Name: ________________________________

22c:016 Computer Science I: Foundations

Final Exam

Monday, December 15, 2014

Place a checkmark by your section:

____	Section A01 Richa Gandhi 8:30 Tues	____	Section A04 Ruchika Salunke 11:00 Tues	____	Section A07 Momina Tabish 5:00 Tues
____	Section A02 Ruchika Salunke 9:20 Tues	____	Section A05 Kyle Diederich 12:30 Tues	____	Section A08 Richa Gandhi 11:00 Tues
____	Section A03 Momina Tabish 11:00 Tues	____	Section A06 Kyle Diederich 2:00 Tues	____	Section A09 Thamer Al Sulaiman 2:00 Tues

Don’t forget to put your name at the top of the exam!

This test consists of 3 questions, some requiring multiple answers, worth a total of 85 points. Because they are of varying difficulty, it is probably good strategy not to get stuck too long on any one question. You have 120 minutes: turn in your cheat sheet and any scratch paper you used with your copy of the exam.

Don’t forget to **write legibly**, and remember that **indentation matters**. Illegible or improperly indented answers will be marked as wrong.

Have a good winter break!

1. Grade Calculation (26 points)

Assume you are given a dictionary that represents grades given to students in a class. The keys to the dictionary represent the student identifiers, while the values are themselves dictionaries containing grades for each element of the class, 3 *homeworks* (30 points each), 2 *exams* (100 points each), 3 *quizzes* (10 points each) and 2 *projects* (20 points each). For example:

```
{ 112233:{'qz':[8,10,5], 'hw':[16,27,25], 'ex':[92,83], 'pr':[18,14]}, ... }
```

Write a function that scans the dictionary and computes a final grade for each student. The final grade should be added to each indexed dictionary as a new key/value pair (use 'fin' as the key, and make the value a one element list). To compute the final grade, you will need to normalize the sum of the scores for each component ('hw', 'qz', 'ex', and 'pr') so that each component is worth 25% of the final grade (remember to drop the lowest quiz grade). Also, please note that the instructor has promised to drop the lowest quiz score. **[8 points]**

```
def computeFinal(D):
 for id in D.keys():
  D[id]['fin'] = ((sum(D[id]['hw'])*25)/90 +
                  (sum(D[id]['ex'])*25/200) +
                  (sum(D[id]['pr'])*25)/40 +
                  (sum(sorted(D[id]['qz'])[1:])*25)/20)
```

Note: extra line breaks are ok inside a parenthesized expression

____

____

____

____

____

____

____

____

____

____

____

Not entirely satisfied with just computing the final grade, you'd like to obtain some simple statistical information. Write a function that takes a component and an index and returns the class min, max and mean grades for that grade item (*i.e.*, $stats(D,'hw',2)$ would return the min, max and mean grades for the 3rd homework item, and $stats(D,'fin',0)$ would return the min, max and mean final grades). **[8 points]**

```
def stats(D, key, index):
  L = [ D[student][key][index] for student in D.keys() ]
  try:
    return((min(L), max(L), sum(L)/len(L)))
  except:
    return((0,0,0))      # for L=[]
```

Finally, you realize you'd like to be able to plot student scores as a histogram. Write a function that "inverts" a specified component and index, producing an ordered list of tuples, where each tuple consists of two integers, one a value between min and max for this grade item, and the other the number of people with those scores (round floating point scores to the lower integer). In other words, $histogram(D, 'qz', 0)$ might well return a value that looks like:

```
[ (2, 12), (3, 14), (4, 0), (5, 0), (6, 42), (7, 35), (8, 19) ]
```

where min and max for this grade item are 2 and 8, respectively. **[10 points]**

```
def histogram(D, key, index):
    H = {}
    for id in D.keys():
        try:
            H[int(D[id][key][index])] = H[int(D[id][key][index])]+1
        except:
            H[int(D[id][key][index])] = 1
    for score in range(min(H.keys()), max(H.keys())+1):
        try:
            H[score]
        except:
            H[score] = 0
    return([ (score, H[score]) for score in range(min(H.keys()), max(H.keys())+1) ])
```

Note: other solutions are possible; the key is to handle the case where the entry is not in H currently. Using try/except is but one solution, as one can always check explicitly instead.

