6.2 Sets

repeats

Write the function repeats(L) which takes a list L and returns a sorted list of all the repeated elements in L.

For example, if L = [1,2,3,2,1], then the repeated elements are 2 and 1. We return a sorted list of these elements, so repeats([1,2,3,2,1]) == [1, 2].

Note: Recall that in this unit, you cannot create new lists or mutate existing lists. Instead, think of a clever way to use sets to solve the problem.

Hints:

You may want to use two sets here as you loop over all the values in the list L -- one set to keep track of the values you have already seen at least once, and another set to keep track of the values you have seen at least twice (that is, the duplicates in L). Recall that you can use sorted(s) here to convert a set into a sorted list of the values in that set.

In [1]:

```
# from cmu_cs3_utils import testFunction

def repeats(L):
    seen = set()
    seenAgain = set()
    for v in L:
        if v not in seen:
            seen.add(v)
        else:
            seenAgain.add(v)
    return sorted(seenAgain)

# @testFunction
def testRepeats():
    assert(repeats([1,2,3,2,1]) == [1,2])
    assert(repeats([1,2,3,2,2,4]) == [2])
    assert(repeats([1,5,3,5,2,3,2,1]) == [1,2,3,5])
    assert(repeats([7,9,1,3,7,1]) == [1,7])
    assert(repeats(list(range(100))) == [ ])
    assert(repeats(list(range(100))*5) == list(range(100)))

    # Verify that the function is nonmutating:
    L = [1,2,3,2,1]
    repeats(L)
    assert(L == [1,2,3,2,1])

def main():
    testRepeats()

# main()
```

hasNoDuplicates

In [2]:

```
# from cmu_cs3_utils import testFunction

def hasNoDuplicates(L):
```

```
    return len(set(L)) == len(L)

# @testFunction
def testHasNoDuplicates():
    assert(hasNoDuplicates([1,2,3,2]) == False)
    assert(hasNoDuplicates([1,2,3,4]) == True)
    assert(hasNoDuplicates([ ]) == True)
    assert(hasNoDuplicates([42]) == True)
    assert(hasNoDuplicates([42, 42]) == False)

    # Verify that the function is nonmutating:
    L = [1,2,3,2]
    hasNoDuplicates(L)
    assert(L == [1,2,3,2])

def main():
    testHasNoDuplicates()

main()
```

inBothLists

In [3]:

```
def inBothLists(L, M):
    return sorted(set(L) & set(M))

# @testFunction
def testInBothLists():
    assert(inBothLists([1,2,3],[3,2,4]) == [2,3])
    assert(inBothLists([3,2,1],[4,1,2,1]) == [1,2])
    assert(inBothLists([3,2,1,2],[2,2,2]) == [2])
    assert(inBothLists([1,2,3],[4,5,6,1]) == [1])
    assert(inBothLists([3,2,1,2],[4]) == [])

    # Verify that the function is nonmutating:
    L = [1,2,3]
    M = [3,2,4]
    inBothLists(L, M)
    assert(L == [1,2,3] and M == [3,2,4])

def main():
    testInBothLists()

main()
```

reverseStrings

Write the function reverseStrings(L) which takes a list L of arbitrary values (all of which are guaranteed to be immutable -- so strings, integers, floats, etc), and returns a set of the strings in L where the reverse of the string is also in L.

For example, if L = ['abc', 42, None, 'cba', 'ack'], then reverseStrings(L) returns the set {'abc', 'cba'}.

In [4]:

```
def reverseStrings(L):
    result = set()
    values = set(L)
    for v in L:
        if isinstance(v, str):
            if v[::-1] in values:
                result.add(v)
```

```
    return result

# @testFunction
def testReverseStrings():
    assert(reverseStrings([ 'abc', 42, None, 'cba', 'ack']) == {'abc', 'cba'})
    assert(reverseStrings([ 'abc', 42, None, 'cb', 'ack']) == set())
    assert(reverseStrings(['kayak', 'wow', 'taco cat']) == {'kayak', 'wow'})
    assert(reverseStrings([42, 24, True, 'eurT']) == set())

def main():
    testReverseStrings()

main()
```

inOnlyOneList

Write the function inOnlyOneList(L, M) which takes two lists L and M, and returns a sorted list of all the values that occur in either L or M but not in both lists.

For example, inOnlyOneList([1,2,3],[3,2,4]) returns [1, 4] because 1 is only in L and 4 is only in M.

Note: Recall that in this unit, you cannot create new lists or mutate existing lists. Instead, think of a clever way to use sets to solve the problem.

In [5]:

```
def inOnlyOneList(L, M):
    return sorted((set(L)-set(M)) | (set(M)-set(L)))

# @testFunction
def testInBothLists():
    assert(inOnlyOneList([1,2,3],[3,2,4]) == [1,4])
    assert(inOnlyOneList([2,3,4],[1,2,3]) == [1,4])
    assert(inOnlyOneList([3,2,1],[4,1,2,1]) == [3,4])
    assert(inOnlyOneList([3,2,1,2],[2,2,2]) == [1,3])
    assert(inOnlyOneList([3,2,1,2],[4]) == [1,2,3,4])
    assert(inOnlyOneList([3,2,1,2],[1,3,2]) == [])

    # Verify that the function is nonmutating:
    L = [1,2,3]
    M = [3,2,4]
    inOnlyOneList(L, M)
    assert(L == [1,2,3] and M == [3,2,4])

def main():
    testInBothLists()

main()
```

isPermutation

Background: A permutation of a list L is a list which contains the same elements as L, but in any order. For example, the following lists are all the permutations of [1, 2, 3]:

[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]

With this in mind, write the function isPermutation(L), which takes a list L, and returns True if L is a permutation of the list of numbers from 0 to (n - 1) inclusive, and False otherwise. n is the length of L.

Note: Recall that in this unit, you cannot create new lists or mutate existing lists. Instead, think of a clever way to use sets to solve the problem.

