `BAC` or `BCA`.
- `score` is an integer value in the range of 0 to 100 inclusive.
  - It denotes the score obtained by `team1` in a match against `team2`.
  - It also denotes the score of $(100 - \texttt{score})$ obtained by team `team2` in the same match against `team1`.

[1] Instead of a list of tuples, so that they are mutable.

[2] Often occurs when a character is undertuned or out of the meta.

A snapshot of the data is provided below, containing $N = 20$ matches between teams of $M = 3$ characters. In general:

- N is a large number.
- The matches are not listed in any specific order in `results`.
- It is possible to have multiple matches between the same teams, with the same `score`. For example, it is possible for `results` to contain matches `['ECA', 'CDE', 13]`, `['CEA', 'CDE', 20]`, `['CEA', 'CDE', 88]` and `['CDE', 'ECA', 68]` which involves the same team composition against each other but with different arrangements and scores.

```
>>> results
[['EAE', 'BCA', 85], ['EEE', 'BDB', 17], ['EAD', 'ECD', 21],
['ECA', 'CDE', 13], ['CDA', 'ABA', 76], ['BEA', 'CEC', 79],
['EAE', 'CED', 8], ['CBE', 'CEA', 68], ['CDA', 'CEA', 58],
['ACE', 'DEE', 24], ['DDC', 'DCA', 61], ['CDE', 'BDE', 67],
['DED', 'EDD', 83], ['ABC', 'CAB', 54], ['AAB', 'BDB', 15],
['BBE', 'EAD', 28], ['ACD', 'DCD', 50], ['DEB', 'CAA', 21],
['EBE', 'AAC', 24], ['EBD', 'BCD', 48]]
```

- The first match of `result` in the example is `['EAE', 'BCA', 85]`, which means that:
  - A team of 1 `A` character and 2 `E` characters (`AEE`)
    has a score of 85 against
    a team of 1 `A` character, 1 `B` character and 1 `C` character (`ABC`).
  - A team of 1 `A` character, 1 `B` character and 1 `C` character (`ABC`)
    has a score of 15 against
    a team of 1 `A` character and 2 `E` characters (`AEE`).
- The last match of `result` in the example is `['EBD', 'BCD', 48]`, which means that:
  - A team of 1 `B` character, 1 `D` character and 1 `E` character (`BDE`)
    has a score of 48 against
    a team of 1 `B` character, 1 `C` character and 1 `D` character (`BCD`).
  - A team of 1 `B` character, 1 `C` character and 1 `D` character (`BCD`)
    has a score of 52 against
    a team of 1 `B` character, 1 `D` character and 1 `E` character (`BDE`).

1.2 Output

The function `analyze(results, roster, score)` returns a list of findings denoted as `[top10matches, searchedmatches]` where:

- `top10matches` is a list of 10 matches with the highest score.
  - The returned matches are sorted firstly in descending order by their scores, followed by ascending lexicographical order for `team1` (where scores are equal), and finally by ascending lexicographical order for `team2` (where scores and `team1` are equal).
  - If there are matches between the same teams with the same score, only one of those matches would be included in the list. The list could however include matches between the same teams but with different scores.
  - If there are less than 10 matches; then the list would contain the maximum number of matches possible.
  - You can assume that there will always be at least one match to be returned in `top10matches`.
- `searchedmatches` is a list of matches with the same score as `score`.
  - `score` is an integer within the range of 0 to 100 inclusive.
  - The returned matches are sorted firstly in ascending lexicographical order for `team1`, and secondly in ascending lexicographical order for `team2` (where `team1` is the same).
  - If there are matches between the same teams, only one of the matches would be included.
  - If the score is not found, then return the matches with the closest score which is higher. If there are no matches with a higher score, then return an empty list.

Consider the example input below:

```
# a roster of 2 characters
roster = 2
# results with 20 matches
results =
[['AAB', 'AAB', 35], ['AAB', 'BBA', 49], ['BAB', 'BAB', 42],
['AAA', 'AAA', 38], ['BAB', 'BAB', 36], ['BAB', 'BAB', 36],
['ABA', 'BBA', 57], ['BBB', 'BBA', 32], ['BBA', 'BBB', 49],
['BBA', 'ABB', 55], ['AAB', 'AAA', 58], ['ABA', 'AAA', 46],
['ABA', 'ABB', 44], ['BBB', 'BAB', 32], ['AAA', 'AAB', 36],
['ABA', 'BBB', 48], ['BBB', 'ABA', 33], ['AAB', 'BBA', 30],
['ABB', 'BBB', 68], ['BAB', 'BBB', 52]]
```

