Final

Graded

Student

Colin Cano

Total Points

7 / 30 pts

Question 1

smooth(L, k)

3 / 4 pts

- **– 0 pts** Correct
- **– 1 pt** Syntax error
- **– 1 pt** Indexing error
- **– 2 pts** Poorly structured iteration
- **– 3 pts** Fails to modify L
- ✔ **– 1 pt** Modifies L but fails to shorten L by (k-1) elements
  3
- **– 4 pts** No answer or answer fails to meet specification

Question 2

int2rom(N)

1 / 4 pts

- **– 0 pts** Correct
- **– 1 pt** Syntax error
- **– 1 pt** Missing or incorrect base case
- **– 2 pts** Missing or incorrect recursive step
- ✔ **– 3 pts** Poorly structured recursion/iteration
- **– 1 pt** Iteration fails to construct and return string
- **– 2 pts** Poorly structured iteration fails to ensure descending order of characters in result
- **– 4 pts** No answer or answer fails to meet specification

2 n is a int, cant slice an int, Your 2 if checks are also wrong

Question 3

Counting substrings **0** / 6 pts

3.1 **countSubstrings(S, start, end)** **0** / 4 pts

- **0 pts** Correct

- **1 pt** Syntax error

- **1 pt** Fails to handle start==end case correctly

- **2 pts** Poorly structured iteration

✔ - **4 pts** No answer or fails to meet specification

1 I do not understand how your code matches the specification. It is vastly different from the intended solution

- **1 pt** No return statement

- **1 pt** poor function call

3.2 **runtime** **0** / 1 pt

- **0 pts** Correct

✔ - **1 pt** Incorrect run time

3.3 **justification** **0** / 1 pt

- **0 pts** Correct

✔ - **1 pt** Poor or incorrect run time argument

Question 4

countMatches(S, target) **1** / 5 pts

- **0 pts** Correct

✔ - **1 pt** Syntax error

- **1 pt** Missing or incorrect base case

✔ - **1 pt** Fails to account for within-string matches

- **2 pts** Missing or incorrect recursive step

✔ - **2 pts** Fails to handle all required data types

- **3 pts** Poorly structured recursion

- **5 pts** No answer or fails to meet specification

Question 5

Multisets **2** / 11 pts

5.1 __len__(self) 0 / 1 pt

– 0 pts Correct

✔ **– 1 pt** Incorrect

We were looking for:
return(sum(self.D.values()))

self.D was a dictionary that contains keys for the number of times an item appears in the multiset.

5.2 __repr__(self) 0 / 2 pts

– 0 pts Correct

– 1 pt Syntax error

– 1 pt Fails to provide matching {{...}}

✔ **– 2 pts** No answer or fails to meet specification

– 2 pts Blank

'{{' and '}}' were required to be in at the start and end of the answer.

For those that did have it, you also were missing too much with issues like no call for __repr__ to help get the lists representation.

Another common mistake was poor indexing. It was required to iterate through the keys of self.D again for this problem.

Lastly, L is NOT a given variable. It was a parameter only to be used in our flatten function, which was a helper function that did not need anything else added to it. Some people used L as a typical variable throughout the problem.

5.3 __contains__(self) 0 / 2 pts

– 0 pts Correct

– 1 pt Syntax error

✔ **– 2 pts** Fails to meet specification

– 1 pt Missing Ghost Keys Check

– 2 pts No Answer

Common Misses with this problem

"Trying to iterate through self. It needed to be self.D, as D is our dictionary within the Multiset class. While self is simply a reference to the class itself. So self can only call functions within the class, but not variables within those functions."
"Not checking if self.D[element] > 0. That was the ghost keys the question was talking about."
"Unknown Syntax was pretty common as well. A lot of guesses for how to access the dictionary. With others even treating it list a list or string instead."

5.4 **union(self, other)** **1** / 3 pts

- **− 0 pts** Correct
- **− 1 pt** Syntax error
- **− 1 pt** Fails to use self.copy() or mset() for new structure
- ✔ **− 1 pt** Failure to sum properly
- ✔ **− 1 pt** Fails to produce correct union (beware zero elements)
- **− 3 pts** No answer or fails to meet specification

5.5 **symmetric_difference(self, other)** **1** / 3 pts

- **− 0 pts** Correct
- **− 1 pt** Syntax error
- **− 1 pt** Fails to use self.copy() or mset() for new structure
- ✔ **− 2 pts** Fails to produce correct symmetric difference (beware zero elements and min/max constraints)
- **− 3 pts** No answer or fails to meet specification
- **− 1 pt** Missing boundary check

Seat: U6 Name: Colin Cano

CS1210 Computer Science I: Foundations
Final Exam
Friday, December 20, 2024

This test consists of 5 problems worth a total of 30 points. **Read each problem carefully.** Don't get stuck too long on any one question: you have 120 minutes. The last problem is a larger implementation which you are being asked to complete; it involves some reading, but the coding component is not itself overly long. Finally, please **write legibly**, and, where code indentation/alignment matter, **line things up cleanly and unambiguously!** Points may be deducted for sloppy syntax (*e.g.*, missing quote marks or commas) or alignment (*e.g.*, unclear scope for conditionals or iteration).