In [6]:
```
def isPermutation(L):
    permutation = set(range(len(L)))
    return set(L) == permutation

# @testFunction
def testIsPermutation():
    assert(isPermutation([0,2,1,4,3]) == True)
    assert(isPermutation([1,3,0,4,2]) == True)
    assert(isPermutation([1,3,5,4,2]) == False)
    assert(isPermutation([1,4,0,4,2]) == False)

    # Verify that the function is nonmutating:
    L = [0,2,1,4,3]
    isPermutation(L)
    assert(L == [0,2,1,4,3])

def main():
    testIsPermutation()

main()
```

upperAndLower

Write the function upperAndLower(s) that takes a string s and returns a set of all the letters that appear in s as both upper and lower case. The letters in the result should all be lowercase.

For example, 'This Is Not That' contains 'I' and 'i' and also 'T' and 't', so upperAndLower('This Is Not That') should return {'i', 't'}.

Hint: You should loop over the string only once, and you should not use s.count(t) or (t in s). Instead, create two sets to track the lowercase and uppercase letters you have seen so far in s. Then use these two sets to compute the result.

In [7]:
```
def upperAndLower(s):
    result = set()
    allLetters = set()
    for char in s:
        allLetters.add(char)
    for char in allLetters:
        if char.islower():
            if char.upper() in allLetters:
                result.add(char)
    return result

# @testFunction
def testUpperAndLower():
    assert(upperAndLower('This Is Not That') == {'i', 't'})
    assert(upperAndLower('') == set())
    assert(upperAndLower('abc DEF!!!') == set())
    assert(upperAndLower('abc abc!!!') == set())
    assert(upperAndLower('abc CbA!!!') == {'a', 'c'})

def main():
    testUpperAndLower()
```

```
main()
```

oneWeatherReport

Each line starts with a city name (which can contain spaces), followed by a colon, followed by the weather for that city. The report can also contain blank lines, and also lines with comments (that start with '#') should be ignored.

With this in mind, write the function oneWeatherReport(report1, report2) that takes two such reports (multiline strings), and returns a set of the cities that are included in exactly one but not both of the two reports.

In [8]:

```
# from cmu_cs3_utils import testFunction

def oneWeatherReport(report1, report2):
    cityNames1 = findCityName(report1)
    cityNames2 = findCityName(report2)
    return ((cityNames1-cityNames2) | (cityNames2-cityNames1))

def findCityName(report):
    cityNames = set()
    for line in report.splitlines():
        if '#' in line:
            pass
        elif ':' in line:
            colonPosition = line.find(':')
            cityName = line[:colonPosition]
            cityNames.add(cityName)
    return cityNames

# @testFunction
def testOneWeatherReport():
    weatherReport1 = '''
# Colder cities:
Boston: cloudy and cold
Chicago: frigid
Denver: sunny and cold

# Warmer cities:
Miami: hot and humid
San Diego: sunny and warm
'''

    weatherReport2 = '''
# Northern cities:
Boston: strong winds
Chicago: heavy rain

# Western cities:
Denver: cold, cold, cold!
Los Angeles: hot, hot, hot!
San Diego: beautiful!
'''
    assert(oneWeatherReport(weatherReport1, weatherReport2) ==
           {'Miami', 'Los Angeles'})



    weatherReport3 = '''
Seattle: freezing!
Pittsburgh: warm and sunny!
```

```
#Hey, there's a comment here!
Orlando: humid and hot!
'''

    weatherReport4 = '''
Orlando: still sunny over here!

Pittsburgh: cloudy and gray
#It might rain in Pittsburgh next week

Seattle: still freezing here!'''
    assert(oneWeatherReport(weatherReport3, weatherReport4) ==
           set())


def main():
    testOneWeatherReport()

main()
```

6.3 Dictionaries

mostCommonName

Background: Given a list of names such as ['Jane', 'Aaron', 'Cindy', 'Aaron'], the most common name is the one that occurs the most frequently in the list.

With this in mind, first write the function getCounts(L), which takes a list L and returns a dictionary where each element in the list is mapped to the number of times it occurs in the list.

Note: You may not use the list method count to solve this problem.

Then, write the function mostCommonName(names) which takes a list of names and returns the name that occurs most frequently. If there is more than one such name, return a set of the most common names. If the list is empty, return None. For the above list, your function should return 'Aaron'.

In [9]:

```
# from cmu_cs3_utils import testFunction

def getCounts(L):
    counts = dict()
    for name in L:
        counts[name] = counts.get(name, 0) + 1
    return counts

def mostCommonName(names):
    counts = getCounts(names)
    champName = set()
    champCount = 0
    for name in counts:
        currCount = counts[name]
        if currCount > champCount:
            champName = set([name])
            champCount = counts[name]
        elif currCount == champCount:
            champName.add(name)
```

```
    if len(champName) == 0:      return None
    elif len(champName) == 1:    return champName.pop()
    return champName

# @testFunction
def testGetCounts():
    L = ['Jane', 'Aaron', 'Cindy', 'Aaron']
    assert(getCounts(L) == {'Jane': 1, 'Aaron': 2, 'Cindy': 1})

    L = ['Jane', 'Aaron', 'Cindy', 'Aaron', 'Jane']
    assert(getCounts(L) == {'Jane': 2, 'Aaron': 2, 'Cindy': 1})

    # Verify that the function is nonmutating:
    L = ['Jane', 'Aaron', 'Cindy', 'Aaron']
    getCounts(L)
    assert(L == ['Jane', 'Aaron', 'Cindy', 'Aaron'])

# @testFunction
def testMostCommonName():
    L = ['Jane', 'Aaron', 'Cindy', 'Aaron']
    assert(mostCommonName(L) == 'Aaron')

    L = ['Jane', 'Aaron', 'Cindy', 'Aaron', 'Jane']
    assert(mostCommonName(L) == {'Jane', 'Aaron'})

    assert(mostCommonName([]) == None)

    L = ['Asad', 'Namrata', 'Kyra', 'Mira']
    assert(mostCommonName(L) == {'Asad', 'Mira', 'Namrata', 'Kyra'})

    # Verify that the function is nonmutating:
    L = ['Jane', 'Aaron', 'Cindy', 'Aaron']
    mostCommonName(L)
    assert(L == ['Jane', 'Aaron', 'Cindy', 'Aaron'])

def main():
    testGetCounts()
    testMostCommonName()

main()
```

getHoursLogged

Background: We can represent the times a person starts and stops working as a list of (time, person) tuples. Each person will have exactly two logs in the list: one for the time they check-in to work and one for the time they check-out of work. For example, consider the following list:

