General assignment guidelines

1. **All imperative programs should have relevant assertions given as comments at important points.** For example, for each loop you should state the invariant at the top of the loop, and the post-condition below the end of the loop. These can be stated in words instead of mathematically if you prefer.
2. If an analysis of efficiency is asked, it must be carried out for all functions that use recursion or loops, including helper functions.
3. Big O notation should be as tight as possible. For example, reporting $O(n^2)$ complexity for an $O(n)$ algorithm could lose you marks, even though it is technically correct.
4. For the programming component of each question, the name and the type of the main function must be exactly as specified in the question. You can take help of the test cases provided to validate your code.
5. Any function that has integer input and output must not perform any intermediate computation using floating-point numbers.

1 Quiz System Design

In this question, you will design the basics of an automatic quiz system, similar to Moodle, through which students can take quizzes in their courses. Each student has an entry number and a list of courses they are taking. Each course has a course code and a list of quizzes. Each quiz has a title and stores the correct options for each question (assume that the question and answer texts don't need to be stored). Each quiz also keeps track of the attempts that different students have made.

So the classes we will need, and their data attributes, are as follows:

- `Student`: This class will have a *public* attribute `entryNo` in string format, and a *private* attribute for a list of `Course` objects.

- `Course`: This class will have a *public* attribute `courseCode` in string format, and a *private* attribute for a list of `Quiz` objects.
- `Quiz`: This class will have a *public* attribute `title` in string format, and a *private* attribute for the list of correct options. It may also need other private attributes to store the students' attempts so that they can be retrieved later.

Objects will be created in this manner.

```
col100q1 = Quiz('Quiz1', ['a','b','b'])
col100q2 = Quiz('Quiz2', ['b','d','c'])
col100 = Course('COL100', [col100q1, col100q2])

mtl100q1 = Quiz('Quiz1', ['a','b','d'])
mtl100q2 = Quiz('Quiz2', ['d','c','a'])
mtl100 = Course('MTL100', [mtl100q1, mtl100q2])

s1 = Student('2019MCS2562', [col100, mtl100])
s2 = Student('2017CS10377', [col100])
```

Now add the following public methods to the `Student` class. You may need to add further public methods to the other classes to make them possible.

1. `attempt(self, courseCode, quizTitle, attemptedAnswers)`: Record the student's answers to the quiz with the given title in the course with the given code. If the quiz has already been attempted by the student, ignore the new attempt.
2. `getUnattemptedQuizzes(self)`: Return a list of pairs `(courseCode, quizTitle)` representing quizzes in the student's courses that they have not attempted yet.
3. `getAverageScore(self, courseCode)`: Return the student's average score, i.e. total correct answers / number of *attempted* quizzes, in the course with the given code.

For example, with the example objects given above, you should get

```
s2.attempt('COL100', 'Quiz1', ['a','b','c'])
s2.getUnattemptedQuizzes()   # returns [('COL100', 'Quiz2')]
s2.getAverageScore('COL100') # returns 2
```

2 Matrices

1. Design a class named `Matrix` which allows us to do linear algebra calculations. Implement the appropriate special methods `__str__`, `__add__`, `__sub__`, `__mul__` so that the class can be used as follows:

```
A = Matrix([[1,2,3], [4,5,6], [7,8,9]])
B = Matrix([[-2,0,5], [0,0,0], [0,10,0]])

A + B   # matrix addition
B - A   # matrix subtraction
A * B   # matrix multiplication
B * 2   # scalar multiplication
```

`print(B)` should give nicely formatted output, e.g.

```
-2  0  5
 0  0  0
 0 10  0
```

2. Note that for an $m \times n$ matrix, the `Matrix` class always takes $O(mn)$ space. If only a few of the entries of the matrix are nonzero, we can do better. Design a class named `SparseMatrix` which stores only the nonzero entries in each row. For example, the matrix `B` above can stored as `[[(0,-2), (2,5)], [], [(1,10)]]`, since the 0th row has -2 at the 0th position and 5 at the 2th position, and so on.

   Implement all the same methods — printing, addition, subtraction, multiplication — for `SparseMatrix`. Assume that we not mix `Matrix` and `SparseMatrix` in these operations.

3. Add a method `toSparse()` in `Matrix`, and a method `toDense()` in `SparseMatrix`. These should return a new `SparseMatrix` or `Matrix` respectively, representing the same data.

3 Bonus: Maze Traversal

Time for a game. The objective is to find a path through a maze. The maze will be represented as a 2D array, i.e. a list of lists, containing the strings `'_'` (an empty space), `'X'` (a wall you cannot walk through), `'S'` (the starting point), and `'E'` (the end point).

The rules of the traversal are as follows. You may only walk one step up (U), one step right (R), one step left (L), or one step down (D). You are not allowed diagonal moves. If a diagonal move is to be made, you have to make it in two moves.

For example, given the maze

```
X X X X X E X X X
X _ _ _ _ _ X _ X
X _ _ X _ X _ _ X
X X _ X _ _ _ X X
X X _ _ _ X X X X
X X X S X X X X X
```

your function should return a *list of strings* giving the steps to be taken to reach from the starting point `S` to the end point `E`. For example, one valid path is [U, R, U, U, U, R, R, U]. Notice that there may be multiple paths in the maze that can yield the right answer; you may report any one such path. Two other valid paths in our example are [U, L, U, U, L, U, R, R, R, R, U] and [U, L, U, U, U, R, R, R, U].

Your path may visit any empty space at most once. Re-visiting an empty space means you went around in a circle, which is wrong (there is no point in going around in circles in a maze).

1. Write a function `traverseMaze(mazeFile)`. The input to the function will be the file-name of a text file that contains the maze, written in the format described above. The function should read the file, compute the path to be traversed in the maze, and return it as a list of strings.
2. Prove all your assertions and invariants, and use them to argue the correctness of your algorithm.

Hint: Consider using a new 2D array to keep track of the spaces you have visited so far, and a stack to remember the order you visited them in.