1. Smoothing a Sequence of Numbers [4 points]

Specification: smooth(L, k) takes a list of numeric values (int or float) and a smoothing parameter, k, where 1 <= k < len(L), and modifies L, "smoothing" the original values in L by replacing each with the average of the k elements starting in that corresponding position. So:

```
>>> L=[3, 4, 2, 6, 1, 8]
>>> smooth(L, 3)
>>> L
[3.0, 4.0, 3.0, 5.0]      # Note (3+4+2)/3 = 3.0
```

9/4 14 / 24/6 = 4

You will recognize this problem as very similar to QotD10, but instead of returning a new list you are to modify the existing list in place. Note that when you "smooth" a list, the result will necessarily be shorter by k-1 elements.

You may **use whichever approach you feel most appropriate**; your solution will be scored based on correctness, efficiency and elegance.

```
def smooth(L):
    for i in range(len(L)-(k-1)):
        L[:]=(sum(L[i:3]))/K          #replaces value
                                      # with smoothed
                                      # valee
```

2. Converting from Integer to Roman Numeral [4 points]

Specification: int2rom(N) takes a positive int, N, and returns a legal Roman numeral consisting of a string containing a sequence of upper case characters representing integer values. You'll recognize this problem as the inverse of QotD20, where you were asked to convert a Roman numeral to an integer.

Recall a Roman numeral is a string read from left to right. Each character represents a specific value:

M=1000, D=500, C=100, L=50, X=10, V=5 and I=1

If sorted from larger to smaller, a string of these is interpreted as the sum of those elements. So:

MDCLXXXV = 1000+500+100+50+10+10+10+5 = 1685

Thus Roman numeral is legal if it is mostly arranged from largest valued character to smallest, with the exception of single-small-large-character combinations at the end of each substring of identical "large" characters. Thus "XXIXV" is legal, but "XIXXV" and "XIXIXV" are not; similarly 1900 is written "MCM" but 1990 is written "MCMXC" and not "MXM". The restrictions on which small-large combinations are legal are defined explicitly by the default mapping provided. I would recommend, but do not require, a recursive approach.

```
def int2rom(N, map=((1000, 'M'), (900, 'CM'), (500,'D'), (400, 'CD'),
                    (100,'C'), (90, 'XC'), (50,'L'), (40, 'XL'),
                    (10,'X'), (9, 'IX'), (5,'V'), (4, 'IV'), (1, 'I'))
    value = {1000:'M', 900:'CM', 500:'D', 400:'CD', 100:'C',
             90:'XC', 50:'L', 40:'XL', 10:'X', 9:'IX', 5:'V',
             4:'IV', 1:'I'}
#base case
    if len(N) == 0:
        return''
    if len(N) > 1 and N[:2] in value:        #checks for combos
        return [value[:2]] + int2rom(N[2:])
    else:
        return [value[0]] + int2rom(N[1:])
```

\

3. Counting Substrings [6 points]

Specification: countSubstrings(S, start, end) takes a string, S, consisting of only lower case letters, and two additional lower-case letters (denoted start and end) and returns the integer count of all the substrings of S delimited by start and end. Your solution should work correctly even if start and end are the same character. So, for example, countSubstrings('abcdcbaafb', 'a', 'b') returns 5 ('**AB**cdcbaafb', '**ABCDCB**aafb', '**ABCDCBAAFB**', 'abcdcb**AAFB**', and 'abcdcba**AFB**') and countSubstrings('abcdcbaafb', 'c', 'c') returns 3 ('ab**C**dcbaafb', 'ab**CDC**baafb', and 'abcd**C**baafb').

There are multiple solutions, one of which is optimal, meaning there is no other with better run time. Full credit will be awarded for this solution, although any correct solution will also earn nearly full credit.

```
def countSubstrings(S, start, end):
    count = 0
    temp = [ ]
    for i in S:                      # O(N)
        if i == target:
            temp.append(target)
            count += len(temp)
            temp.pop(target)
        elif i == end:
            temp.append(end)
            count += len(temp)
            temp.pop(end)
    return count
```

Assuming S has N characters, what is the algorithmic complexity of your solution?