L = [(0, 'Spongebob'), (10, 'Krabs'), (30, 'Squidward'), (55, 'Krabs'), (250, 'Squidward'), (300, 'Spongebob')]

This means that Spongebob works between minutes 0 and 300, Krabs works between minutes 10 and 55, and Squidward works between minutes 30 and 250. With this in mind, write the function getHoursLogged(logs), which takes such a list, and returns a dictionary mapping each person in the logs to the total number of minutes they worked. You are guaranteed that the tuples are ordered by the time values. For the above list, your function should return { 'Spongebob': 300, 'Krabs': 45, 'Squidward' : 220 }.

In [10]:
```
# from cmu_cs3_utils import testFunction
import copy

def getHoursLogged(logs):
    names = set()
    hoursLogged = dict()
    for time, name in logs:
        if name not in names:
            names.add(name)
            hoursLogged[name] = time
        else:
            hoursLogged[name] = time - hoursLogged[name]
    return hoursLogged


# @testFunction
def testGetHoursLogged():
    logs1 = [(0, 'Spongebob'), (10, 'Krabs'), (30, 'Squidward'),
             (55, 'Krabs'), (250, 'Squidward'), (300, 'Spongebob')]
    assert(getHoursLogged(logs1) ==
        {'Spongebob': 300, 'Krabs': 45, 'Squidward': 220})

    logs2 = [(10, 'A'), (20, 'A')]
    assert(getHoursLogged(logs2) == {'A': 10})

    logs3 = [(10, 'A'), (20, 'A'), (21, 'B'), (22, 'B')]
    assert(getHoursLogged(logs3) == {'A': 10, 'B': 1})

    assert(getHoursLogged([]) == dict())

    # Verify that the function is nonmutating:
    L = [(10, 'A'), (20, 'A')]
    getHoursLogged(L)
    assert(L == [(10, 'A'), (20, 'A')])

def main():
    testGetHoursLogged()

main()
```

invertDictionary

Background: Given a dictionary d that maps keys to values, we will say that the inverse of d is the dictionary that maps the original values back to their keys. Since multiple keys can map to the same value in d, we will map the values to a set of keys that originally mapped to them. For example, if we have this dictionary:

{ 'Chicago' : 'IL', 'Pittsburgh' : 'PA', 'Philadelphia' : 'PA' }

Then the inverted dictionary would be: { 'IL' : {'Chicago'}, 'PA' : {'Pittsburgh', 'Philadelphia'} }

With this in mind, write the function invertDictionary(d), which takes a dictionary d and returns the inverted dictionary as defined above. You may assume that all the values in the original dictionary are immutable, so they are all legal keys in the resulting inverted dictionary.

In [11]:
```
# from cmu_cs3_utils import testFunction

def invertDictionary(d):
    inverted = dict()
```

```
    for city in d:
        if d[city] not in inverted:
            state = d[city]
            inverted[state] = set([city])
        else:
            inverted[state].add(city)
    return inverted

# @testFunction
def testInvertDictionary():
    d = {'Chicago' : 'IL', 'Pittsburgh' : 'PA', 'Philadelphia' : 'PA'}
    inverted = {'IL' : {'Chicago'}, 'PA' : {'Pittsburgh', 'Philadelphia'}}
    assert(invertDictionary(d) == inverted)
    # Verify that the function is nonmutating:
    assert(d == {'Chicago' : 'IL', 'Pittsburgh' : 'PA', 'Philadelphia' : 'PA'}

    d = {1 : 'a', 2 : 'b', 3 : 'c', 4 : 'd'}
    inverted = {'a' : {1}, 'b' : {2}, 'c' : {3}, 'd' : {4}}
    assert(invertDictionary(d) == inverted)

    d = {'x' : 5, 'y' : 5, 'z' : 5}
    inverted = {5 : {'x', 'y', 'z'}}
    assert(invertDictionary(d) == inverted)

    assert(invertDictionary(dict()) == dict())

def main():
    testInvertDictionary()

main()
```

sparseMatrixAdd

Background: We have previously represented a matrix by a 2d list of numbers in Python. To add two matrices, we add each corresponding element of the two matrices. So m3[0][0] == m1[0][0] + m2[0][0] and so on. For example,

```
m1 = [[ 0, 1 ],
      [ 1, 0 ]]

m2 = [[ 1, 1 ],
      [ 2, 0 ]]

# m3 = m1 + m2
m3 = [[ 1, 2 ],
      [ 3, 0]]
```

A "sparse" matrix is a matrix that is mostly 0's, with just a few non-zero values. Instead of a 2d list, we will represent a sparse matrix using a dictionary, that maps (row, col) tuples to the value at that position. We will also store the dimensions explicitly in the keys 'rows' and 'cols'. So, this matrix: m = [[0, 0, 0], [0, 5, 0]]

can be represented as this sparse matrix: sm = { 'rows': 2, 'cols': 3, (1,1): 5 } With this in mind, write the function sparseMatrixAdd(sm1, sm2), which takes two such sparse matrices sm1 and sm2, and returns the sparse matrix that results from adding them. If the input matrices are not the same size, use the larger size in each dimension for your result.

