Lab Test 1 [25 marks]

Coding time: 1.5 hours

General Instructions:

1. You will perform the lab test on your personal laptop.

2. You are not allowed to communicate with anyone or access any network during the test. Disable the following connections on your laptop before the test begins: Wi-Fi, Bluetooth, and any other communication devices (e.g. 3G/4G modems).

4. You may refer to any file on your laptop during the test.

5. 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, misspelling, etc. in the output.

6. Do not hardcode. We will use different test cases to test and grade your solutions.

7. Follow standard Python coding conventions (e.g., naming functions and variables).

8. Python script file 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.

9. 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: Lum Ting Wong
# Email ID: lum.ting.wong.2019
```

Instructions on how to submit your solutions:

1. Before the test begins, you will be instructed to download the resource files of the test from eLearn. After downloading the resource files, you should disble the internet on your laptop.

2. You will be given a password to unzip the downloaded .zip file. After unzipping the file, **you should rename the folder to your email id.** For example, if your SMU email is lum.ting.wong.2019@sis.smu.edu.sg, you should rename the folder to lum.ting.wong.2019. You need to save your solutions to this folder for submission later. You may be penalized for not following our instructions.

3. When the test ends, you will be instructed to enable your laptop's internet and submit your solutions as a single zip file to eLearn.

Question 1 **(Difficulty Level: *)** **[5 marks]**

Inside **q1.py**, implement a function called `check_sum()`. The function takes in three integers (called `n1`, `n2` and `n3`) as its parameters. You can assume that `n1`, `n2` and `n3` are always integers.

The function returns `True` if one of the numbers is the sum of the other two numbers; otherwise, it returns `False`.

For example, `check_sum(4, 2, 1)` returns `False` because we cannot find a number among 4, 2 and 1 that is the sum of the other two numbers. On the other hand, `check_sum(3, 5, 2)` returns `True` because 5 = 3 + 2.

Some other examples are shown below:

- `check_sum(2, -1, 1)` returns `True` because 1 = 2 + (-1).
- `check_sum(1, 0, 0)` returns `False`.
- `check_sum(-1, 0, -1)` returns `True` because -1 = 0 + (-1).

Use **q1_test.py** to test your code. You do not need to modify `q1_test.py`. We will only grade your `q1.py`.

Question 2 [7 marks]

Part (a) (Difficulty Level: *)

In **`q2a.py`**, implement a function called `get_number_of_long_strings()`. The function takes in the following parameters:

- `str_list` (type: `list`): This is a list of strings. You can assume that each element in this list is a string. It is possible for `str_list` to be empty.
- `n` (type: `int`): This is a number. You can assume that this is always a positive integer.

The function returns the number of strings in `str_list` that are **strictly** longer than `n` (i.e., a string whose length is exactly `n` is NOT included). If there's no string in `str_list` longer than `n` characters, the function returns 0.

For example, `get_number_of_long_strings(['123456', 'abc', '++++', '!@#$%'], 4)` returns 2, because among the strings in the list, only `'123456'` and `'!@#$%'` are longer than 4 characters.

Some other examples are shown below:

- `get_number_of_long_strings(['!!!!', 'apple', 'sheep', 'dinosaur', '@'], 6)` returns 1 because only the string `'dinosaur'` contains more than 6 characters.
- `get_number_of_long_strings(['abcdefg', '1234567890'], 100)` returns 0 because there is no string in the list with more than 100 characters.
- `get_number_of_long_strings([''], 1)` returns 0.
- `get_number_of_long_strings([], 1)` returns 0.

Use **`q2a_test.py`** to test your code. You do not need to modify `q2a_test.py`. We will only grade your `q2a.py`.

Part (b) (Difficulty Level: **)

In **`q2b.py`**, implement a function called `add_even_numbers()`. The function takes in a list of strings called `str_list` (type: `list`) as its parameter, where each string in `str_list` contains a sequence of positive integers separated by `'|'`, e.g., `'12|89|34'`. The function should add up all the even numbers found in the given list.

You can assume that each string in `str_list` is always properly formatted as described above and always contains at least one integer. It is possible, though, for `str_list` to be empty.

E.g.,

- `add_even_numbers(['1|2|3', '10|50'])` returns 62 (because 2 + 10 + 50 = 62, while 1 and 3 are not added).
- `add_even_numbers(['99|1|27', '11|5'])` returns 0 (because there is no even number).
- `add_even_numbers(['1', '12'])` returns 12.
- `add_even_numbers([])` returns 0.

Use **`q2b_test.py`** to test your code. You do not need to modify `q2b_test.py`. We will only grade your `q2b.py`.

Question 3 **[6 marks]**

The regular entrance fees to a theme park are as follows:

- adult: $75
- child: $50

However, the theme park also offers the following discounted packages:

- Package A (for 2 adults): $140
- Package B (for 1 adult + 1 child): $110
- Package C (for 2 adults + 2 children): $200

Part (a) **(Difficulty Level: *)**

Assume that `n` adults and `n` children are going to the theme park together. Implement a function inside **`q3a.py`** called `calculate_entrance_fees_1()` that takes in `n` as its only parameter. You can assume that `n` is always a non-negative integer. The function returns the lowest total price this group has to pay to enter the theme park.

(**Hint**: Since the number of adults is the same as the number of children, this group should consider **Package B** and **Package C** only to minimize the total price.)

E.g.,

- `calculate_entrance_fees_1(1)` returns 110 (because 1 adult and 1 child can buy 1 Package B).
- `calculate_entrance_fees_1(2)` returns 200 (because 2 adults and 2 children can buy 1 Package C).
- `calculate_entrance_fees_1(7)` returns 710 (because 7 adults and 7 children can buy 3 Package C plus 1 Package B).

Use **`q3a_test.py`** to test your code. You do not need to modify `q3a_test.py`. We will only grade your `q3a.py`.

Part (b) (Difficulty Level: **)

