6.009 Quiz 3 -- Spring 2017

This quiz is **open-book and closed-internet**. Feel free to use any physical materials you have brought, but you may not access resources online (except web.mit.edu/6.009) or access files on your computer outside of the quiz directory. Proctors will be available to answer administrative questions and clarify specifications of coding problems, but they should not be relied on for coding help.

Each test case is worth one point for a total of 20 points.

You *must* submit your modified `quiz.py` via web.mit.edu/6.009 before the deadline in order to receive credit. This quiz assumes you have Python 3.5 (or a later version) installed on your machine.

The `resources` directory contains the **Python Language Reference** and the **Python Library Reference** in PDF form. Please **do not import any Python modules** to use as part of your solutions -- quizzes with `import` statements (or their equivalent!) will be given a grade of 0.

As in the labs, `test.py` can be used to test your code in `quiz.py`. The problems have different numbers of tests as specified. Remember that you can run specific tests by listing the test numbers, like so

```
python3 test.py 1 3 5      # run tests #1, #3, #5
```

Your score will be computed from the number of tests you pass. If you pass all the tests for a problem, you'll receive full credit. If you pass 2 of the 5 tests, you'll receive 40% of the credit for the problem. Note that to pass some of the tests in the time allotted, your solution may have to be reasonably efficient.

Problem 1: `Database Search` (tests 1-12)

This question consists of 3 parts and is related to Question 3 of the first practice Quiz 3.*The solution to Question 3 is available in the resources directory, and you can use the code as a starting point if you wish.*

You are given a database of scheduling information about a set of classes over a semester. We make the following assumptions about the schedule:

1. A class meets at most once on any given day.
2. Meetings last exactly 1 hour.

3. Meetings can only begin on the hour at 09:00, 10:00, ..., 16:00.
4. The semester consists of 15 weeks: Week `"1"`, Week `"2"`, ..., Week `"15"`.
5. All meetings occur in one of the following buildings: `"Walker"`, `"Stata"`, `"W20"`, `"Koch"`, `"Kresge"`, `"W66"`, `"W36"`, `"E17"`, `"E51"`.

Each day in the semester is identified by its `WEEK_NUMBER` and its `DAY_OF_WEEK`, e.g. the 5th Friday.

The database consists of two parts, the `default_db` database and the `update_db` database. `default_db` contains the original schedule that is now **outdated**. `default_db` is a list of weekly entries. Each entry is itself a list of strings: a class name, a time of day, a day of the week, and a building location. The time is the hour at which the meeting starts, according to a 24-hour clock. These meetings were originally scheduled to occur every week.

The following is a small example `default_db` that contains three weekly entries.

```
[
    ["6.009", "15", "Monday", "W36"],
    ["6.009", "14", "Friday", "Koch"],
    ["6.01", "10", "Tuesday", "Stata"],
]
```

Next, the `update_db` database contains updates to the original schedule. There are two types of updates: ADDs and DELETEs. We represent `update_db` as a list of updates. Each update is itself a list of strings: an update type, a class name, a time, a day of the week, a building location, and a week number. Unlike the weekly entries in `default_db`, each update in `update_db` only gives information for a specific day. Each ADD update refers to a meeting that does not already exist in `default_db`. Each DELETE update refers to a meeting that exists in `default_db`.

Extending the example above, the following is an `update_db` that contains two updates.

```
[
    ["ADD", "6.009", "12", "Thursday", "Walker", "15"],
    ["DELETE", "6.009", "15", "Monday", "W36", "15"]
]
```

After updating the original schedule in `default_db` with `update_db`, we see that our schedule should consist of the following meetings:

1. `"6.009"` meets at `"15"` on `"Monday"`s in `"W36"` on every week *except* Week `"15"`.
2. `"6.009"` meets at `"12"` on `"Thursday"` in `"Walker"` during Week `"15"`.

3. `"6.009"` meets at `"14"` on `"Friday"` in `"Koch"` every week.
4. `"6.01"` meets at `"10"` on `"Tuesday"` in `"Stata"` every week.

In the following parts, you will implement methods to retrieve information about the schedule. They require an implementation of the `build_rep()` method. `build_rep()` processes the two databases and returns new representations of the information. The output of `build_rep()` is passed into the other methods. You can choose whatever representation you think will work best for the methods of Parts A, B and C.

The `default_db` and `update_db` databases will be provided as arguments to `build_rep()`, but not the other methods. For your convenience, you can look through raw text versions of the databases in the `database/` folder. It might be useful to search these files when debugging. We do not recommend printing the entire databases at once, since they are quite large.

Part A: Tests (1-3)

For Part A you will implement the `get_near_classes()` method to the specification below.