In [12]:

```
# from cmu_cs3_utils import testFunction
import copy

def sparseMatrixAdd(sm1, sm2):
    # if the index is new, add it to the result dictionary
    # if the index is the same, add it to the value that already exists
    result = dict()
    maxRows = 0
    maxCols = 0
    for key in sm1:
        if key == 'rows':
            if sm1['rows'] > maxRows:
                maxRows = sm1['rows']
        if key == 'cols':
            if sm1['cols'] > maxCols:
                maxCols = sm1['cols']
        else:
            if key in result:
                result[key] += sm1[key]
            else:
                result[key] = sm1[key]

    for key in sm2:
        if key == 'rows':
            if sm2['rows'] > maxRows:
                maxRows = sm2['rows']
        if key == 'cols':
            if sm2['cols'] > maxCols:
                maxCols = sm2['cols']
        else:
            if key in result:
                result[key] += sm2[key]
            else:
                result[key] = sm2[key]

    result['rows'] = maxRows
    result['cols'] = maxCols
    return result

# @testFunction
def testSparseMatrixAdd():

    assert(sparseMatrixAdd(
        {'rows':2, 'cols':2, (1,1):2},
        {'rows':2, 'cols':2, (1,1):5}) ==
        {'rows':2, 'cols':2, (1,1):7})

    assert(sparseMatrixAdd(
        {'rows':2, 'cols':2, (1,1):2},
        {'rows':3, 'cols':3, (1,2):5}) ==
        {'rows':3, 'cols':3, (1,1):2, (1,2): 5})

    assert(sparseMatrixAdd(
        {'rows':5, 'cols':4, (1,1):2, (1,2):3},
        {'rows':3, 'cols':6, (1,1):5, (2,2):6}) ==
        {'rows':5, 'cols':6, (1,1):7, (1,2):3, (2,2):6})

    # Verify that the function is nonmutating:
    d1 = {'rows':2, 'cols':2, (1,1):2}
    d2 = {'rows':2, 'cols':2, (1,1):5}
    sparseMatrixAdd(d1, d2)
    assert(d1 == {'rows':2, 'cols':2, (1,1):2} and
           d2 == {'rows':2, 'cols':2, (1,1):5})
```

```
def main():
    testSparseMatrixAdd()

main()
```

integerLetterFrequencies

Background: Given a string s, the "frequency" of a letter in s is the letter's count divided by the length of s, expressed as a percentage (i.e. multiplied by 100). So, the frequency of 'a' in the string 'abcabcab' can be calculated as follows:

Number of Occurences of 'a': 3 Length of String: 8 Frequency of 'a' in 'abcabcab': (3 / 8) * 100 = 37.5 The "integer frequency" of a letter in s is the integer floor of its frequency (i.e. the largest integer less than or equal to the calculated frequency). Thus, the integer frequency for the above example is 37.

With that in mind, write the function integerLetterFrequencies(s) that takes a string s and returns a dictionary mapping each uppercase letter from 'A' to 'Z' to its corresponding integer frequency in s. Note that:

Your dictionary should not include letters that never occur in s nor non-letters (like digits, spaces, and punctuation). Your function should be case insensitive. A lowercase 'a' in s counts as an uppercase 'A'. For example, if we consider the string 'abca-bcab', we can obtain integerLetterFrequencies('abca-bcab') as follows:

Length of 'abca-bcab': 9 Integer Frequencies of Its Letters: 'A': (3 / 9) *100 = 33.333... --> 33 'B': (3 / 9)* 100 = 33.333... --> 33 'C': (2 / 9) * 100 = 22.222... --> 22 Returned Dictionary: {'A': 33, 'B': 33, 'C': 22}

Additional Specifications: When you write this function, you should only loop once over the input string s, and you should not use built-in string methods such as s.count(t) to calculate letter frequencies. You also should not check if a character exists in a string using c in s. However, you are allowed to check if a character exists as a key in a dictionary.

Hint: Create a dictionary that maps each letter in s (case-insensitively) to its number of occurences.

In [13]:

```
# from cmu_cs3_utils import testFunction
import math

def integerLetterFrequencies(s):
    result = dict()
    count = dict()
    for val in s:
        if val.isalpha():
            if val in result or val.upper() in result:
                count[val.upper()] += 1
                percentage = math.floor((count[val.upper()]/len(s))*100)
                result[str(val.upper())] = percentage
            else:
                count[val.upper()] = 1
                result[str(val.upper())] = math.floor((1/len(s))*100)
    return result

# @testFunction
```

```
def testIntegerLetterFrequencies():
    assert(integerLetterFrequencies('abca-bcab') == {'A': 33, 'B': 33, 'C': 22
    assert(integerLetterFrequencies('XyYxX') == {'X': 60, 'Y': 40})
    assert(integerLetterFrequencies('AaAaAaAa!') == {'A': 88})
    assert(integerLetterFrequencies('.A1BBB') == {'A': 16, 'B':50})
    assert(integerLetterFrequencies('?*!#') == dict())
    assert(integerLetterFrequencies('a') == {'A': 100})
    assert(integerLetterFrequencies('b ') == {'B': 50})
    assert(integerLetterFrequencies('') == dict())


def main():
    testIntegerLetterFrequencies()

main()
```

countMap

Write the function countMap(L) that takes a list L of sets containing immutable values (such as strings, integers, and floats) and returns a dictionary mapping integer counts to a set of values occuring that many times within L.

For example, say that L = [{'a', 'b'}, {'a', 'c'}, {'a', 'b', 'd'}]. We see that 'a' occurs 3 times, 'b' occurs 2 times, and 'c' and 'd' each occur 1 time. So, the resulting dictionary should be { 1: {'c', 'd'}, 2: {'b'}, 3: {'a'} }).

Hint: It may be helpful to first create a dictionary that maps each present immutable value to the overall number of times it occurs in the sets in L. In the example above, this would be: {'a': 3, 'b': 2, 'c': 1, 'd': 1}.

