Problem Solving Methodology in IT (COMP1001)

Pre-exam Test

20 November 2018

Please read the instructions and information below carefully.

1. Before the commencement of the test:
    a. Put away your phones, calculators, and other electronic devices. You cannot use them during the test.
    b. Take a deep breath and relax. Don't let your mind dwell on the negative what-if questions.
2. About the questions and submissions:
    a. Each question carries the same weight.
    b. The overall mark of the test component is based on the best five out of the 10 questions in the two tests.
    c. Submit the *.py* files for all questions to Blackboard. Name your files by *"your student ID"_Q1.p*y for Q1 and so on. You should upload your *.py* file as soon as you finish one. There is no limit on the number of submissions.
3. During the test:
    a. You can use everything available in your PC and access everything under Blackboard and nothing else.
    b. You may use pencil/ball pen, eraser and ruler and nothing else.
    c. No communication with others in any form
    d. Bathroom policy: one at any time. Raise your hand and get permission before going.
    e. Raise your hand if you have any question about the test paper.
4. At the end of the test period:
    a. The test will end at 4:20pm for group 2 and at 6:20pm for group 1. You must stop when the time is up.
    b. If you finish early, you may let us know and shut down your PC. But you cannot leave the venue until the end of the test. You may use your own notebook to work on other coursework.
    c. Please logout your machine and clean up your table before leaving.

1. [Weight = 1] Implement a Python function called *compareBinary*() that will compare two strings of binary numbers. The strings must include the radix point. For example, *compareBinary*("1.11","11.11") returns -1 (less than), *compareBinary*("11.11","11.11") returns 0 (equal to), and *compareBinary*("11.11","1.11") returns 1 (greater than). It is acceptable to have leading zeroes and trailing zeroes, such as "01.1" and "1.10". You cannot use any Python built-in functions, such as *eval*(), *float*(), and *len*(), and import modules. But you are allowed to use any string methods and operators (such as >, ==, and <).

   The two string methods below may be useful to you:
   - `"00110".lstrip("0")` → `"110"` for removing the leading zeroes
   - `"01100".rstrip("0")` → `"011"` for removing the trailing zeroes

```
Function compareBinary(s1, s2)
    Input: s1 and s2 are strings of binary numbers which must contain
    the radix point.
    Output:
    Return 1 if the binary number in s1 is greater than that in s2.
    Return 0 if the binary number in s1 is equal to that in s2.
    Return -1 if the binary number in s1 is less than that in s2.
```

   Include the statements below in your *.py* file. The expected results are 1, -1, 0, 0, and 0.

```
print(compareBinary("10.11","1.11"))
print(compareBinary("10.0","10.01"))
print(compareBinary("011.1","11.1"))
print(compareBinary("11.1","11.10"))
print(compareBinary("0.",".0"))
```

2. [Weight = 1] In the class we have examined how the limited computer memory could affect the accuracy of representing real numbers. In this question you are asked to use the string's format method to print out the values of *n*, and their values of *round*(*n*, *m*), where *n* = 0.45, 1.45, …, 7.45, and *m* = 1, 2, and 25.

   Below shows the required format for the outputs for the first 5 values. Please note that
   - the number of places in the fractional part is 25;
   - no additional space between each value of *n* and the text before and after it;
   - 4 spaces are used to print each value of *m* in left-justified manner;
   - all trailing zeros are removed; and
   - there is an empty line between the printouts for adjacent values.

   All other formatting details are not essential.

```
The value of 0.45 is 0.4500000000000000111022302  m is 1   : 0.5
The value of 0.45 is 0.4500000000000000111022302  m is 2   : 0.4500000000000000111022302
The value of 0.45 is 0.4500000000000000111022302  m is 25  : 0.4500000000000000111022302

The value of 1.45 is 1.4499999999999999555910790  m is 1   : 1.399999999999999911182158
The value of 1.45 is 1.4499999999999999555910790  m is 2   : 1.449999999999999955591079
The value of 1.45 is 1.4499999999999999555910790  m is 25  : 1.449999999999999955591079

The value of 2.45 is 2.4500000000000001776356839  m is 1   : 2.5
The value of 2.45 is 2.4500000000000001776356839  m is 2   : 2.450000000000000177635684
The value of 2.45 is 2.4500000000000001776356839  m is 25  : 2.450000000000000177635684

The value of 3.45 is 3.4500000000000001776356839  m is 1   : 3.5
The value of 3.45 is 3.4500000000000001776356839  m is 2   : 3.450000000000000177635684
The value of 3.45 is 3.4500000000000001776356839  m is 25  : 3.450000000000000177635684

The value of 4.45 is 4.4500000000000001776356839  m is 1   : 4.5
The value of 4.45 is 4.4500000000000001776356839  m is 2   : 4.450000000000000177635684
The value of 4.45 is 4.4500000000000001776356839  m is 25  : 4.450000000000000177635684
```

