UNIVERSITY OF SASKATCHEWAN

Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 3

ADTs and Testing

Date Due: June 6, 2019, 7pm **Total Marks: 58**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help. It might help you, but it adds an extra step for the markers.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Read the purpose of each question. Read the Evaluation section of each question.

Version History

- **05/30/2019**: released to students

Question 1 (10 points):

Purpose: Completing a test script for an ADT.

Degree of Difficulty: Easy.

On the course Moodle, you'll find:

- The file `Statistics.py`, which is an ADT covered in class and in the readings. For your convenience, we removed some of the calculations and operations (e.g., `var()` and `sampvar()`, that were not relevant to this exercise, which would have made testing too onerous.
- The file `test_statistics.py`, which is a test-script for the `Statistics` ADT. This test script currently only implements a few basic tests.

In this question you will complete the given test script. Study the test script, observing that each operation gets tested, and sometimes the tests look into the ADT's data structure, and sometimes, the operations are used to help set up tests. You'll notice that there is exactly one test for each operation, which is inadequate.

Design new test cases for the operations, considering:

- Black-box test cases.
- White-box test cases.
- Boundary test cases, and test case equivalence classes.
- Test coverage (percentage of functions tested)
- Unit vs. integration testing.

Running your test script on the given ADT should report no errors, and should display nothing except the message `'*** Test script completed ***'`.

What to Hand In

- A Python script named `a3q1_testing.py` containing your test script.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your test cases for `Statistics.add()` have good coverage.
- 5 marks: Your test cases for `Statistics.mean()` have good coverage.

Addendum on floating point computations

Floating point values use a finite number of binary digits ("bits"). For example, in Python, floating point numbers use 64 bits in total. Values that require fewer than 64 bits to represent are represented exactly. Values that require more than 64 bits to represent are limited to 64 bits exactly, but truncating the least significant bits (the very far right).

You are familiar with numbers with infinitely many decimal digits: π, for example, is often cut off at 3.14 when calculating by hand. Inside a modern computer, π is limited to about 15 decimal digits, which fits nicely in 64 bits. Because long fractions are truncated, many calculations inside the computer are performed with numbers that have been truncated, leading to an accumulation of small errors with every operation.

Another value with an infinite decimal fraction is the value 1/3. But because a computer uses binary numbers, some values we think of naturally as "finite" are actually infinite. For example, 1/10 has a finite decimal representation (0.1) but an infinite binary representation.

The errors that come from use of floating point are unavoidable; these errors are inherent in the accepted standard methods for storing data in computers. This is not weakness of Python; the same errors are inherent in all modern computers, and all programming languages.

We have to learn the difference between *equal*, and *close enough*, when dealing with floating point numbers.

- A floating point literal is always equal to itself. In other words, there is no randomness in truncating a long fraction; the following script will display `Equal` on the console.

```
if 0.1 == 0.1:
    print('Equal')
else:
    print('Not equal')
```

- An arithmetic expression involving floating point numbers is equal to itself. In other words, there is no randomness in errors resulting from arithmetic operations; the following script will display `Equal` on the console.

```
if 0.1 + 0.2 + 0.3 == 0.1 + 0.2 + 0.3:
    print('Equal')
else:
    print('Not equal')
```

- If two expressions involving floating point arithmetic are different, the results may not be equal, even if, in principle, they *should be* equal. In other words, errors resulting from floating point arithmetic accumulate differently in different expressions. The following script will display `Not equal` on the console.

```
if 0.1 + 0.1 + 0.1 == 0.3:
    print('Equal')
else:
    print('Not equal')
```

As a result of the error that accumulates in floating point arithmetic, we have to expect a tiny amount of error in every calculation involving floating point data. We should almost never ask if two floating point numbers are equal. Instead we should ask if two floating point numbers are *close enough* to be considered equal, for the purposes at hand.

The easiest way to say *close enough* is to compare floating point values by looking at the absolute value of their difference:

```
# set up a known error
calculated = 0.1 + 0.1 + 0.1
expected = 0.3

# now check for exactly equal
if calculated == expected:
   print('Exactly equal')
else:
   print('Not exactly equal')

# now compare absolute difference to a pretty small number
if abs(calculated - expected) < 0.000001:
   print('Close enough')
else:
   print('Not close enough')
```

The Python function `abs()` takes a numeric value, and returns the value's absolute value. The absolute value of a difference tells us how different two values are without caring which one is bigger. If the absolute difference is less than a well-chosen small number (here we used 0.000001), then we can say it's close enough.

In your test script for this question, you can check if the ADT calculates the right answer by checking if its answer is close enough to the expected value. If it's not, there's a problem!

Question 2 (10 points):

Purpose: Adding operations to an existing ADT; completing a test script

Degree of Difficulty: Easy.