In [14]:

```
# from cmu_cs3_utils import testFunction

def getCount(L):
    count = dict()
    for s in L:
        for v in s:
            count[v] = count.get(v, 0) + 1
    return count

def countMap(L):
    result = dict()
    charCount = getCount(L)
    for letter in charCount:
        number = charCount[letter]
        if number in result:
            result[number].add(letter)
        else:
            result[number] = set([letter])
    return result

# @testFunction
def testCountMap():
    assert(countMap([{'a', 'b'}, {'a', 'c'}, {'a', 'b', 'd'}])
           == {1: {'c', 'd'}, 2: {'b'}, 3: {'a'} })
    assert(countMap([{'d'}, {'x', 'y'}]) == {1: {'d', 'x', 'y'}})
    assert(countMap([{'a', 'b', 'c'}]) == {1: {'a', 'b', 'c'}})
    assert(countMap([{1, 'a'}, {2, 'a'}]) == {1: {1, 2}, 2: {'a'}})
    assert(countMap([{1, 1.1, 'a'}, {1.1, 'b', ('a', 'b')}])
           == {1: {1, 'a', 'b', ('a', 'b')}, 2: {1.1}})
    assert(countMap([]) == dict())
```

```
def main():
    testCountMap()

main()
```

friendsOfFriends

Background: We can create a dictionary mapping people to sets of their friends. For example:

{ 'Fred' : {'Wilma', 'Betty', 'Robert'}, 'Betty' : {'Alice', 'Wilma'}, 'Wilma' : {'Fred'}, 'Alice' : {'Betty'}, 'Robert' : set() }

A friend-of-friend of a person is a friend of one of that person's friends. For example, Alice is a friend-of-friend of Fred, since Alice is one of Betty's friends, and Betty is a friend of Fred.

We don't consider direct friends as friends-of-friends, so Wilma would not be a friend-of-friend of Fred, even though she's one of Betty's friends. Similarly, a person can't be one of their own friends-of-friends. So Fred would not be a friend-of-friend of himself, even though he is one of Wilma's friends.

With this in mind, write the function friendsOfFriends(friends), which takes a dictionary mapping people to sets of their friends, and returns a dictionary mapping people to sets of their friends-of-friends. For example, calling friendsOfFriends on the dictionary of friends above would return:

{ 'Fred' : {'Alice'}, 'Betty' : {'Fred'}, 'Wilma' : {'Betty', 'Robert'}, 'Alice' : {'Wilma'}, 'Robert' : set() }

Notes:

You may assume that everyone listed in any of the friend sets is also included as a key in the dictionary. You may not assume that if Person1 lists Person2 as a friend, Person2 will also list Person1 as a friend. Sometimes friendships are only one-way.

In [16]:

```
# from cmu_cs3_utils import testFunction
import copy

def friendsOfFriends(friends):
    fof = dict()
    for person in friends:
        fof[person] = set()
        for friend in friends[person]:
            for friendOfFriend in friends[friend]:
                if (friendOfFriend not in friends[person] and
                    friendOfFriend != person):
                    fof[person].add(friendOfFriend)
    return fof

# @testFunction
def testFriendsOfFriends():
    friends = {
                'Fred'   : {'Wilma', 'Betty', 'Robert'},
                'Betty'  : {'Alice', 'Wilma'},
                'Wilma'  : {'Fred'},
                'Alice'  : {'Betty'},
                'Robert' : set()
```

```
            }

    fof = {
            'Fred'  : {'Alice'},
            'Betty' : {'Fred'},
            'Wilma' : {'Betty', 'Robert'},
            'Alice' : {'Wilma'},
            'Robert': set()
          }
    friends2 = copy.deepcopy(friends) # Used to verify function is nonmutating
    assert(friendsOfFriends(friends) == fof)
    # Verify that the function is nonmutating:
    assert(friends == friends2)

    friends = {
                'A' : {'B'},
                'B' : {'C'},
                'C' : {'A'}
              }

    fof = {
            'A' : {'C'},
            'B' : {'A'},
            'C' : {'B'}
          }
    assert(friendsOfFriends(friends) == fof)

    friends = {
                'Melissa' : {'Joshua', 'Anna'},
                'Joshua' : {'Orelia'},
                'Anna' : {'Cynthia'},
                'Orelia' : set(),
                'Cynthia' : set()
              }

    fof = {
            'Melissa': {'Orelia', 'Cynthia'},
            'Joshua': set(),
            'Anna': set(),
            'Orelia': set(),
            'Cynthia': set()
          }
    assert(friendsOfFriends(friends) == fof)

    assert(friendsOfFriends(dict()) == dict())


def main():
    testFriendsOfFriends()

main()
```

movieAwards

Write the function movieAwards(oscarResults) which takes a set of tuples, where each tuple holds the name of a category and the name of the winning movie, and returns a dictionary mapping each movie to the number of awards it won.

Note: Remember that sets and dictionaries are unordered, so the returned dictionary may be in a different order than what we have shown for the example above, and that is ok.

In [17]:

```
# from cmu_cs3_utils import testFunction

def movieAwards(oscarResults):
    result = dict()
    for (prize, movie) in oscarResults:
        result[movie] = result.get(movie, 0) + 1
    return result

# @testFunction
def testMovieAwards():
    test = {('Best Picture', 'The Shape of Water'),
            ('Best Actor', 'Darkest Hour'),
            ('Best Actress', 'Three Billboards Outside Ebbing, Missouri'),
            ('Best Director', 'The Shape of Water')}
    result = {'Darkest Hour': 1,
              'Three Billboards Outside Ebbing, Missouri': 1,
              'The Shape of Water': 2}
    assert(movieAwards(test) == result)

    test = {('Best Picture', 'Moonlight'),
            ('Best Director', 'La La Land'),
            ('Best Actor', 'Manchester by the Sea'),
            ('Best Actress', 'La La Land')}
    result = {'Moonlight': 1,
              'La La Land': 2,
              'Manchester by the Sea': 1}
    assert(movieAwards(test) == result)

    test = {('Best Picture', '12 Years a Slave'),
            ('Best Director', 'Gravity'),
            ('Best Actor', 'Dallas Buyers Club'),
            ('Best Actress', 'Blue Jasmine')}
    result = {'12 Years a Slave': 1,
              'Gravity': 1,
              'Dallas Buyers Club': 1,
              'Blue Jasmine': 1}
    assert(movieAwards(test) == result)

    test = {('Best Picture', 'The King\'s Speech'),
            ('Best Director', 'The King\'s Speech'),
            ('Best Actor', 'The King\'s Speech')}
    result = {'The King\'s Speech': 3}
    assert(movieAwards(test) == result)

    test = {('Best Picture', 'Spotlight'), ('Best Director', 'The Revenant'),
            ('Best Actor', 'The Revenant'), ('Best Actress', 'Room'),
            ('Best Supporting Actor', 'Bridge of Spies'),
            ('Best Supporting Actress', 'The Danish Girl'),
            ('Best Original Screenplay', 'Spotlight'),
            ('Best Adapted Screenplay', 'The Big Short'),
            ('Best Production Design', 'Mad Max: Fury Road'),
            ('Best Cinematography', 'The Revenant')}
    result = {'Spotlight': 2,
              'The Revenant': 3,
              'Room': 1,
              'Bridge of Spies': 1,
              'The Danish Girl': 1,
              'The Big Short': 1,
              'Mad Max: Fury Road': 1}
    assert(movieAwards(test) == result)

    assert(movieAwards(set()) == dict())
```

```
def main():
    testMovieAwards()

main()
```

