uOttawa

Assignment 4

Read the instructions below carefully. The instructions must be followed. This assignment is worth 6% of your grade. The assignment is due on Saturday 4th of April at 11:59 PM. No late assignments will be accepted.

The goal of this assignment is to learn and practice (via programming) the following concepts that we have learnt thus far: efficient algorithms and their analysis, design of custom made classes and their use in making objects and dictionaries and set, finally recursion (a way to solve problems that we will learn in the next couple of Online classes and labs).

This assignment has three parts. Each part explains what needs to be submitted. Put all those required documents into a folder called a4_xxxxxx. Zip that folder and submit the zip file as explained in Lab 1. In particular, the folder should have the following 6 files:

a4_part1_xxxxxx.py
a4_part1_xxxxxx.txt

and

a4_part2_xxxxxx.py

a4_part2_testing_xxxxxx.txt

and

a4_part3_xxxxxx.py
a4_part3_xxxxxx.txt

For every function and method that you design in this assignment you must include the type contract inside a docstring (the rest of the usual info that goes into the docstring can be omitted for this assignment)

As always, your programs must run without syntax errors.

No global variables are allowed in functions: only use variables that are in the function's namespace. That is, only use variables that are parameters or are otherwise added to the function's namespace such as through assignment or iteration

****
Big (O)1:
(30 points)
****

For this part, you will be solving some problems and implementing the solutions. For this part, your goal is to design and implement solutions that are as fast as possible, and to analyze these solutions as well. You will submit two files:

A4_part1_xxxxxx.py needs to contain the functions you are asked to write for each question.

A4_part1_xxxxxx.txt needs to contain, for each function, a

rough analysis of its running time in terms of n (where n is the length of the input list a, i.e. n=len(a)) and where n=10,000. For example, if your function looks like this:

```
def riddle(a):
        s=0
        for x in a:
            for y in a:
                s = s+ x*y
        return s
```

Then you would write: This function has two nested for loops. The inner loop executes 10,000 times for each step of the outer loop, which also executes 10,000 times. Therefore, this function performs roughly 10,000*10,000=100,000,000 operations. If your function uses pyton's `sort()` method or pyton's `sorted()` function to sort the list a, just say that that function call does about 140,000 operations (since pyton's sort does roughly $n \log_2 n$ operations).

Alternatively, if you prefer, you can write your analysis in terms of general n. Thus for the above function riddle, one would say that its running time is O(n^2).

In all of the questions below you may assume that a is a list containing numbers.

1a) (5 points) Write a function, largest_34(a), that returns the sum of the 3rd and 4th largest values in the list a. For simplicity, you may assume that the numbers in the list a

are all distinct and that the list a has at least 4 elements. For example:

```
>>> largest_34([1000, 1, 100, 2, 99, 200, -100])
199
```

1b) (5 points) Write a function, largest_third(a), that computes the sum of the `len(a)//3` of the largest values in the list a. For simplicity, you may assume that the numbers in the list a are all distinct and that the list a has at least 3 elements.

1c) (5 points) Write a function, third_at_least(a), that returns a value in a that occurs at least `len(a)//3 + 1` times. If no such element exists in a, then this function returns `None`.

1d) (15 points) Write a function, sum_tri(a,x), that takes a list , a, as input and returns `True` if there exists indices i, j and k (where i and j and k are not necessarily distinct) such that a[i]+a[j]+a[k]=x. Otherwise it returns `False`. For example, if a=[1, 5, 8, 2, 6, 55, 90] and x=103, then sum_tri(a, x) would return True since a[1]+a[2]+a[6]=5+8+90=103. If a=[-1, 1, 5, 8, 2, 6] and x=-3, sum_tri(a, x) would return True since a[0]+a[0]+a[0]=-1+ -1 + -1 = -3. If a=[-10,2] and x=-18, sum_tri(a, x) would return True since a[0]+a[0]+a[1]=-10+-10+2=18. If a=[1, 1, 5, 8, 2, 6] and x=1000 would return False, since there are not indices i, j and k such that a[i]+a[j]+a[k]=1000.