INPUTS:

1. `buildings`, a list of buildings as strings
2. `rep`, the representation of `default_db` and `update_db` produced by `build_rep()`

OUTPUT:

A list, in no particular order, of all classes that meet**only** in the list of buildings given by `buildings`.

EXAMPLES:

Using the above example,

1. `get_near_classes(["Walker"], rep)` should return `[]`.
2. `get_near_classes(["Walker", "W36", "Stata"], rep)` should return `["6.01"]`.
3. `get_near_classes(["Walker", "W36", "Koch", "Stata"], rep)` could return `["6.009", "6.01"]` or the same set of classes in a different order.

Part B: Tests (4-8)

For Part B you will implement the `earliest_meeting()` method which retrieves the **time of the earliest meeting in a given building on a given day of the week across all weeks**.

INPUTS:

1. `building`, a string such as `"Stata"`.
2. `day_of_week`, a string such as `"Friday"`.
3. `rep`, the representation of `default_db` and `update_db` produced by `build_rep()`

OUTPUT:

An **integer** earliest time (hour, such as `9`), the earliest meeting given by the combined database of classes, occurring on **any** week in `building` on `day_of_week`. If no meetings take place on `day_of_week` in that building on any week, return `None`.

EXAMPLE:

Using the tiny database in the example above, `earliest_meeting("Stata", "Tuesday", default_db, update_db)` should return `10`, while `earliest_meeting("Stata", "Monday", default_db, update_db)` should return `None`.

Part C: Tests (9-12)

For Part C you will implement the `have_conflicts()` method to the specification below.

INPUTS:

1. `class_list`, a list of classes as strings
2. `rep`, the representation of `default_db` and `update_db` produced by `build_rep()`

OUTPUT:

A Boolean (`True` / `False`) indicating whether any two classes in `class_list` conflict. Two classes conflict when they meet on the same day of the week during the same week at the same time.

EXAMPLE:

Using the above example, `have_conflicts(["6.009", "6.01"], rep)` should return `False`.

Problem 2: `k-Label Poker` (tests 13-20)

In a deck of normal playing cards, each card has two labels, a rank (1,2,...,K) and a suit (diamonds, spades, clubs, hearts), so (K,spades) and (4,diamonds) are both possible cards. In our version of poker, each card has `k` different labels, where each label is a positive integer 1,

…, `L`. Therefore, if `k = 4` and `L = 5`, then (1,1,1,2) and (2,1,5,1) are both valid cards.

Your task is to count the number of straights that are possible in k-label poker with hands of size `n`. Repeat cards within a hand are not allowed. In our version of poker, a straight happens if all of the labels in the hand form a nondecreasing sequence. For example, if `k = 2`, `L = 5`, and `n = 3`, then the hand `[(1,1), (1,2), (3,5)]` would be a valid straight since the sequence 1,1,1,2,3,5 is nondecreasing but `[(3,5), (3,1), (2,2)]` would not. Note that `[(1,2), (1,1), (3,5)]` is a different hand (and is not a straight) from `[(1,1), (1,2), (3,5)]` (which is a straight).

Note that this problem has a very special definition of a straight that is different from the usual poker straight (if you happen to play poker). This specialness includes two unusual aspects: first, the labels for any given card in the hand must be non-descending; and second, the labels aggregated (in order) across all (ordered) cards in the hand must also be non-descending.

Please implement `count_straights(k, L, n)`, where `k` is the number of labels for a given card, `L` is the maximum integer label for any card, and `n` is the length of hand. Your function should return the number of possible straights in hands of size `n` where each card has `k` labels of maximum value `L`. Cards cannot be repeated in a straight.

Performance hint: You will pass most of the tests with a brute-force search that generates all the hands and counts the number of straights, but may time out on some tests. It is not necessary to generate all possible hands to count straights, but if you find that difficult to reason about or code, try the brute-force method.

Example 1: `count_straights(2, 3, 4)` The possible cards are (1,1), (1,2), (1,3), ..., (3, 3).

We are considering hands of length 4. In this case, 5 straights are possible:

```
[(1,1), (1,2), (2,2), (2,3)]
[(1,1), (1,2), (2,2), (3,3)]
[(1,1), (1,2), (2,3), (3,3)]
[(1,1), (2,2), (2,3), (3,3)]
[(1,2), (2,2), (2,3), (3,3)]
```

Example 2: `count_straights(3, 5, 2)`

The possible cards are (1,1,1), (1,1,2), (1,1,3), ..., (5,5,5).

We are considering hands of length 2. In this case, 205 straights are possible:

```
[(1,1,1), (1,1,2)]
[(1,1,1), (1,1,3)]
…
[(1,2,3), (4,4,5)]
[(1,2,3), (4,5,5)]
…
[(4,5,5), (5,5,5)]
```