makeSpeciesDictionary

Background: We will encode animal data in a multiline string, where each line represents an animal. The lines will all be of the form species,breed,name. For example:

'''\ dog,labrador,fred cat,persian,betty dog,shepherd,barney dog,labrador,fred dog,labrador,wilma '''

With this in mind, write the function makeSpeciesDictionary(animalData), which takes data formatted as shown above, and returns a dictionary mapping each species to another dictionary that maps each breed of that species to a set of the names in the data for that species. For example, if we call makeSpeciesDictionary on the string above, it should return:

{ 'dog' : { 'labrador' : {'fred', 'wilma'}, 'shepherd' : {'barney'} }, 'cat' : { 'persian' : {'betty'} } }

In [18]:

```
# from cmu_cs3_utils import testFunction

def makeSpeciesDictionary(animalData):
    if animalData == '': return dict()
    result = dict()
    for line in animalData.splitlines():
        firstCommaIndex = line.find(',')
        species = line[:firstCommaIndex]
        line = line[firstCommaIndex+1:]
        if species not in result:
            result[species] = dict()

        secondCommaIndex = line.find(',')
        breed = line[:secondCommaIndex]
        if breed not in result[species]:
            result[species][breed] = set()

        name = line[secondCommaIndex+1:]
        result[species][breed].add(name)

    return result

# @testFunction
def testMakeSpeciesDictionary():
    animalData = '''\
dog,labrador,fred
cat,persian,betty
dog,shepherd,barney
dog,labrador,fred
dog,labrador,wilma
'''
    result = {
                'dog' :
                        {
                          'labrador' : {'fred', 'wilma'},
                          'shepherd' : {'barney'}
                        },
                'cat' :
                        {
                          'persian' : {'betty'}
                        }
```

```
            }
    assert(makeSpeciesDictionary(animalData) == result)

    assert(makeSpeciesDictionary("") == dict())


def main():
    testMakeSpeciesDictionary()

main()
```

6.4 Efficiency

getPairSum

Write the function getPairSum(L, target) which takes a list L of integers and a target value (also an integer), and returns a pair of numbers from L as a tuple if they add up to the given target number. If no such pair exists, it returns None. If there is more than one valid pair, you can return any of them.

For example, getPairSum([5, 2], 7) may return either (5, 2) or (2, 5). getPairSum([1, 0, 3], 2) should return None since there is no pair of numbers in [1, 0, 3] that sum to 2.

Important Note: Your solution must run in no worse than O(N) time. This will be manually graded. Passing the autograder does not necessarily mean your function meets the efficiency requirements.

In [19]:

```
# from cmu_cs3_utils import testFunction
import copy

def getPairSum(L, target):
    seen = set()
    for i in range(len(L)):
        num1 = L[i]
        num2 = target - num1
        if num2 not in seen:
            seen.add(num1)
        else:
            return (num1, num2)
    return None

# This helper function is used by the test function below
def isPairSum(L, target, pair):
    L = copy.copy(L)
    # Check that the pair is a tuple of length 2
    if type(pair) != tuple or len(pair) != 2: return False
    n1, n2 = pair
    # Check that the sum of the pair equals the target
    if n1 + n2 != target: return False
    # Check that both elements are in the list
    if n1 not in L: return False
    L.remove(n1)
    return n2 in L

# @testFunction
def testGetPairSum():
    L = [5, 2]
    target = 7
    pair = getPairSum(L, target)
```

```
    assert(isPairSum(L, target, pair))

    L = [10, -1, 1, -8, 3]
    target = 2
    pair = getPairSum(L, target)
    assert(isPairSum(L, target, pair))

    L = [10, -1, 1, -8, 3, 1]
    target = 2
    pair = getPairSum(L, target)
    assert(isPairSum(L, target, pair))

    L = [3, 3]
    target = 6
    pair = getPairSum(L, target)
    assert(isPairSum(L, target, pair))

    assert(getPairSum([1, 0, 3], 2) == None)
    assert(getPairSum([10, -1, 1, -8, 3, 1], 10) == None)
    assert(getPairSum([], 0) == None)

def main():
    testGetPairSum()

main()
```

containsPythagoreanTriple

Write the function containsPythagoreanTriple(L) which takes a list L of positive integers, and returns True if there are three values (a, b, c) anywhere in L such that (a, b, c) form a Pythagorean Triple (where `a**2 + b**2 == c**2`), and False otherwise.

For example, containsPythagoreanTriple([1, 3, 6, 5, 1, 4]) returns True. The list contains (3, 4, 5), which is a Pythagorean Triple because `3**2 + 4**2 == 5**2`.

Important Note: Your solution must run in no worse than O(N^2) time. This will be manually graded. Passing the autograder does not necessarily mean your function meets the efficiency requirements.