Assume now that `m` adults and `n` children (where `m` may be different from `n`) are going to the theme park together. Write another function called `calculate_entrance_fees_2()` inside **`q3b.py`** that takes in `m` and `n` as two parameters. You can assume that both `m` and `n` are always non-negative integers. The function should return the lowest total price the group needs to pay to enter the theme park.

For example,

- `calculate_entrance_fees_2(2, 1)` returns 185 because 2 adults and 1 child can buy 1 Package B plus 1 regular adult ticket, which is $110 + $75 = $185. This is lower than buying 2 regular adult tickets and 1 regular child ticket, which would be $75 X 2 + $50 = $200.
- `calculate_entrance_fees_2(5, 7)` returns 610 because 5 adults and 7 children can buy
  - 2 Package C
  - 1 Package B
  - 2 regular child tickets

  which is $200 x 2 + $110 + $50 x 2 = $610.
- `calculate_entrance_fees(3, 0)` returns 215 because 3 adults can buy 1 Package A and 1 regular adult ticket.

(**Hint**: You should try to import and call the `calculate_entrance_fees_1()` function you implemented in `q3a.py` to help Part (b).)

Use **`q3b_test.py`** to test your code. You do not need to modify `q3b_test.py`. We will only grade your `q3b.py`.

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

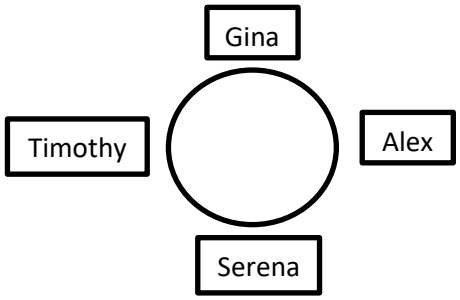
A group of `n` ladies and `n` gentlemen are going to a banquet together, and they are going to sit around a **round** table. Each lady should be seated between two gentlemen. There are, however, some constraints for the seating plan. The constraints fall into two types:

- The first type of constraints specifies two people who must sit next to each other. E.g., `("Serena", "Alex")` indicates that Serena MUST sit beside Alex (either on the left or on the right).
- The second type of constraints specifies two people who cannot sit next to each other. E.g., `("Serena", "Harry")` indicates that Serena CANNOT sit next to Harry.

Part (a)

Implement a function called `check_seating_arrangement()` inside **`q4a.py`**. This function takes in the following 3 parameters:

- `arrangement`: This is a list of `2*n` strings representing the counterclockwise seating arrangement for `n` ladies and `n` gentlemen. For example, `['Alex', 'Gina', 'Timothy', 'Serena']` corresponds to the following arrangement:



  Note that the first person in the list and the last person in the list also sit next to each other.

  You can assume that this list is already arranged such that each lady is between two gentlemen, i.e., the list is always something like (lady, gentleman, lady, gentleman, …) or (gentleman, lady, gentleman, lady, …).

- `must_list`: A list of tuples, where each tuple has two strings representing the names of two people who MUST sit next to each other. The list might be empty. You can assume that each name appearing here is a name that has appeared in the list `arrangement`.

- `cannot_list`: A list of tuples, where each tuple has two strings representing the names of two people who CANNOT sit next to each other. The list might be empty. You can assume that each name appearing here is a name that has appeared in the list `arrangement`.

The function returns `True` if the given arrangement satisfies both types of constraints specified by `must_list` and `cannot_list`. Otherwise, the function returns `False`.

E.g.,

- `check_seating_arrangement(['Serena', 'Timothy', 'Lucy', 'Harry', 'Gina', 'Alex'], [('Serena', 'Alex'), ('Harry', 'Lucy')], [('Gina', 'Timothy')])` returns `True` because all constraints are satisfied (i.e., Serena and Alex are sitting next to each other, Harry and Lucy are sitting next to each other, and Gina and Timonthy are not sitting next to each other.

- `check_seating_arrangement(['Serena', 'Harry', 'Lucy', 'Timothy', 'Gina', 'Alex'], [('Serena', 'Alex'), ('Harry', 'Lucy')], [('Gina', 'Timothy')])` returns `False` because the constraint that Gina cannot sit next to Timothy is not satisfied.

Use **`q4a_test.py`** to test your code. You do not need to modify `q4a_test.py`. We will only grade your `q4a.py`.

Part (b)

Implement a function called `get_seating_arrangement()` inside `q4b.py` that takes in the following 4 parameters:

- `female_list`: A list of `n` strings representing the names of the ladies.
- `male_list`: A list of `n` strings representing the names of the gentlemen.
- `must_list`: A list of tuples, where each tuple has two strings representing the names of two people who MUST sit next to each other. The list might be empty.
- `cannot_list`: A list of tuples, where each tuple has two strings representing the names of two people who CANNOT sit next to each other. The list might be empty.

The function should return a seating arrangement that satisfies all the constraints. (If there are more than one such possible seating arrangements, the function needs to return only one of them.) If no such seating arrangement exists, the function returns `None`.

You can assume that the lengths of `female_list` and `male_list` are the same, and their lengths are not 0.

You can also assume that each name appearing in `must_list` (or `cannot_list`) appears in either `female_list` or `male_list`.

E.g.,

- `get_seating_arrangement(['Gina','Lucy','Serena'], ['Alex','Harry','Timothy'], [('Serena','Alex'),('Harry','Lucy')], [('Gina','Timothy')])` could return `['Serena', 'Timothy', 'Lucy', 'Harry', 'Gina', 'Alex']`. It could also return `['Lucy','Timothy','Serena','Alex','Gina','Harry']` or `['Gina','Alex','Serena','Timothy','Lucy','Harry']`…

- `get_seating_arrangement(['Gina','Lucy','Serena'], ['Alex','Harry','Timothy'], [('Gina','Alex'),('Gina','Harry'),('Harry','Lucy')], [('Timothy','Lucy')])` returns `None` because there is not any seating arrangement satisfying all constraints.

Important Note:

- You should try to import and use the `check_seating_arrangement()` function implemented in `q4a.py` for Part (b) of Q4. If your `check_seating_arrangement()` function is not implemented correctly but your `q4b.py` works correctly with a correct `check_seating_arrangement()` function, you can still get marks for Part (b).
- You can use the `permutations()` function from the library `itertools` to help you generate all permutations of a list. Try the code below and observe the output to understand the behavior of the function `itertools.permutations()`. Note that we use `list()` to convert the returned value of `itertools.permutations()` into a list. We also use `list()` to convert each permutations into a list.

