Lab Test 2 [25 marks]

Coding time: 1 hour and 30 mins

Instructions on how to download the resource files:

1. Before the test begins, you will be instructed to download the resource files from eLearn LMS.

General Instructions:

1. You will take the lab test on your personal laptop.
2. You are not allowed to communicate with anyone or accessing any website (except for downloading/uploading files from/to eLearn) during the test. **Anyone caught communicating with another person or accessing any website during the test WILL be considered as cheating and subjected to disciplinary actions.**
3. You may refer to any file on your laptop during the test.
4. Make sure your code can generate exactly the same output as we show in the sample runs. You may be penalized for missing spaces, missing punctuation marks, misspellings, etc. in the output.
5. Do not hardcode. We will use different test cases to test and grade your solutions.
6. Python script files that cannot be executed will NOT be marked and hence you will be awarded 0 marks. You may wish to comment out the parts in your code which cause execution errors.
7. Include your name as author in the comments of all your submitted source files. For example, include the following block of comments at the beginning of each source file you need to submit.

```
# Name    : BAI She Jing
# Email ID: shejing.bai.2022
```

Instructions on how to submit your solutions:

1. When the test ends, zip up all the files required for submission in a zip archive. The name of the zip archive should be your email ID. For example, if your email is `shejing.bai.2022@sis.smu.edu.sg`, you should name the archive as **`shejing.bai.2022.zip`**. You may be penalized for not following our instructions.
2. Once everybody has zipped up his/her files, your invigilator will instruct you to submit your solutions as a single zip file to eLearn under Assignments.

Question 1 (Difficulty Level *) [7 marks]

Implement a function called `check_record` in `q1.py` to check whether a given record follows a correct format. This function takes a tuple (data type: `tuple`) called `record`. You can assume that this tuple has 3 elements. The first element of the `record` is always a `str`, its second element is an `int`, and the third element of it is also an `int`. This function returns a boolean value (data type: `bool`) that represents whether a given `record` satisfies both of the following two conditions. If so, the function returns `True`, otherwise `False`.

The conditions that the `record` should satisfy are as follows:

1. The second element of the `record` is the length of the first element.
2. The third element of the `record` is strictly greater than the second element.

Examples:

- `check_record(('abc', 3, 51))` returns `True` because the tuple satisfies all conditions (the length of `'abc'` is 3, and 51 is strictly greater than 3).
- `check_record(('apple', 5, 3))` returns `False` because the tuple violates the condition 2 (3 is not greater than 5).
- `check_record(('game', 3, 4))` returns `False` because the tuple violates the condition 1 (the length of `'game'` is 4 not 3).
- `check_record(('', -1, 0))` returns `False` because the tuple violates the condition 1.

Use `q1_test.py` to test your code. Your `q1.py` should contain the function definition of `check_record`. We will only grade your `q1.py` and we will use a different test script for grading.

Question 2 [6 marks]

Part A (Difficulty Level: *)

Implement a function called `get_n_odd_numbers` in `q2a.py`. This function takes in two parameters called `filename` (type: `str`) and `n` (type: `int`). `n` is a non-negative integer.

We assume that the text file of `filename` has `L` lines (`L >= 1`; there is at least one line in the text file), and each line is a number. For example, `sample1.txt` has 6 lines as shown below.

```
100
3
52
621
1812
31
```

The function will return a list of `n` odd numbers from the top of the given file. For example, `get_n_odd_numbers('sample1.txt', 2)` returns `[3, 621]` because 3 and 621 are the first two odd numbers in the file.

If the number of odd numbers in the file is less than `n`, the function returns a list of all the odd numbers stored in the file. For example, `get_n_odd_numbers('sample1.txt', 100)` returns `[3, 621, 31]` because the file does not have 100 odd numbers and thus the function returns all the odd numbers in the file.

More examples:

- `get_n_odd_numbers('sample1.txt', 1)` returns `[3]`
- `get_n_odd_numbers('sample1.txt', 3)` returns `[3, 621, 31]`
- `get_n_odd_numbers('sample1.txt', 0)` returns `[]`

Use `q2a_test.py` to test your code. Your `q2a.py` should contain the function definition of `get_n_odd_numbers`. We will only grade your `q2a.py` and we will use a different test script for grading.