Which returns the following output:

```
"""
Example 1
"""

# looking for a score of 64
score = 64

# running the function
>>> analyze(results, roster, score)
# the following is returned for [top10matches, searchedmatches]
[[['ABB', 'AAB', 70],
['ABB', 'BBB', 68],
['AAB', 'BBB', 67],
['AAB', 'AAB', 65],
['AAB', 'AAA', 64],
['ABB', 'ABB', 64],
['AAA', 'AAA', 62],
['AAB', 'AAA', 58],
['ABB', 'ABB', 58],
['AAB', 'ABB', 57]],
[['AAB', 'AAA', 64], ['ABB', 'ABB', 64]]]
```

`top10matches` as a list of 10 matches. The following observations can be made:

- The characters in the team compositions are sorted in lexicographical order.
- The highest score of 70 comes from the match `['AAB', 'BBA', 30]`. As discussed in earlier Section 1.1, this match would also have an equivalent of `['BBA', 'AAB', 70]`.
- For the score of 68, there are in fact 2 entries. The first entry is `['ABB', 'BBB', 68]` and the second entry being `['BBB', 'BBA', 32]` which is equivalent to `['BBA', 'BBB', 68]`. These 2 entries are however not unique, and thus only 1 would be presented in the output.
- The matches are sorted according to their score.
- For matches with the same score, they are sorted according to the lexicographical order of the teams.

`searchedmatches` is observed to be `['AAB', 'AAA', 64], ['ABB', 'ABB', 64]`.

- The team matchups are unique.
- The matches are sorted according to the lexicographical order of the teams.

```
"""
Example 2
"""

# looking for a score of 63
score = 63

# running the function
>>> analyze(results, roster, score)
# the following is returned for [top10matches, searchedmatches]
[[['ABB', 'AAB', 70],
['ABB', 'BBB', 68],
['AAB', 'BBB', 67],
['AAB', 'AAB', 65],
['AAB', 'AAA', 64],
['ABB', 'ABB', 64],
['AAA', 'AAA', 62],
['AAB', 'AAA', 58],
['ABB', 'ABB', 58],
['AAB', 'ABB', 57]],
[['AAB', 'AAA', 64], ['ABB', 'ABB', 64]]]
```

In Example 2, a search for `score=63` would return all the matches with a score of 64 for `searchedmatches`; due to the fact that there is no score of 63 and 64 is the closest score which is higher than 63.

```
"""
Example 3
"""

# looking for a score of 71
score = 71

# running the function
>>> analyze(results, roster, score)
# the following is returned for [top10matches, searchedmatches]
[[['ABB', 'AAB', 70],
['ABB', 'BBB', 68],
['AAB', 'BBB', 67],
['AAB', 'AAB', 65],
['AAB', 'AAA', 64],
['ABB', 'ABB', 64],
['AAA', 'AAA', 62],
['AAB', 'AAA', 58],
['ABB', 'ABB', 58],
['AAB', 'ABB', 57]],
[]]
```

If `score=71` instead, then `searchedmatches` would be an empty list because there are no matches that ended with a score of 71 or above.

```
"""
Example 4
"""

# looking for a score of 0
score = 0

# running the function
>>> analyze(results, roster, score)
# the following is returned for [top10matches, searchedmatches]
[[['ABB', 'AAB', 70],
['ABB', 'BBB', 68],
['AAB', 'BBB', 67],
['AAB', 'AAB', 65],
['AAB', 'AAA', 64],
['ABB', 'ABB', 64],
['AAA', 'AAA', 62],
['AAB', 'AAA', 58],
['ABB', 'ABB', 58],
['AAB', 'ABB', 57]],
[['AAB', 'ABB', 30]]]
```

Lastly in Example 4 above, a search for `score=0` would return matches with the score of 30 as there is no score of 0 and 30 is the closest score which is higher than 0.

1.3 Complexity

The function `analyze(results, roster, score)` must run in a time complexity of $O(NM)$ and a space complexity of $O(NM)$ worst case where:

- N is the number of matches within `results`.
- M is the number of characters within a team for each match.
- You can treat `roster` as a constant in your complexity analysis.
- You would need to account for the comparison cost for `string`. You can however treat the comparison cost for `integers` as $O(1)$.
- Make sure you account for the extreme and boundary cases during your complexity analysis, for example, the case where all matches within `result` have the same score.

Warning

For all assignments in this unit, you may **not** use python **dictionaries** or **sets**. This is because the complexity requirements for the assignment 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 in-built 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 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. Note that use of these functions/techniques is **not forbidden**, however you should exercise care when using them.

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