2. Stemming and Regular Expressions (8 points)

Intrigued by Project 2, you resolve to modify your code over the holiday and produce a better stemmer. You find a website that describes *snowball*, a language specifically designed to support the development of word stemmers in a variety of languages. There you read about the Porter stemmer, and decide to implement your own in Python.

The Porter stemmer relies on finding two special regions in each word, designated R1 and R2. The definitions of these regions are as follows:

> R1 is the (possibly null) region after the first non-vowel following a vowel, or is the null region at the end of the word if there is no such non-vowel.

and

> R2 is the (possibly null) region after the first non-vowel following a vowel in R1, or is the null region at the end of the word if there is no such non-vowel.

For English, the vowels are defined as 'aeiouy' and the consonants are everything else. So, for example:

```
b e a u t i f u l
         |-------| R1
             |---| R2

a c h i e v e m e n t
   |-----------------| R1
           |---------| R2

s u b v e r s i v e
     |-------------| R1
           |-------| R2

s p r i n k l e d
         |-------| R1
                || R2 (null)
```

Both R1 and R2 can be identified with a single regular expression. R1 is found by matching that regular expression against the whole word, while R2 is found by matching the same regular expression against R1. What is the regular expression? (Note: partial credit will not be awarded without an explanation of each element of the expression).

re.match('^[^aeiouy]*[aeiouy]+[^aeiouy](.*)$', string)
start of word — 0 or more consonants — 1 or more vowels — 1 consonant — rest of word

3. Implementing Sets (51 points)

We are implementing a new set data structure object called MySet. Internally, MySet is just a list of unique items that are maintained in sorted order, a design decision which makes the implementation more efficient than just keeping a list of elements. The basic class definition looks like:

```
defclass MySet ():
    def __init__(self):
        self.L = []
```

The new object has a single *attribute*, *L*, which contains a list of elements. **For now, we will assume that the contents of the list will always be integers.**

```
>>> M = MySet()
>>> M.L                          # value of private attribute L
[]
```

The *L* attribute should be considered private, and all access to the object will be through four methods: *check*(), *member*(), *insert*(), and *delete*().

The *check*() method should return True if the object's internal list *L* consists of correctly sorted unique elements, and False otherwise.

```
>>> M.check()
True
```

Your solution shouldn't take more than a few lines of code. **[6 points]**

```
def check(self):
 for i in range(len(self.L)-1):
  if self.L[i] > self.L[i+1]:
   return(False)
 return(True)
```

____

____

____

____

Most of the remaining methods share the need to find elements in the set, or at least determine where the element would be if it were in the ordered set. These methods will make use of an efficient, private, "helper" method __*find*__() that returns two values. The first value, a Boolean, indicates whether the element is in the ordered set, while the second value, an integer, tells you the index of the element (if the first value is True) or the index corresponding to where the element would have been found (if the first value is False).

```
>>> M.L             # assume some elements have been inserted...
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> M.__find__(7)
(True, 7)
>>> M.__find__(-1)
(False, 0)
>>> M.__find__(12)
(False, 10)
```

Remember, because our data are a set, there will never be duplicate items in the internal list. We'll implement __*find*__() in a minute, but for now let's focus on the other methods assuming we have a working (and efficient!) __*find*__().

The *member*() method searches the ordered set and returns True if the requested *element* is in the set and False otherwise.

```
>>> M.member(4)
True
>>> M.member(-1)
False
```

Your implementation should use __*find*__(); again, just a few lines of code. **[6 points]**

```
def member(self, element):
  return(self.__find__(element)[0])
```

____

____

____

____

____

____

The *insert*() method takes a new *element* and inserts it into the internal list; it should return True on success and False on failure, such as when *element* is already a member of the set. This method is the only means you have of adding new elements to the set.

```
>>> M.insert(7)
False
>>> M.insert(10)
True
```

Your implementation should again use __*find*__(). **[6 points]**

```
def insert(self, element):
  if self.L == []:
    self.L = [element]
    return(True)
  result = self.__find__(element)
  if not result:
    self.L=self.L[:result[1]] + [element] + self.L[result[1]:]
    return(True)
  return(False)
```