(**Hint**: You can first find all the odd numbers and store them in a list. You can then get the first n elements from this list and return them as a new list.)

Part B (Difficulty Level: **)

Implement a function called `filter_words` in `q2b.py`. This function takes in two parameters called `filename` (type: `str`) and `end` (type: `str`). We note that `end` is a string with a single character, and it is not a space character.

We assume that the text file of `filename` has `L` lines (`L` >= 1; there is at least one line in the text file), and each line can have multiple words. There is always a space between two words. For example, `w_sample1.txt` has 5 lines as shown below.

```
there separate stair phone laptop
coffee bar
milk bubble ice after
air cake player focus
bread chair flair
```

The function will return a list of words ending with `end` in `filename`. Words in the returned list follow the same order of appearance as that in `filename`. If no words ending with `end`, it returns an empty list.

Examples:

- `filter_words('w_sample1.txt', 'e')` returns `['there', 'separate', 'phone', 'coffee', 'bubble', 'ice', 'cake']` in this exact order.
- `filter_words('w_sample1.txt', 'r')` returns `['stair', 'bar', 'after', 'air', 'player', 'chair', 'flair']`.
- `filter_words('w_sample1.txt', 'd')` returns `['bread']`

Use `q2b_test.py` to test your code. Your `q2b.py` should contain the function definition of `filter_words`. We will only grade your `q2b.py` and we will use a different test script for grading.

Question 3 (Difficulty Level: **) [5 marks]

Part A

Implement a function called `read_courses` in `q3a.py`. The function takes the following parameters:

- `filename` (type: `str`): the name of the input file containing the information of courses taken by students.

The input text file consists of multiple lines of course score records. Each record is in one row with five columns separated by the pipe character ('|').

- The first column is the term ID in the `'yyyy-Tt'` format. Here we assume there are only two terms within each year, i.e., `'T1'` and `'T2'`.
- The second column is the student name.
- The third column is the course name.
- The fourth column is the credit of the course, which is a positive `int`.
- The fifth column is the final score of the course. A course score is a decimal number between 0.0 and 4.0.

You can assume each a student will never take the same course twice. An example of the input text file named `'course_scores.txt'` is as follows. Note that the rows are not sorted in any order.

```
2021-T2|Eric Wong|Math|2|3.6
2022-T1|Eric Wong|Chemistry|1|4.0
2021-T1|Andy Lau|English|2|3.6
2022-T1|Jeffrey Heer|Chemistry|1|3.8
2022-T1|Andy Lau|Math|2|4.0
```

The function returns a dictionary in which the keys are the student names, and the corresponding values are lists of tuples. Each tuple contains the course information of a student in the form of `(course name, year, term, credit, score)` exactly as shown in the input file. For each list, the tuples must appear in the same order as that in the input file.

Example:

- `read_courses('course_scores.txt') returns {'Eric Wong': [('Math', 2021, 2, 2, 3.6), ('Chemistry', 2022, 1, 1, 4.0)], 'Andy Lau': [('English', 2021, 1, 2, 3.6), ('Math', 2022, 1, 2, 4.0)], 'Jeffrey Heer': [('Chemistry', 2022, 1, 1, 3.8)]}`

Use `q3a_test.py` to test your code. Your `q3a.py` should contain the function definition of `read_courses`. We will only grade your `q3a.py` and we will use a different test script for grading.

Part B

Implement a function called `calculate_student_gpa` in `q3b.py` to calculate students' weighted average GPA in a specific period. The function takes the following parameters:

- `filename` (type: `str`): the name of a file containing the information of courses taken by students as described in Part A.
- `start_term` (type: `tuple`): a tuple specifying the starting term of the courses to be included in the calculation of the course GPAs. It consists of two positive `int` elements: year and term. For example, `(2022, 1)` indicates the first term of 2022.
- `end_term` (type: `tuple`): a tuple specifying the ending term of the courses to be included in the calculation of the course GPAs. All the courses of `start_term`, the courses of `end_term`, and any courses in between are included in the GPA calculation.

