CS:1210 Final Exam

May 15th, 5:30 pm to 7:30 pm

Instructions: (i) This is an open notes exam and you have 2 hours to complete it. There are 4 problems in the exam and these appear on 9 pages. (ii) Make sure that you do not have any electronic devices (laptops, phones, calculators, dictionaries, etc.) on your desk; you are not allowed to access these during your exam. (iii) Write as neatly as you can. (iv) Show all your work, especially if you want to receive partial credit.

Name:

Circle your discussion section:

SCA (M, W evening)	A01 (8:30-9:20)	A02 (11-11:50 MH)	A03 (11-11:50 SH)
	A04 (12:30-1:20)	A05 (2:00-2:50)	A06 (5:00-5:50)

1. (**40 points**) On *Lists and Strings.*

   (a) (**10 points**) Suppose that `L = ["It is", "celebration", "signal versus noise", "Information technology", "candidates", "typical symptoms"]`. Evaluate each expression and write down its *value*.

   (a) `[x.split() for x in L if " " in x]`

   (b) `[x[:4] for x in L[2:4]]`

   (c) `"-".join(L[2].split())`

   (d) `[len(x.split()) for x in L].count(2)`

   (e) `[x for x in L for y in x if y in ["v", "y"]]`

(b) (**10 points**) Given below is a partially completed function that takes as parameter a list of word-frequency tuples. For example, the list sent into this function might be `[("ok", 4), ("as", 3), ("hill", 6), ("ok", 11), ("zoo", 4)]`. The function processes the list and combines the frequencies of words that occur multiple times and returns a list of word-frequency tuples with no duplicate words. For example, for the parameter given above, the function returns `[("as", 3), ("hill", 6), ("ok", 15), ("zoo", 4)]`. The ordering of the tuples in the returned list does not matter. There are 4 blanks in the code below.

```
def combineDuplicates(L):
    L.sort() # sort L so that tuples with same word are together
    newL = [] # create a new list that will be returned

    # first word and its frequency are saved; as we encounter
    # more copies of this word, freqSum will be updated
    newWord = L[0][0]
    freqSum = L[0][1]

    # process the remaining words
    for i in range(1, len(L)):
        # if current word is different from previous word

        if ___________________________________________ :

            # append previous word and its frequency sum to newL
            newL.append((newWord, freqSum))

            # save current word and its frequency

            newWord = ____________________

            freqSum = ____________________

        # if current word is same as previous word
        # update the frequency sum
        else:

            __________________________________

    # process the last word and frequency
    newL.append((newWord, freqSum))

    return newL
```

(c) (**20 points**) Your task is to write a program that reads a text file called `test.txt` and outputs the frequencies of upper case letters in the text file. For example, if `test.txt` contains

```
AbcBAD
CAb
CDC
testing
```

then the first five lines of your program's output should be

```
A 3
B 1
C 3
D 2
E 0
```

The remaining 21 output lines contain the remaining upper case letters with 0 as their frequency. My code for this is 10 lines long; I use the built-in Python functions `ord` and `chr`.

2. (**40 points**) On *dictionaries.*

(a) (**10 points**) Suppose that `D` is the dictionary `D = {"class":[7, 0, 1], "thing":[3, 1, 4], "big": [1, 1, 9]}`. Write down the value of `D` after each of the following Python statements. Evaluate each statement starting with the same value of the dictionary `D`, shown above. Not all of the statements are guaranteed to evaluate correctly. So if you think a statement will cause an error indicate this and briefly explain why an error is caused.

   (a) `D["big"][2] = "big"`

   (b) `D[D["big"][2]] = "big"`

   (c) `D.update({"thing":[], "ok":[2]})`

   (d) `D[[3, 1, 9]] = "extra"`

   (e) `D[tuple("o k".split())] = 11`

(b) (**15 points**) Let `wordNetwork` be the dictionary that represents the *word network* that we constructed in implementing a solution to the word ladder game problem. Thus every legal 5-letter word is a key in `wordNetwork` and for any legal 5-letter word `w`, `wordNetwork[w]` is a list consisting of all legal 5-letter words that can be obtained from `w` by changing exactly one letter in `w`. For example, `wordNetwork["hello"]` is the list `["cello, "hallo", "hells", "hullo", "jello"]`.