```
import itertools

a_list = ['A', 'B', 'C']
my_permutations = list(itertools.permutations(a_list))
for perm in my_permutations:
    print(list(perm))
```

Use **`q4b_test.py`** to test your code. You do not need to modify `q4b_test.py`. We will only grade your `q4b.py`.

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

A function called `transform()` takes in two strings called `str1` and `str2` as its two parameters. The function tries to transform `str1` into `str2` by swapping two *adjacent* characters (i.e., two characters next to each other) every time. The function returns a list of strings that represents a sequence of transformations that can turn `str1` into `str2`.

You can assume that `str1` can always be turned into `str2` after a sequence of swaps of adjacent characters, i.e., `str1` and `str2` contain the same set of characters. You can also assume that `str1` does not contain any duplicate characters.

E.g., `transform('ABC', 'CBA')` may return `['ABC', 'ACB', 'CAB', 'CBA']` as a sequence of transformations. We can see that in the returned list, each string differs from the previous string with only two adjacent characters swapped, as illustrated below:

A**BC** (swapping B and C) → **AC**B (swapping A and C) → C**AB** (swapping A and B) → CBA

`transform('ABC', 'CBA')` may also return `['ABC', 'BAC', 'BCA', 'CBA']`, which is also a valid sequence of transformations. The function can return any valid sequence of transformations.

Another example: `transform('ABCDE', 'DBECA')` may return `['ABCDE', 'ABDCE', 'ADBCE', 'DABCE', 'DBACE', 'DBAEC', 'DBEAC', 'DBECA']`.

Implement this function `transform(str1, str2)` in **`q5.py`**.

Important Note: You are not allowed to use `while()` loop for this question. You are not allowed to use the `random` module for this question. Basically, you are not allowed to randomly generate swaps to try to solve the problem.

Use **`q5_test.py`** to test your code. You do not need to modify `q5_test.py`. We will only grade your `q5.py`.