The function returns a dictionary that stores 1) the total number of courses taken by each student from `start_term` (inclusive) to `end_term` (inclusive) and 2) the corresponding weighted GPA. The weighted GPA is calculated by using the course credits and should keep ***one decimal point***. We can assume that each student takes at least one course in the duration from `start_term` to `end_term`. You must use the built-in function `round()` to specify the number of decimal points of the weighted GPAs. For example, `round(5.76543, 1)` will return `5.8`.

The weighted GPA is calculated using the following formula:

$$gpa = \frac{c_1 * s_1 + c_2 * s_2 + \cdots + c_K * s_K}{c_1 + c_2 + \cdots + c_K}$$

where K is the total number of courses taken by a student in the specified period, c_i is the credit of the `i`-th course and s_i is the final score of the `i`-th course.
You can import and use the function implemented in Part A of this question.

Examples:

- If the function is invoked like this:
  `calculate_student_gpa('course_scores.txt',(2022,1),(2022,1))`
  it should return
  `{'Eric Wong':(1,4.0), 'Andy Lau': (1,4.0), 'Jeffrey Heer': (1,3.8)}`

- If the function is invoked like this:
  `calculate_student_gpa('course_scores.txt',(2021,1),(2022,1))`
  it should return
  `{'Eric Wong':(2,3.7), 'Andy Lau': (2,3.8), 'Jeffrey Heer': (1,3.8)}`

  Note: The weighted GPA of `'Eric Wong'` from the first term of 2021 to the first term of 2022 is calculated as: $\frac{3.6 * 2 + 4.0 * 1}{2+1} = 3.7$; Similarly, the weighted GPA of `'Andy Lau'` is calculated as: $\frac{3.6*2+4.0*2}{2+2} = 3.8$.

Use `q3b_test.py` to test your code. Your `q3b.py` should contain the function definition of `calculate_student_gpa`. We will only grade your `q3b.py` and we will use a different test script for grading.

Question 4 (Difficulty Level: ***) [7 marks]

Note that Part A is NOT related to Part B/C of this question.

Part A

In `q4a.py`, implement a function called `get_flight_durations`. This function takes in the following parameter:

- `filename` (type: `str`): the name of a file containing the departure and arrival information of various flights in a **single day**, starting from 1:00 AM till 11:59 PM, as obtained from `flightradar24.com`. Note that all the timings in the file are in the same Singapore time zone. Each flight's duration is less than 12 hours. The format of the file will be explained below.

The function returns a Python dictionary, in which keys are the names of airlines, e.g., `'Singapore Airlines'`, `'AirAsia'`, etc., and the corresponding values are lists of tuples. Each tuple represents information of a flight operated by the airline in the form of `(flight_code, origin, destination, scheduled_duration, actual_duration)`, **only if both the departure and arrival information of that flight can be found in** `filename`. Note that the departure and arrival times of any flights in the file are on the same day, i.e., there will be no flight departing and arriving on different days. In each tuple, `flight_code`, `origin`, and `destination` are `str` values. The `scheduled_duration` and `actual_duration` of each flight are in minutes (positive `int`). The scheduled duration of a flight is calculated by:

```
scheduled_duration = scheduled_arrival_time – scheduled_departure_time
```

In the same way, the actual duration of a flight is calculated by:

```
actual_duration = actual_arrival_time – actual_departure_time
```

Each flight is represented in `filename` as a row of text containing several words separated by a single space character. Each line has nine (9) data fields in the order as shown below, where each field can have one or more words separated by a single space character. Note that the rows are not sorted in any order.

```
SCHEDULED_TIME FLIGHT_CODE LOCATION LOCATION_CODE AIRLINE AIRCRAFT
AIRCRAFT_NUMBER DEPARTED/LANDED ACTUAL_TIME
```