Write a function called `deleteWord` that takes the dictionary `wordNetwork` and a word `w` and deletes `w` completely from the word network. Remember that `w` occurs as a key in `wordNetwork` and in addition it occurs as an element in each of the neighbor-lists of its neighbors. So, the word `"hello"` occurs as a key in `wordNetwork`, and in addition it occurs in the neighbor-list of `"cello"`, `"hallo"`, etc. Thus, the word `w` needs to be deleted both as a key and as a member of neighbor-lists. My code for this function, including the function header, is 4 lines long.

(c) (**15 points**) Write a function called `evaluate` that takes two parameters: (i) a dictionary of variables and associated values and (ii) an arithmetic expression involving variables and the "+" operator. The function should return the value of the expression obtained after substituting values for the variables in the expression. For example, suppose that `D` is the dictionary {`"y": 3, "count": 0, "i": 1, "z": 10`}. Then the function call

```
evaluate(D, " y +z+z +count ")
```

should return the value 23, since 3 + 10 + 10 + 0 is 23. Note that the expression is a string with arbitrarily many spaces between consecutive items in the expression. The expression is guaranteed to contain only the `‘‘+’’` operator and will be correctly formed (e.g., there will not be two consecutive `‘‘+’’` operators or two consecutive variables not separated by `‘‘+’’`). However, it is not guaranteed that the variables in the expression are present in the given dictionary `D`. If this happens, the function should just return `None` since it cannot evaluate the expression. My code for this function is 9 lines long.

3. (**30 points**) On *recursion*

(a) (**15 points**) Consider the recursive function below.

```
def mystery(s):
    vowels = ["a", "e", "i", "o", "u"]

    minLoc = len(s)
    for ch in vowels:
        if ch in s:
            loc = s.index(ch)
            if loc < minLoc:
                minLoc = loc

    #base case
    if minLoc == len(s):
        return [s]

    # recursive case
    if minLoc < len(s):
        return [s[:minLoc]] + mystery(s[minLoc+1:])
```

(i) What does the function call `mystery("breakfast")` return?

(ii) What does the function call `mystery("Westward Ho!")` return?

(iii) Describe in one sentence what the function does.

(b) (**15 points**) Here is the code for *quickSort*, discussed in class and posted on the course website. Note that We have inserted two print statements into the function `quickSort`.

```
def swap(L, i, j):
    temp = L[i]
    L[i] = L[j]
    L[j] = temp

def partition(L, first, last):
    p = first
    for current in range(p+1, last+1):
        if L[current] < L[p]:
            swap(L, current, p+1)
            swap(L, p, p+1)
            p = p + 1
    return p

def quickSort(L, first, last):
    if first < last:
        p = partition(L, first, last)
        quickSort(L, first, p-1)
        print L[first:p]
        quickSort(L, p+1, last)
        print L[p+1:last+1]
```

What output does the following function call produce:

`quickSort(L, 3, 7)`

when `L = [10, 11, 2, 9, 1, 4, 8, 2, 12, 1, 15]`?

4. (**40 points**) On *objects and classes*. We want to define a class called `rational` that can be used to represent and manipulate rational numbers. Here is an example of how a user might interact with our `rational` class:

```
>>> p = rational(10, 12)
>>> p
5/6
>>> q = rational(6, 9)
>>> q
2/3
>>> p+q
3/2
```

Here is the initialization method of the rational class.

```
class rational():
    def __init__(self, p, q):
        g = gcd(p, q)
        self.num = p/g
        self.den = q/g
```

Here I am assuming that there is a pre-defined greatest common divisor function called `gcd` that I can use. Your task is to write *all* the other methods you need in order to make the interaction shown above possible. You do not need to implement any methods not needed for this interaction.