On the course Moodle, you'll find:

- The file `Statistics.py`, which is an ADT covered in class and in the readings. For your convenience, we removed some of the calculations and operations (e.g., `var()` and `sampvar()`, that were not relevant to this exercise, which would have made testing too onerous.

In this question you will define three new operations for the ADT:

- `count(stat)` Returns the number of data values that the Statistics record `stat` has already recorded.
- `maximum(stat)` Returns the maximum value ever recorded by the Statistics record `stat`. If no data was seen, returns `None`.
- `minimum(stat)` Returns the minimum value ever recorded by the Statistics record `stat`. If no data was seen, returns `None`.

Hint: To accomplish this task, you may have to modify other operations of the `Statistics` ADT.

You will also improve the test script from Q1 to test the new version of the ADT, and ensures that all operations are correct. Remember: you have to test all operations because you don't want to introduce errors to other operations by accident; the only way to know is to test all the operations, even the ones you didn't change. To make the marker's job easier, label your new test cases and scripts so that they are easy to find.

What to Hand In

- A text file named `a3q2_changes.txt` (other acceptable formats) describes changes you made to the existing ADT operations. Be brief!
- A Python program named `a3q2.py` containing the ADT operations (new and modified), but no other code.
- A Python script named `a3q2_testing.py` containing your test script.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 1 mark: Your added operation `count(stat)` is correct.
- 1 mark: Your added operation `maximum(stat)` correct.
- 1 mark: Your added operation `minimum(stat)` correct.
- 3 marks: Your modifications to other operations are correct.
- 4 marks: Your test script has good coverage for the new operations.

Question 3 (28 points):

Purpose: To implement an ADT from a requirements specification.

Degree of Difficulty: Moderate. Time is the main factor here.

Bioinformatics is an area of computer science that uses computation to help solve biological problems. Biological data is often represented as simple strings. For instance, an organism's DNA (genetic information) is typically represented as a string (often called a sequence) comprised of the characters 'A', 'T', 'G' and 'C'.

You are required to design and implement an ADT named Experiment. This data structure will store a collection of DNA sequences (strings) and be used to carry out some common bioinformatics operations. Multiple sequences will be read in from a file and stored in a list. The file `seqeunces1.txt` is provided as a small example input file. If you open this file in a text editor the contents should look something like this:

```
>gene1
ATGATGATGGCG
>gene2
GGCATATC
CGGATACC
>gene3
TAGCTAGCCCGC
```

This file is very similar to the type of output generated by many biological experiments. While knowledge of the underlying biology is not important for this question, there are a few characteristics of this file to note. First, individual sequences are separated by "header lines" within the file. Header lines will always begin with a '>' character followed by some descriptor. Second, an individual sequence can span more than one line and the number of lines does not have to be consistent throughout the file. For example in `sequences1.txt`, the sequence corresponding to gene2 spans two lines, while the sequences for gene1 and gene3 are on a single line. This will be an important consideration for you `create()` operation!

The Experiment ADT has the following 7 operations:

- `Experiment.create(filename)` Creates and returns a data structure to store the experimental data; here, we'll store the strings in a list. Each element of the list represents a DNA sequence found in the file with the given `filename`. This operation should also ensure only the allowed characters are present in the DNA sequence. If there are other characters in any of the strings, do not include that string in the list. For example:

```
sequences = Experiment.create('sequences1.txt')
```

- `Experiment.display(sequences)` Returns a string representing the data stored within the Experiment. The format of the string should be similar to the input file (see below). For this assignment, the descriptor in each header line does NOT have to match the original descriptor from the input file. Instead you can use sequential numerical labels that correspond to the data's position in the list. For example:

```
sequences = Experiment.create('sequences1.txt')
print(Experiment.display(sequences))
# >1
# ATGATGATGGCG
# >2
# GGCATATCCGGATACC
# >3
# TAGCTAGCCCGC
```

- `Experiment.numSequences(sequences)` Returns the number of sequences stored in the Experiment ADT. For example:

```
sequences = Experiment.create('sequences1.txt')
print(Experiment.numSequences(sequences))
# 3
```

- `Experiment.averageLength(sequences)` Returns the average length of the sequences stored in the Experiment ADT (as an **integer**). For example:

```
sequences = Experiment.create('sequences1.txt')
print(Experiment.averageLength(sequences))
# 13
```

- `Experiment.lengthDistribution(sequences)` Returns a dictionary that where the key-value pairs correspond to unique sequence lengths (the keys) and integer counts of the number of sequences with that length (the values), respectively. For example, in one of the files there are 2 sequences of length 12, and one sequence of length 16:

```
sequences = Experiment.create('sequences1.txt')
print(Experiment.lengthDistribution(sequences))
# {12: 2, 16: 1}
```