An example of the input file named `'flight1.txt'` is shown below. **Note that the line number for each line shown in the 1st column is not part of the file**.

```
1 9:35 AM AK705 Singapore (SIN) AirAsia A320 (9M-AGI) Departed 9:43 AM
2 1:10 PM MH1436 Langkawi (LGK) Malaysia Airlines B738 (9M-MLI) Departed 1:25 PM
3 10:40 AM AK705 Kuala Lumpur (KUL) AirAsia A320 (9M-AGI) Landed 10:29 AM
4 9:05 PM PR2526 Cagayan de Oro (CGY) Philippine Airlines A320 (RP-C8613) Landed 10:23 PM
5 12:30 PM SQ177 Singapore (SIN) Singapore Airlines B78X (9V-SCD) Departed 12:38 PM
6 10:55 AM AK1721 Penang (PEN) AirAsia A320 (9M-AJN) Landed 10:41 AM
7 2:15 PM MH1436 Kuala Lumpur (KUL) Malaysia Airlines B738 (9M-MLI) Landed 2:14 PM
8 9:25 AM AK1721 Singapore (SIN) AirAsia A320 (9M-AJN) Departed 9:28 AM
```

- `SCHEDULED_TIME`: the scheduled (departure or arrival) time of the flight. It is defined by the first two words in each line, separated by a single space. It indicates the time in a single day, e.g., `'9:35 AM'`, `'12:30 PM'`, etc. Note that there can be a leading zero for the minutes, but no leading zero for the hour. For example, `'9:05 PM'` is a valid time representation in the file, but not `'09:05 PM'`. It is assumed that the times are in the range of `[1:00 AM – 11:59 PM]`. This field represents either the scheduled departure or scheduled arrival time of the flight as determined by the field `'DEPARTED/LANDED'` in the same line, as explained below.
- `FLIGHT_CODE`: a single word indicating the code of a particular flight, e.g., `'AK705'`. There can be at most two rows with the same flight code in the input file: one for departure, and the other one for arrival.
- `LOCATION`: one or more words separated by a space character indicating the name of the city/state, e.g., `'Cagayan de Oro'`. This field represents either the destination or origin of the flight as explained later by the field `'DEPARTED/LANDED'` of the same line.
- `LOCATION_CODE`: a single word indicating the abbreviated name for a city/state used in aviation. The location code is always enclosed by a pair of parentheses, e.g., `'(SIN)'`.
- `AIRLINE`: one or more words separated by a space character indicating the name of the airline operating this flight, e.g., `'Singapore Airlines'`.
- `AIRCRAFT`: a single word indicating the model of the aircraft used in this flight, e.g., `'A320'`.
- `AIRCRAFT_NUMBER`: a single word indicating the registration number of the aircraft. This field is always enclosed by a pair of parentheses, e.g., `'(9M-MLI)'`.
- `DEPARTED/LANDED`: this field has only one of two possible values, which are `'Departed'` or `'Landed'`.
    - If the value is `'Departed'`, the `LOCATION` field indicates the destination of the flight; otherwise, the `LOCATION` field indicates the origin of the flight.
    - This field can also be used to determine departure or arrival times. If the value is `'Departed'`, the `SCHEDULED_TIME` and `ACTUAL_TIME` fields indicate the flight's scheduled and actual departure time, respectively. Otherwise, these two fields will indicate the flight's scheduled and actual arrival time, respectively.
- `ACTUAL_TIME`: the actual time (departure or arrival) of the flight. The format is the same as that for `SCHEDULED_TIME`.

In the above example file, line 1 shows a flight by `AirAsia` with the flight code of `AK705` with `Singapore` as the destination. Its scheduled departure time was `9:35 AM`, but its actual departure time was `9:43 AM`. Similarly, line 7 shows a flight operated by `Malaysia Airlines` originated from `Kuala Lumpur`. It landed at `2:14 PM`, while its scheduled arrival time was `2:15 PM`.

Example:

- `get_flight_durations('flight1.txt')` returns the following dictionary `{'AirAsia': [('AK705', 'Kuala Lumpur', 'Singapore', 65, 46), ('AK1721', 'Penang', 'Singapore', 90, 73)], 'Malaysia Airlines': [('MH1436', 'Kuala Lumpur', 'Langkawi', 65, 49)]}`. Note that the origin and destination of each flight in the dictionary do not include the location code. The scheduled and actual durations of each flight are in minutes (positive `int`). The order of the tuples in **each list** must follow the same order of appearance of `flight_code` in `filename`. Any flights that do not have both the departure and arrival information are not included in the returned dictionary.

The function returns an empty dictionary if there are no flights with both departure and arrival information in the input file.

