Review

Exam 3 Topics:
Dictionaries, sets, efficiency, recursion, OOP

3. (8 points) **Free Response: Six-Seven Twins**

Two **distinct** positive integers `a` and `b` are *Six-Seven-Twins* (coined term) if they contain the same number of 6s and the same number of 7s in their decimal representation, and each number contains at least one 6 and one 7. Other digits, if any, do not matter.

For example:

- 1667 and 7606 are Six-Seven-Twins (each has two 6s and one 7)
- 1667 and 7617 are **not** Six-Seven-Twins (different counts of 6s and 7s)
- 177 and 771 are **not** Six-Seven-Twins (no 6s)

With this in mind, write a function `containsSixSevenTwins(L)` that takes a list `L` of positive integers and returns `True` if there exist two **distinct** numbers in the list that are Six-Seven-Twins, and `False` otherwise.

Your solution must run in $O(N)$ time, where $N =$ `len(L)`.

Notes: When analyzing efficiency, assume that scanning the digits of a single number takes constant time. What matters for efficiency is the length of the list, not the number of digits in each number. Your goal is to design an $O(N)$ solution with respect to the number of elements in the list.

Here are some test cases:

```
assert containsSixSevenTwins([67, 76]) == True
assert containsSixSevenTwins([7, 77, 777]) == False
assert containsSixSevenTwins([166, 606, 700]) == False
assert containsSixSevenTwins([1, 2, 3]) == False
assert containsSixSevenTwins([671, 761, 42]) == True
assert containsSixSevenTwins([123, 890, 45, 67]) == False
assert containsSixSevenTwins([676, 676]) == False   # same value twice: not distinct
assert containsSixSevenTwins([676]) == False        # only one element
assert containsSixSevenTwins([666777, 67677, 776766]) == True
```

Dictionaries and Sets

- You should know the basic functionality
- Free Response Problems usually have efficiency requirements

Example: Longest Matching Schedule (F23)

You are given a dictionary rosters that maps each program (e.g., 'CS', 'IS', 'BS') to another dictionary. That inner dictionary maps each course number to a set of student names enrolled in that course.

Your task is to write a function `longestScheduleMatch(rosters)` that returns the size of the **longest matching schedule**.

A matching schedule occurs when two or more students are enrolled in **exactly the same set of courses**. In other words, if two students take the **same combination of courses** (no more, no less), that combination counts as one matching schedule.

```
'CS':
    {'15-213': {'ahmed'},
     '15-150': {'ahmed'}
    },
'IS':
    {'67-262': {'betty', 'ahmed'}
    },
'HCI':
    {'05-318': {'fred'}},
'BS':
    {'70-340': {'omar', 'huda'},
     '70-257': {'ahmed', 'betty'}
    }
```

⇨ 1

```
'IS':
    {'67-262': {'betty', 'ahmed'}
    },
'CS':
    {'15-150': {'ahmed', 'betty'}
    },
'BA':
    {'09-105': {'fred'}
    },
'BS':
    {'70-340': {'omar', 'huda'},
     '70-257': {'ahmed', 'betty'}
    }
```

⇨ ?

Recursion

- Some problems are easier if you create *recursive helpers*
- Some problems require the use of a *wrapper*

Example: Recursive Rearrange Letters (F23)

8. (12 points) **Free Response: Recursive Rearrange Letters**

Write the **recursive** function `recRearrangeLettersInFront(s)`, which takes a string `s` (which may contain any valid ASCII characters). The function should return a new string with all the alphabetical letter characters moved to the front, maintaining their original order. The non-alphabetical characters should remain at the end of the result string, preserving their original relative order as in the input string `s`.

For instance,

```
assert(recRearrangeLettersInFront('teatime4me') == 'teatimeme4')
assert(recRearrangeLettersInFront('h3lp3rfunct10n5') == 'hlprfunctn33105')
assert(recRearrangeLettersInFront(':-soeasy)') == 'soeasy:-)')
assert(recRearrangeLettersInFront('1Ch34?e\te&7') == 'Chee134?\t&7')
assert(recRearrangeLettersInFront('ILove15-112') == 'ILove15-112')
assert(recRearrangeLettersInFront('123') == '123')
assert(recRearrangeLettersInFront('') == '')
```

Your solution must use recursion. If you use any loops or iterative functions, you will receive no points on this problem.

OOP (F24)

6. (16 points) **Free Response**: OOP

Write the classes `Light` and `DimmableLight` so that the following test code runs without errors. Do not hardcode against the values used in the testcases, though you can assume the testcases cover the needed functionality. You must use proper object-oriented design or you will lose points.

```
a = Light(500, "Bedroom")
assert str(a) == "Light (lumens=500, location=Bedroom, active=off)"
assert a.isOn() == False
a.toggleSwitch()
assert str(a) == "Light (lumens=500, location=Bedroom, active=on)"
assert a.isOn() == True
a.toggleSwitch()
assert str(a) == "Light (lumens=500, location=Bedroom, active=off)"
assert a.isOn() == False

# The only attributes that matter for equality are the lumens and location
b = Light(500, "Bedroom")
b.toggleSwitch()
assert b.isOn() == True
assert a == b
assert a != Light(501, "Bedroom")
assert a != Light(500, "Bathroom")
assert a != "Light (lumens=500, location=Bedroom, active=off)"

d = DimmableLight(300, "Dining Room")
assert str(d) == "DimmableLight (lumens=300, location=Dining Room, active=off, intensity=100%)"
d.dim(75)
assert str(d) == "DimmableLight (lumens=300, location=Dining Room, active=off, intensity=75%)"
# Invalid dim values do nothing
d.dim(101)
d.dim(-1)
assert str(d) == "DimmableLight (lumens=300, location=Dining Room, active=off, intensity=75%)"
d.toggleSwitch()
assert str(d) == "DimmableLight (lumens=300, location=Dining Room, active=on, intensity=75%)"

# Like a normal light, DimmableLights are equal based only on the lumens and location
e = DimmableLight(300, "Dining Room")
e.toggleSwitch()
e.dim(15)
assert d == e
assert d != DimmableLight(301, "Dining Room")
assert d != DimmableLight(300, "Living Room")
assert d != "DimmableLight (lumens=300, location=Dining Room, active=on, intensity=75%)"
# Lights and DimmableLights are never equal
assert Light(300, "Dining Room") != DimmableLight(300, "Dining Room")

# Both work in sets
s = set()
s.add(a)
s.add(d)
s.add(e)
assert len(s) == 2
assert Light(500, "Bedroom") in s
assert DimmableLight(300, "Dining Room") in s
assert Light(500, "Bathroom") not in s

# Various type checking things
assert isinstance(a, Light)
assert not isinstance(a, DimmableLight)
assert isinstance(d, Light)
assert isinstance(d, DimmableLight)
```