Lab 7 – CSE 101 (Spring 2019)

1. Objectives

The primary objectives of this lab assignment are:

- To understand time complexity with Python examples.
- Tutorial project on isearch and isort.
- To become more comfortable writing Python programs that feature Boolean operators, while loops, and lists.

2. Time complexity with Python examples

Visit the following link that explains understanding time complexity with Python examples. The graduate TAs will walk you through the article along with examples to clarify concepts of time complexity.

https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

3. Tutorial project

Go to page 110 of the book Explorations in Computing by John S. Conery. Start a new shell session on command prompt/terminal and type the following statement:

```
>>> from PythonLabs.IterationLab import *
```

Complete all the tasks T57 – T70 from the tutorial.

4. Write 3n + 1 Sequence String Generator function

Create a new Python file sequencegen.py and write the function sequence that takes one parameter: n, which is the initial value of a sequence of integers.

The `3n + 1` *sequence* starts with an integer, `n` in this case, wherein each successive integer of the sequence is calculated based on these rules:

1. If the current value of `n` is **odd**, then the next number in the sequence is three times the current number plus one. **Note:** Make sure to use integer division.
2. If the current value of `n` is **even**, then the next number in the sequence is half of the current number.
3. Integers in the sequence are generated based the two rules above until the current number becomes `1`.

Your function should start with an empty string and append characters to it as described next. The function keeps calculating the numbers in the `3n + 1` *sequence*, and while there are still numbers left in the sequence (meaning that the number has not reached `1` yet), it keeps appending letters to the string based on these rules:

1. If the number is divisible by `2`, then append 'A' to the string.

2. If the number is divisible by 3, then append 'B' to the string.
3. If the number is divisible by 5, then append 'C' to the string.
4. If the number is divisible by 7, then append 'D' to the string.

Note: The original number n also should contribute a letter (or letters) to the string. Also, append letters to your accumulating string for **all** conditions met above. That means, for example, if the current number is 6, the letters 'A' and 'B' will both be appended to the string, in that order. Apply the rules in the order given. Returning to the example for 6, your code must append the letters in the order 'AB', not 'BA'.

Note: If you find any helper functions useful, feel free to use them. In fact, I suggest that you try to define and use helper functions. This idea applies not only to this problem, but to any problem.

In the end, your function should return the string generated (accumulated) by the procedure described above. If n initially is less than 1, the function returns an empty string.

Example calls:

```
sequence(4) returns 'AA'
sequence(12) returns 'ABABBACCAAAA'
sequence(24) returns 'ABABABBACCAAAA'
```

Let's consider the second example above for a closer look:

```
 Sequence:| 12 | 6  | 3  | 10 | 5  | 16 | 8  | 4  | 2
----------+----+----+----+----+----+----+----+----+----
 Letters: | AB | AB | B  | AC | C  | A  | A  | A  | A
```

5. Submission

Submit completed sequencegen.py program on blackboard.