The *delete*() method removes *element* from the set; it should return True on success (*i.e.*, the element was removed) and False on failure, such as when *element* was not in the set in the first place.

```
>>> M.delete(3)
True
>>> M.delete(3)
False
```

Your implementation should use *find*(). **[6 points]**

```
def delete(self, element):
    result=self.__find__(element)
    if result[0]:
        self.L=self.L[:result[1]] + self.L[result[1]+1:]
        return(True)
    return(False)
```

We next turn our attention to an efficient __*find*__() method. We'll implement *find*() as a recursive function and take advantage of the fact that the elements are maintained in sorted order: the *lo* and *hi* parameters are used to focus on different segments of the internal list. The default values for *lo* and *hi* ensure find starts by considering the entire list by default. **[8 points]**

```
def __find__(self, element, lo=0, hi=None):
  # Can't use len(self.L) as hi default value because of scoping
  if hi == None:
     hi = len(self.L)
  if lo == hi:
    return(False, lo)
  mid = (lo+hi)//2
  if self.L[mid] == element:
     return(True, mid)
  elif self.L[mid] > element:
     return(self.__find__(element, lo, mid))
  return(self.__find__(element, mid+1, hi))
```

Your friend suggests adding an attribute *reverse* to the object that governs whether the internal list *L* is kept in ascending or descending order. Given that you are using __*find*__() to support all the set access functions, and that you are not going to look at private attribute *L*, do you think this suggestion makes sense? Explain briefly. **[5 points]**

No, it doesn't matter which order the list is sorted. The only concern is that the list be sorted so that we may use binary search rather than having to scan the entire list each time we want to __*find*__() and element.

Finished with your code, you test it out and are delighted to note that it also supports a mix of integer and floating point:

```
>>> M.insert(8.5)
True
>>> M.L
[0, 1, 2, 4, 5, 6, 7, 8, 8.5, 9, 10]
```

and that it even works for, *e.g.*, tuples, although it fails for mixed types:

```
>>> M = MySet([(1, 'bar'), (7, 'baz'), (3, 'foo')])
>>> M.insert((2, 'boo'))
True
>>> M.L
[(1, 'bar'), (2, 'boo'), (3, 'foo'), (7, 'baz')]
>>> M.insert(4)
Traceback (most recent call last):
    ...some text removed here...
TypeError: unorderable types: tuple() > int()
```

You therefore realize that items in the ordered set must be of the same type, and the first element added establishes the type of the entries allowed. Which methods (*__init__*(), *__find__*(), *insert*(), *delete*(), *member*(), and/or *check*()) need to be modified to avoid errors like the one just shown (*e.g.*, by rejecting an unsuitable insert while printing out an appropriate message)? Choose any one of the methods that needs to be changed and describe the simplest way to address this issue. **[5 points]**

Since all the methods that need to be changed, *insert*()**,** *delete*() **and** *member*() **use** *__find__*()**, one solution is to change** *__find__*() **to check type of its argument against the type of the first member of self.L if self.L is not []. Neither** *__init__*() **nor** *check*() **need to be changed.**

Finally, annoyed with the complexity of the implementation, you convince yourself that using just a regular list instead of an ordered list would work as well. Your friend, a CS major, points out that using unordered lists to implement sets with *check*(), *member*(), *insert*() and *delete*() methods would not be as efficient. Explain why this is the case, which of these four methods would be affected. What is the difference in efficiency in order-of notation? (Hint: how would you implement __*find*__() if the private attribute L were not ordered?) **[5 points]**

member()**,** *insert*() **and** *delete*() **would be** $O(N)$ **rather than** $O(logN)$**, since we would have to scan the list each time when searching for an element in** __*find*__()**, while the obvious implementation of** *check*() **on an unordered list is** $O(N^2)$**.**

At this point, you're ready to throw out your code and use Python's built in set data structure, which is essentially based on dictionaries with keys but no values. Is this a good idea? Give two good reasons to support your answer. **[4 points]**

Dictionaries are even faster than our binary search versions of *member*()**,** *insert*()**, and** *delete*() **and they automatically enforce uniqueness, which we check with** *check*()**. Also, they support mixed types.**