:

:

   For *round*(1.45,1), why is the actual value not rounded to only 1 fractional place like other cases? Answer it in a Python comment at the end of your *.py* file.

3. [Weight = 1] In Q1 of Assignment 7, we use another data abstraction to solve the MCGW problem. As recalled, the state is given by *ABC*, where *A*, *B*, *C* are the locations (*E/W*) of the cabbage, goat, and wolf, respectively. In this question, you are asked to implement the two functions below. The second function *fullyConnectedGraph*() is used for generating the answer to part (b) of Q1 in Assignment 7. You cannot import any Python module.

   [Weight = 0.3]

```
Function genStates():
    """
    A function to generate a set of all possible states (E/W,E/W,E/W)
    Input: None
    Output: Return a set of all possible states in a Python tuple. Each
    state is implemented as a string of three binary values 'E'/'W'.
    """
```

   [Weight = 0.7]

```
Function fullyConnectedGraph(S):
    """
    A function to generate a graph which contains all possible links
    among the 8 nodes
    Input: S is a set of the 8 legal states.
    Output: Return a graph which contains all links (whether legal or
    illegal) among the 8 nodes. See below for the expected output.
    """
```

   Include the following statement in your *.py* file.

```
print(fullyConnectedGraph(genStates()))
```

   The expected output is:

```
{'EEE': ['EEW', 'EWE', 'WEE'], 'EEW': ['EEE', 'EWW', 'WEW'],
'EWE': ['EEE', 'EWW', 'WWE'], 'EWW': ['EEW', 'EWE', 'WWW'],
'WEE': ['EEE', 'WEW', 'WWE'], 'WEW': ['EEW', 'WEE', 'WWW'],
'WWE': ['EWE', 'WEE', 'WWW'], 'WWW': ['EWW', 'WEW', 'WWE']} .
```

4. [Weight = 1] This question is an extension to Q3 of Assignment 7. In this question you are asked to implement two more functions for this group of 170 people.

```
def leastPopular(peopleDict):

    """
    Input: peopleDict is a dictionary of all people.
    Output: Return a set of least popular people.
    """
def averagePopularity(peopleDict):
    """
    Input: peopleDict is a dictionary of all people.
    Output: Return the average number of friends that a person has.
    """
```

At the end of Content of Blackboard, you will find *data.txt*. Use the set of *allPeople* and dictionary of *friends*, and the two statements below to find and print a set of least popular people and the average number of friends that a person has.

```
print(leastPopular(friends))
print(averagePopularity(friends))
```

5. [Weight = 1 In the number-guessing game problem of A3, we use *random.randint*(*a, b*) to generate a random number from the range [*a*,*b*] (including *a* and *b*), where *a* and *b* are integers and $a \leq b$. In this question we answer a simple question about this random number generator: how many distinct numbers will this function generate after invoking it a number of times? Implement the function below to answer this question.

```
Function uniquePercentage(lower, upper, numTrial)
   Input:
      lower: an integer that is the lower bound of the range.
      Upper: an integer that is the upper bound of the range.
      numTrail: the number of times of invoking random.randint(lower,
                upper)
   Output:
      Return the percentage of the numbers in [lower,upper] generated
      by random.randint(lower,upper)
```

For example, if *random.randint*(1,10) generates 1, 5, 6, and 9 after invoking it a number of times, the function returns 40 (i.e., 40%).

Include the code below in your *.py* file for testing.

```
for n in range(100,251):
    print(n, uniquePercentage(1,100,n))
```

Hint: Set has a very useful property that it does not have duplicate elements.

End