Use `q4a_test.py` to test your code. Your `q4a.py` should contain the function definition of `get_flight_durations`. We will only grade your `q4a.py` and we will use a different test script for grading.

Part B

In `q4b.py`, implement a function called `find_words`. This function takes in the following parameters:

- `word_list` (type: `list`): a non-empty list of words. Each word is a non-empty string of lowercase alphabets.
- `input_str` (type: `str`): a non-empty string of lowercase alphabets.
- `step` (type: `int`): a positive integer.

For each word `w` in `word_list`, the function finds if it is possible to construct `w` using the characters in `input_str` when we scan `input_str` from left to right. The rule is that if the character `input_str[i]` is used for `w[k]`, i.e., `w[k] == input_str[i]`, another character `input_str[j]` will be used for `w[k+1]` only if `w[k+1] == input_str[j]`, and `j - i >= step`.

The function returns **a list of lists** of corresponding indexes for characters in `input_str` that can be used to construct each word in `word_list`:

- There can be multiple instances of characters in `input_str` that we can use to construct a word in `word_list`. In this part of the question, the function `find_words` only finds the first instance of character indexes which meet the above rule.
- If a word cannot be constructed using the above rule, an empty list should be included for that word in the returned list.

Examples:

- `find_words(['no', 'cat', 'doog', 'not'], 'caaaatdoogname', 1)` returns `[[], [0, 1, 5], [6, 7, 8, 9], []]` **in this exact order**. We note that the first word in the list could not be constructed using the characters in the input string when we scan the string from left to right following the above rule. The second word in the list, `'cat'`, can be constructed using characters at indexes of 0, 1, and 5 in the input string, given the `step` of 1. Note that we only return the list consisting of the first set of indexes that meets the rule when scanning the input string from left to right. As a result, the list `[0, 2, 5]` is not a valid answer for `'cat'` this question. The corresponding index lists in the returned list must be in the same order as that in the `word_list`. The indexes in each index list must be in increasing order.

- `find_words(['cat', 'dog'], 'gaaaatttdoonagmedogcaaggt', 2)` returns `[[19, 21, 24], [8, 10, 13]]`.

- `find_words(['notin'], 'notin', 3)` returns `[[]]`.

Use `q4b_test.py` to test your code. Your `q4b.py` should contain the function definition of `find_words`. We will only grade your `q4b.py` and we will use a different test script for grading.

Part C

In `q4c.py`, implement a function called `find_all_words`. This function takes in the same parameters as described in Part B of this question. The function returns a list of **all** possible lists of corresponding indexes for characters in `input_str` that can be used to construct each word in `word_list`, following the same rule defined in Part B. If a word cannot be constructed using the above rule, an empty list should be included in the returned list for that word.

Examples:

- `find_all_words(['pane', 'bull', 'not'], "bbulllpanredno", 1)` returns `[[6, 7, 8, 10], [0, 2, 3, 4], [0, 2, 3, 5], [0, 2, 4, 5], [1, 2, 3, 4], [1, 2, 3, 5], [1, 2, 4, 5], []]`. Note that the lists must appear in this exact order, e.g., all the lists of indexes for `'pane'` must appear before the lists for `'bull'`. All index lists for a word must appear in an increasing order considering each index in a list, starting with the first index, and so on. For example, for the word `'bull'`, the list `[0, 2, 3, 4]` must appear before the list `[0, 2, 3, 5]`. Similarly, the list `[0, 2, 4, 5]` must appear before the list `[1, 2, 3, 4]` but after `[0, 2, 3, 5]`.

- `find_all_words(['plane', 'bull', 'not'], "planredbbunloll", 2)` returns `[[], [7, 9, 11, 13], [7, 9, 11, 14], []]`.

- `find_all_words(['cat', 'dog', 'pig'], "catdog", 1)` returns `[[0, 1, 2], [3, 4, 5], []]`.

- `find_all_words(['cat', 'dog', 'pig'], "catdog", 2)` returns `[[], [], []]`

Use `q4c_test.py` to test your code. Your `q4c.py` should contain the function definition of `find_all_words`. We will only grade your `q4c.py` and we will use a different test script for grading.

END OF PAPER