In [20]:

```
# from cmu_cs3_utils import testFunction, rounded

def containsPythagoreanTriple(L):
    squaredList = [val ** 2 for val in L]
    additionList = []
    for i in range(len(squaredList)):
        for j in range(i+1, len(squaredList)):
            additionList.append(squaredList[i]+squaredList[j])
    for val1 in additionList:
        for val2 in squaredList:
            if val1 == val2:
                return True
    return False


# @testFunction
def testContainsPythagoreanTriple():
    # 3, 4, 5 is a Pythagorean Triple
    assert(containsPythagoreanTriple([1, 3, 6, 5, 1, 4]) == True)
    # 5, 12, 13 is a Pythagorean Triple
    assert(containsPythagoreanTriple([5, 7, 1, 6, 12, 13]) == True)
    # 6, 8, 10 is a Pythagorean Triple
```

```
    assert(containsPythagoreanTriple([8, 6, 10]) == True)
    # 8, 15, 17 is a Pythagorean Triple
    assert(containsPythagoreanTriple([2, 8, 15, 16, 17]) == True)

    assert(containsPythagoreanTriple([1, 3, 6, 2, 8, 1, 4]) == False)
    assert(containsPythagoreanTriple([21, 3, 2, 7, 5]) == False)
    assert(containsPythagoreanTriple([5, 13]) == False)
    assert(containsPythagoreanTriple([]) == False)

def main():
    testContainsPythagoreanTriple()

main()
```

findTriplets

Write the function findTriplets(L) which takes a list L of integers and returns a set of all triplets in the list whose sum is equal to 0. Each triplet in the set should be sorted.

For example, if L = [-1, 0, -3, 2, 1], then findTriplets(L) should return {(-1, 0, 1), (-3, 1, 2)}. If there are no valid triplets, return the empty set.

Important Note: Your solution must run in no worse than O(N2) time. This will be manually graded. Passing the autograder does not necessarily mean your function meets the efficiency requirements.

In [21]:

```
# from cmu_cs3_utils import testFunction

'''
x1 + x2 + x3 = 0
x1 + x2 = -x3
'''

def findTriplets(L):
    result = set()
    if len(L) >= 3:
        # checkedSet = set()
        for i in range(len(L)):
            checkedSet = set()
            # checkedSet.add(L[i])
            for j in range(i+1, len(L)):
                sumValue = L[i] + L[j]
                if (-sumValue) in checkedSet:
                    newList = sorted([L[i], L[j], -sumValue])
                    result.add(tuple(newList))
                else:
                    checkedSet.add(L[j])
    return result

# @testFunction
def testFindTriplets():
    L = [-1, 0, -3, 2, 1]
    result = {(-1, 0, 1), (-3, 1, 2)}
    assert(findTriplets(L) == result)

    L = [5, 6, -1, -8, 3]
    result = {(-8, 3, 5)}
    assert(findTriplets(L) == result)

    L = [1, -2, 0, 1, 0, 0]
    result = {(-2, 1, 1), (0, 0, 0)}
```

```
    assert(findTriplets(L) == result)

    assert(findTriplets([10, 20, 30]) == set())
    assert(findTriplets([-10, 20, 1]) == set())
    assert(findTriplets([0, 0]) == set())
    assert(findTriplets([]) == set())

    result = {(-10, 1, 9), (-10, 0, 10), (-1, 0, 1)}
    assert(findTriplets([7, -10, 10, 0, -1, 8, -2, 1, -4, 9]) == result)

def main():
    testFindTriplets()

main()
```