****

PART 2 **Object-oriented programming** (40 points) 10 each

****

1) Write a class Rectangle that represents a rectangular two-dimensional region. Your Rectangle objects should have the following methods:

Method	Description
`def __init__(self,x,y,w,h)`	Initializes a new Rectangle whose top-left corner is specified by the given (*x*, *y*) coordinates and with the given width and height, w by h. Raise a ValueError on a - negative width or height.
`def height(self)`	A property representing the rectangle's height.
`def width(self)`	A property representing the rectangle's width.
`def x(self)`	A property representing the rectangle's *x*-coordinate.
`def y(self)`	A property representing the rectangle's *y*-coordinate.
`def __str__(self)`	Returns a string representation of this Rectangle, such as "Rectangle[x=1,y=2,width=3,height=4]".

2) Add the following accessor method to your Rectangle class: def contains(self, x, y). Your method

should return True if the given coordinates lie inside the bounds of this Rectangle.

3) Add the following method to your Rectangle class: def union(self,rect) Your method should accept another Rectangle as a parameter, and return a new Rectangle that represents the area occupied by the tightest bounding box that contains both the current Rectangle (self) and the given other Rectangle.

4) Add the following method to your Rectangle class: def intersection(self,rect) Your method should accept another Rectangle as a parameter, and return a new Rectangle that represents the largest rectangular region completely contained within both the current Rectangle (self) and the given other Rectangle. If the rectangles do not intersect at all, returns a Rectangle with its width and height both equal to 0.

(Bonus)

5) Add the following method to your Rectangle class: def__eq__(self,rect) Your method should accept another Rectangle as a parameter and return True if the two rectangles have exactly the same state, including their x, y, width, and height values.

Note: you have to test each your class and methods by giving examples of your calls and counter calls

****

PART 3: **Sets, Dictionaries, and Recursion** (30 points)

****

1) (10 points) Write a function called **overlap** that takes a set of integers and a list of integers as parameters and that returns a new set containing values that appear in both structures. For example, given set and list:

```
set1: {0, 19, 8, 9, 12, 13, 14, 15}
list1: [0, 19, 2, 4, 5, 9, 10, 11]
```

If the following call is made:

```
overlap(set1, list1)
```

the function would return:

```
{0, 19, 9}
```

You are not allowed to construct any structures, besides the set you will return, to solve this problem. You may not alter the passed in list or set. **You may not convert the list to a set**.

2) (Bonus) Write a function called **reverse** that accepts a dictionary from strings to strings as a parameter and returns a new dictionary that is the reverse of the original. The reverse of a dictionary is a new dictionary that uses the values from the original as its keys and the keys from the original as its values. Since a dictionary's values need not be unique but its keys must be, you should have each value map to a set of keys. For example, if passed the following dictionary:

```
{42: "Marty", 81: "Sue", 17: "Ed", 31:
```

```
"Dave", 56: "Ed", 3: "Marty", 29: "Ed"}
```

You should reverse it to become (the order of the keys and values does not matter):

```
{"Marty": [42, 3], "Sue": [81], "Ed": [17, 56, 29], "Dave": [31]}
```

3) (20 Points) Implement a recursive python function digit_sum(n) to calculate the sum of all digits of the given integer n. Then, implement a recursive python function to compute the digital root digital_root(n) of the given integer n. Your function must use the digit_sum function.The *digital root* of a number is calculated by taking the sum of all of the digits in a number, and repeating the process with the resulting sum until only a single digit remains. For example, if you start with 1969, you must first add 1+9+6+9 to get 25. Since the value 25 has more than a single digit, you must repeat the operation to obtain 7 as a final answer.Your function digital_root(n) must use digit_sum function. Place all of your 4 functions in your part3 file.

Note: You can implement iterative versions of the two functions for yourself, but submit the recursive versions only. Your recursive functions must not use loops.