$O(\log N)$ _____ $O(N)$ _____ $O(N \log N)$ _____ $O(N^2)$ X $O(2^N)$ _____

Explain your reasoning (2-3 sentences, max):

Since there is a for loop and if, it would add up to be $O(N^2)$

4. Counting Matches [5 points]

Specification: Write a function countMatches(S, target) that returns the number of times the target argument appears in S, where S is composed of lists, tuples, ranges, strings, sets, dictionaries (both keys and values) as well as integers, Booleans, and floats. So (emphasis added to show matches):

```
>>> countMatches({(1, (2, 1), range(4)), 'cs1210', 1, ((1,))}, 1)
5
>>> countMatches([('get back jack'), range(9), {'track':'hack'}], 'ack')
4
```

A few important considerations. First, the type of the target matters: the integer 456 will not match the string '123456789' although though the string '456' does. Conversely, the string '456' does not match the integer 456. Also, because the structure can be nested arbitrarily deeply, you should probably consider a recursive solution.

```
def countMatches(S, target):
    if len(S) == 0:        #base case
        return 0
    if target in S[0]:
        return(S.count(target) + countMatches(S[1:], target)
    else:
        return countMatches(S[1:], target)  #removes from list
```

#returns count of how many times target is in S[0]

5. Multisets [11 points]

Here, we will develop a new class for "multisets" called mset. A multiset is exactly like a set with one important difference: it can contain multiple copies of the same object, while still implementing the usual-theoretic operations (*e.g.*, union, intersection and set difference) and maintaining as much of the set class' performance as possible (*i.e.*, O(1) membership operations). Here's an example of the behavior we are trying to capture, where the double curly brackets visually indicate these are multisets.

```
>>> x = mset((1, 1, 1, 2, 3, 3))              # Constructor
>>> y = mset((2, 2, 2, 3, 3, 4, 4, 5))        # Constructor
>>> len(x), len(y)                             # Number of elements
(6, 8)
>>> 2 in x                                     # Membership
True
>>> x | y                                      # Union
{{2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 1, 1, 1}}
>>> y & x                                      # Intersection
{{2, 3, 3}}
>>> x - y                                      # Set difference
{{1, 1, 1}}
>>> x ^ y                                      # Symmetric set difference
{{1, 1, 1, 2, 2, 4, 4, 5}}
```

Implementing this set explores some new Python ground that we did not cover in class, so I will explain as we go and only ask you to implement the portions that fit what we've covered.

Our new mset class represents multisets as dictionaries, where the value represents the number of times the key occurs in the multiset.

```
class mset():
    def __init__(self, S=()):                  # Invoked by constructor
        self.D = dict(zip(set(S),[0]*len(set(S))))
        for e in S:
            self.D[e] += 1
```

The first method we implement, copy(), returns a new copy of the current multiset (this will be useful later).

```
    def copy(self):
        new = mset(())                          # Create an empty multiset
        new.D = self.D.copy()                   # Copy counts from self.D
        return(new)                             # Return the new multiset
```

The mset() class, like many other objects in Python, has a collection of what are called "dunder" or "magic" methods, which are the internal double-underscore methods, like __len__(), that integrate objects into the Python language (__len__() is invoked by the len() function):

```
    def __len__(self):                          # Implements len()
        return (len(self)
```

Another dunder method is the __repr__() method, which returns the string corresponding to the current multiset's current value (*e.g.*, {{2, 3, 3}} for x & y above). Here, I provide a helper function flatten(), which takes a list of lists and returns a flattened version of the list, which will be useful as you complete the __repr__() method so that it behaves as shown above.

```
def __repr__(self):                    # Returns string for printing
    def flatten(L):  # Converts [[1, 1], [2], [3, 3, 3]] to [1, 1, 2, 3, 3, 3]
        if len(L) == 0:
            return([])
        return(L[0] + flatten(L[1:]))
    return(set(L))
```

The membership operator "in" is implemented by __contains___(self, element), which returns a Boolean (keep in mind that operations on mulitsets can leave extraneous "ghost" keys in self.D with 0 values).

```
def __contains__(self, element):       # Membership: is element in self?
    return(element in self)
```

The union, intersection, set difference, and symmetric set difference methods take one argument (a multiset) and are invoked by their corresponding dunder methods:

```
def __or__(self, other):               # Implements m1 | m2
    return(self.union(other))
def __and__(self, other):              # Implements m1 & m2
    return(self.intersection(other))
def __sub__(self, other):              # Implements m1 - m2
    return(self.difference(other))
def __xor__(self, other):              # Implements m1 ^ m2
    return(self.symmetric_difference(other))
```

Using intersection and set difference as models, implement union and symmetric set difference.

```
def intersection(self, other):
    result = mset(())                  # Create a new multiset
    for k in self.D:                   # Compute intersection
        if k in other.D:               # ...see note above about ghost values!
            result.D[k]=min(self.D[k], other.D[k])
    return(result)                     # Return new multiset
```

```
def difference(self, other):
    result = self.copy()             # Make a copy of this multiset
    for k in other.D:                # Compute set difference
        if k in result.D:
            result.D[k] -= min(result.D[k], other.D[k])
    return(result)                   # Return the altered copy

def union(self, other):
    result = Mset([])
    for k in self.D:
        if k in other.D:
            result.D[k]+= self.D[k] + other.D[k]
    return result

def symmetric_difference(self, other):
    result = self.copy()
    for k in other.D:
        if k in result.D:
            result.D[k]+= max(result.D[k], other.D[k])
    return result
```

...intentionally left blank...