- `Experiment.averageGCcontent(sequences)` Calculates and returns the average GC content of the of the sequences stored in the Experiment ADT (as a **percentage**). GC content is simply a count of the 'G' and 'C' characters present in sequence. For example:

```
sequences = Experiment.create('sequences1.txt')
print('%.2f' % Experiment.averageGCcontent(sequences))
# 57.64
```

- `Experiment.removeLowQuality(sequences, minCutoff, maxCutoff)` Removes sequences stored in the Experiment ADT that have a GC content below `minCutoff` or above `maxCutoff`. This operation does **NOT** return anything. For example:

```
sequences = Experiment.create('sequences1.txt')
Experiment.removeLowQuality(sequences, 55, 65)
print(Experiment.display(sequences))
# >1
# GGCATATCCGGATACC
```

Testing

Write a test script that tests your implementation of the ADT, and ensures that all operations are correct. Think about the same kinds of issues:

- Black-box test cases.
- White-box test cases.
- Boundary test cases, and test case equivalence classes.
- Test coverage (percentage of functions tested)
- Unit vs. integration testing.

Running your test script on your correct ADT should report no errors, and should display nothing except the message `'*** Test script completed ***'`.

What to Hand In

- A Python program named `a3q3.py` containing the ADT operations (new and modified), but no other code.
- A Python script named `a3q3_testing.py` containing your test script.

Evaluation

- 2 marks: Your operation `create(filename)` is correct.
- 2 marks: Your operation `display(sequences)` is correct.
- 2 marks: Your operation `numSequences(sequences)` is correct.
- 2 marks: Your operation `averageLength(sequences)` is correct.
- 2 marks: Your operation `lengthDistribution(sequences)` is correct.
- 2 marks: Your operation `averageGCcontent(sequences)` is correct.
- 2 marks: Your operation `removeLowQuality(sequences, minCutoff, maxCutoff)` is correct.
- 14 marks: Your test script checks that the operations work correctly.

Question 4 (10 points):

Purpose: To practice using an ADT you built yourself in a relatively simple application.

Degree of Difficulty: Easy

We are now going to use your Experiment ADT to display information for a given sequencing file. Write a Python program that creates a new Experiment ADT and displays the following information:

- The number of valid sequences
- The average GC content
- The Average Sequence Length
- The Sequence Length Distribution

It should then use the `removeLowQuality(sequences, minCutoff, maxCutoff)` operation and then display all of the information above again. An example of the expected output when you run your program with the small example sequence file (`sequences1.txt`), a minCutoff of 55 and a maxCutoff of 65 is provided below. Note the `filename`, `minCutoff` value and `maxCutoff` value should be stored as variable at the beginning of your program.

DO QUESTION 3 FIRST! You are expected to import your previous question's Experiment ADT file as a module for use in this question. If your Experiment ADT works (which it should. You tested it, right?) this question is very easy.

```
# Initial Statistics ------------------------
#    Number of Sequences: 3
#    Average GC content: 57.64 %
#    Average Sequence Length: 13
#    Sequence Length Distribution:
#        Length: Number of Sequences
#        12: 2
#        16: 1
#
# Removing Sequences with GC content below 55 or above 65
# Updated Statistics ----------------------
#    Number of Sequences: 1
#    Average GC content: 56.25 %
#    Average Sequence Length: 16
#    Sequence Length Distribution:
#        Length: Number of Sequences
#        16: 1
```

Demonstrate your program on the sequence file `sequences2.txt` three times with different values for `minCutoff` and `maxCutoff` each time. Copy/paste the command-line output, as in the above example into your demo file. Consider this a demonstration, not testing. Because this application is pretty simple, and because you tested your Experiment ADT very thoroughly, you only need to run 3 demonstrations of your program working.

What to Hand In

- Hand in a well-written, suitably documented program named `a3q4.py`.
- Submit your demonstrations in a file named a3q4-demo.txt (PDF, TXT, RTF are all fine).

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your program makes appropriate use of the Experiment ADT, and is otherwise correct.
- 5 marks: Your demonstration shows your program working. If we run your program on the command line, we'll see the same results.

Source of Data

The data provided in `sequences2.txt` is a subset of a real sequencing data from the following paper: Zoe E Gillespie, Kimberly MacKay, Michelle Sander, Brett Trost, Wojciech Dawicki, Aruna Wickramarathna, John Gordon, Mark Eramian, Ian R Kill, Joanna M Bridger, Anthony Kusalik, Jennifer A Mitchell & Christopher H Eskiw (2015) Rapamycin reduces fibroblast proliferation without causing quiescence and induces STAT5A/B-mediated cytokine production, Nucleus, 6:6, 490-506, DOI: 10.1080/19491034.2015.1128610.

The authors have made this data publicly available at Gene Expression Omnibus; accession number GSE65145.