6.5 Unit 6 Exercises

semesterGrades

In [23]:

```
# from cmu_cs3_utils import testFunction, rounded

def gradebookAsTable(gradebookText):
    table = []
    isNumber = False
    for line in gradebookText.splitlines():
        newLine = []
        for item in line.split():
            if item.isnumeric():
                newLine.append(int(item))
            else:
                newLine.append(item)
        table.append(newLine)
    return table

def studentNames(gradebookText):
    table = gradebookAsTable(gradebookText)
    result = []
    for lineNumber in range(1, len(table)):
        result.append(table[lineNumber][0])
    return tuple(sorted(result))

def categories(gradebookText):
    gradebookTable = gradebookAsTable(gradebookText)
    titleList = gradebookTable[0]
    result = []
    for index in range(1, len(titleList)):
        titleStr = titleList[index]
        newStr = ''
        for char in titleStr:
            if char.isalpha():
                newStr += char
        if newStr not in result:
            result.append(newStr)
    return result

def studentScores(gradebookText, categoryWeights, student):
    gradebookTable = gradebookAsTable(gradebookText)
    rows, cols = len(gradebookTable), len(gradebookTable[0])
    names = [gradebookTable[row][0] for row in range(rows)]
    categoryName = categories(gradebookText)
    header = gradebookTable[0][1:] + ['test']
    result = dict()
```

```
    semesterGrade = 0
    if student in names:
        rowIndex = names.index(student)
    for colIndex in range(len(categoryName)):
        category = categoryName[colIndex]
        sumOfScore = 0
        number = 0
        for testCategoryIndex in range(len(header)):
            testCategory = header[testCategoryIndex]
            if category in testCategory:
                score = gradebookTable[rowIndex][testCategoryIndex+1]
                if isinstance(score, int):
                    sumOfScore += gradebookTable[rowIndex][testCategoryIndex+1
                    number += 1
            else:
                if number != 0:
                    grade = rounded(sumOfScore / number)
                    result[category] = grade
                continue
    for colIndex in range(len(categoryName)):
        category = categoryName[colIndex]
        weight = categoryWeights.get(category, 0) / 100
        score = result.get(category, 0)
        semesterGrade += int(score) * weight
    result['semester'] = rounded(semesterGrade)
    return result

def semesterGrades(gradebookText, categoryWeights):
    names = list(studentNames(gradebookText))
    newCategory = []
    for category in categories(gradebookText):
        newCategory += [category + 'Avg']
    result = []
    header = ['student'] + newCategory + ['semesterAvg'] + ['semesterGrade']
    result.append(header)
    for name in names:
        studentScore = studentScores(gradebookText, categoryWeights, name)
        scoreList = []
        for categoryGrade in studentScore:
            scoreList.append(studentScore[categoryGrade])
        letterGrade = getLetterGrade(scoreList[-1])
        nameScoreList = [name] + scoreList + [letterGrade]
        result.append(nameScoreList)
    return getStringResult(result)

def getLetterGrade(numericGrade):
    if numericGrade >= 90:   return 'A'
    elif numericGrade >= 80: return 'B'
    elif numericGrade >= 70: return 'C'
    elif numericGrade >= 60: return 'D'
    else:                    return 'F'

def getStringResult(resultTable):
    result = ''
    header = getHeaderString(resultTable[0])
    result += header
    result += '\n'
    rows, cols = len(resultTable), len(resultTable[0])
    for rowIndex in range(1, rows):
        studentScore = getScoreString(resultTable[0], resultTable[rowIndex])
        if rowIndex == rows - 1:
            result += studentScore
        else:
            result += studentScore
```

```
            result += '\n'
    return result

def getHeaderString(headerList):
    headerStr = ''
    for index in range(len(headerList)):
        category = headerList[index]
        if index == 0:
            headerStr += category
        else:
            headerStr += '   ' + category
    return headerStr

def getScoreString(headerList, scoreList):
    scoreStr = ''
    name = scoreList[0]
    scoreStr += name.ljust(len('student'))
    for index in range(1, len(scoreList)):
        grade = scoreList[index]
        scoreStr += f'{grade}'.rjust(len(headerList[index])+2)
    return scoreStr


def getSimpleGradebook():
    return '''\
student  quiz1
wilma       90'''

def getMediumGradebook1():
    return '''\
student  hw1  quiz1  exam1
wilma     90     89     75'''

def getMediumGradebook2():
    return '''\
student  quiz1
wilma       80
barney      90'''

# @testFunction
def testGradebookAsTable():
    gradebook1 = getSimpleGradebook()
    assert(gradebookAsTable(gradebook1) ==
    [ ['student', 'quiz1'],
      ['wilma', 90] ])

    gradebook2 = getMediumGradebook1()
    assert(gradebookAsTable(gradebook2) ==
    [ ['student', 'hw1', 'quiz1', 'exam1'],
      ['wilma', 90, 89, 75] ])

# @testFunction
def testStudentNames():
    gradebook1 = getSimpleGradebook()
    assert(studentNames(gradebook1) == ('wilma',))

    gradebook2 = getMediumGradebook2()
    assert(studentNames(gradebook2) == ('barney', 'wilma'))

# @testFunction
def testCategories():
    gradebook1 = getSimpleGradebook()
    assert(categories(gradebook1) == ['quiz'])
```

```
    gradebook2 = getMediumGradebook1()
    assert(categories(gradebook2) == ['hw', 'quiz', 'exam'])

# @testFunction
def testStudentScores():
    gradebook1 = getMediumGradebook2()
    categoryWeights = {'quiz': 100}
    assert(studentScores(gradebook1, categoryWeights, 'barney') ==
           {'quiz': 90, 'semester': 90})

    gradebook2 = getMediumGradebook1()
    categoryWeights = {'quiz': 30, 'hw': 30, 'exam': 40}
    assert(studentScores(gradebook2, categoryWeights, 'wilma') ==
           {'hw': 90, 'quiz': 89, 'exam': 75, 'semester': 84})


# @testFunction
def testSemesterGrades():
    gradebook1 = getSimpleGradebook()
    categoryWeights = {'quiz': 100}
    assert(semesterGrades(gradebook1, categoryWeights) == '''\
student  quizAvg  semesterAvg  semesterGrade
wilma         90           90              A''')

    gradebook2 = getMediumGradebook1()
    categoryWeights = {'quiz': 30, 'hw': 30, 'exam': 40}
    assert(semesterGrades(gradebook2, categoryWeights) == '''\
student  hwAvg  quizAvg  examAvg  semesterAvg  semesterGrade
wilma       90       89       75           84              B''')

    gradebook3 = getMediumGradebook2()
    categoryWeights = {'quiz': 100}
    assert(semesterGrades(gradebook3, categoryWeights) == '''\
student  quizAvg  semesterAvg  semesterGrade
barney        90           90              A
wilma         80           80              B''')


# @testFunction
def testAll():
    gradebookText = '''\
    student  hw1   hw2   hw3   hw4  quiz1   quiz2   quiz3  exam1  exam2
    wilma     94    97    91    83     81      95      87    exc     94
    fred      90    88    92    85     77      81      90     88     82
    barney    75    83    71   exc     68      71      73     72     76
    betty    100   100    99   100    exc      98      92     88    100'''
    categoryWeights = { 'hw':50, 'quiz':10, 'exam':40 }

    gradebookTable = [
      ['student','hw1','hw2','hw3','hw4',
       'quiz1','quiz2','quiz3','exam1','exam2'],
      ['wilma',94,97,91,83,81,95,87,'exc',94],
      ['fred',90,88,92,85,77,81,90,88,82],
      ['barney',75,83,71,'exc',68,71,73,72,76],
      ['betty',100,100,99,100,'exc',98,92,88,100],
    ]
    assert(gradebookAsTable(gradebookText) == gradebookTable)
    assert(studentNames(gradebookText) == ('barney', 'betty', 'fred', 'wilma')
    assert(categories(gradebookText) == [ 'hw', 'quiz', 'exam' ])
    assert(studentScores(gradebookText, categoryWeights, 'wilma') ==
          {'hw':91, 'quiz':88, 'exam':94, 'semester':92})
    assert(studentScores(gradebookText, categoryWeights, 'fred') ==
          {'hw':89, 'quiz':83, 'exam':85, 'semester':87})
    assert(studentScores(gradebookText, categoryWeights, 'barney') ==
```

```
          {'hw':76, 'quiz':71, 'exam':74, 'semester':75})
    assert(studentScores(gradebookText, categoryWeights, 'betty') ==
          {'hw':100, 'quiz':95, 'exam':94, 'semester':97})
    assert(semesterGrades(gradebookText, categoryWeights) == '''\
    student  hwAvg  quizAvg  examAvg  semesterAvg  semesterGrade
    barney      76       71       74           75              C
    betty      100       95       94           97              A
    fred        89       83       85           87              B
    wilma       91       88       94           92              A''')

def main():
    testGradebookAsTable()
    testStudentNames()
    testCategories()
    testStudentScores()
    testSemesterGrades()
    testAll()

# main()
```