Quiz 3

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 funprog). 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 problem is worth 5 points, for a total of 15 points, which will be scaled to one quiz's worth of credit.

You *must* submit your quiz via funprog before the deadline in order to receive credit. This quiz assumes you have Python 2.7 installed on your machine.

The `resources` directory contains **Python documentation** for commonly-used data structures.

Question 1: `get_popular_edge`

Eagerly looking forward to summer break, you've decided to help out a local airline company, UpInTheAir, in exchange for a free ticket to insert-beachy-getaway-name for your summer vacation. UpInTheAir wants to make sure that their airspace isn't too congested as that could lead to crashes and more importantly lost revenue.

UpInTheAir has modeled their airspace as a graph. Nodes represent cities and an edge from A to B represents that UpInTheAir offers a flight from A to B. Namely, edges are directed! For instance, if there exists an edge from Star City to Central City, then UpInTheAir offers a flight from Star City to Central City, but not necessarily in the other direction. More concretely, a graph is a dictionary with city names as keys and a list of reachable cities as values. For example, this is a miniature example airspace graph:

```
{
  "Star City": ["Central City"],
  "Gotham": ["Star City"],
  "Central City": ["Gotham"]
}
```

If you started at Star City the only way to get to Gotham would be to fly first to Central City and then to Gotham.

UpInTheAir will provide you with their airspace map and a list of travel itineraries which are a list of multi-hop trips. For instance, [[“Star City”, “Central City”, “Gotham”], [“Central City”, “Gotham”]] is a valid list of travel itineraries. You can think of it as a list of paths. Unfortunately, UpInTheAir has

some stale data, and so some paths contain cities that no longer are served by UpInTheAir or contain hops that are no longer offered (alternatively said, some nodes or edges no longer can be found in the graph).

What you need to do in order to score your free vacation is to help UpInTheAir determine which edge (airspace) is most frequently traversed. Of course, this edge must exist in the current airspace and can only use valid cities. **If you encounter an itinerary (path) that contains an invalid entry – a node or an edge – all edges in that path are disqualified and cannot be counted towards the most popular edge!** In the example above, ("Central City", "Gotham") is traversed twice and all other edges are traversed once and so that's the most congested edge.

Your job is to implement get_popular_edge(graph, paths) such that given the graph of UpInTheAir's airspace and a list of paths, it returns a tuple representing the most congested edge.

(HINT: You might find it helpful to implement a helper method for determining if an edge is valid, or if a path is valid.)

You're a couple of methods away from that dream vacation. Good luck!

Question 2: `can_ben_lose`

You and Ben Bitdiddle are at it again! Delighted by your success in the last Tic Tac Toe tournament, you have decided to enter the MIT Finals Week Tic Tac Toe Toe-rnament. The final round is tomorrow, and you are giving it your all. You're up against your friend and arch rival, the devious Ben Bitdiddle.

(For some background for those of you who didn't take the practice quiz) Ben Bitdiddle, Tic Tac Toe ace, is the reigning champion of the MIT Finals Week Tic Tac Toe Toe-rnament. You, brave 6.S04 student, are aiming to unseat this legend, and claim the trophy for yourself. You've been training against the infamous Bitdiddle for weeks, and you've come to us in 6.S04 for some strategy help. Fortunately, we've got your back.

To aid you in your training, we would like you to implement the function `can_ben_lose`. Given a board and whether or not it is Ben's turn `benTurn` (a Boolean), it should tell you if there is any way Ben can lose the game. Obviously, given a blank board, Ben can lose, but can he lose if the board is partially filled in? That is your job to tell us. Given a partially filled in board, you should report whether or not Ben lost. Assume Ben can play in any open space on the board, and does not necessarily move to the optimal space. **Note that Ben always chooses to play the 'o' player. Also note that if the game ends in a tie, they both lose.**

Here is a sample board:

```
[
  [1, 0, 1],
  [0, 0, -1],
  [0, 1, -1]
]
```

where 1 is 'x', 0 is 'o' and -1 is an empty spot.

It is your turn. Ben can lose! Success! You've got this! The toe-rnament will be easy with 6.S04 on your side!

Question 3: `earliest_meeting`

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. Days of the week are given by strings such as `"Monday"` or `"Saturday"`.
4. Meetings can only begin on the hour at 09:00, 10:00, ..., 16:00, given by integer hours: `9`, `10`, ... `16`.
5. The semester consists of 15 weeks: Week `"1"`', Week `"2"`, ..., Week `"15"`.
6. 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` (string) and its `DAY_OF_WEEK` (string), e.g., the 5th Friday. Note that the representation from the practice quiz did not have the extra attribute corresponding to a building.

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: the name of the class, the hour, day of the week, and building in which it meets, respectively.

```
[
  ["6.S04", "15", "Monday", "W36"],
  ["6.S04", "14", "Friday", "Koch"],
  ["6.01", "10", "Tuesday", "Stata"],
]
```

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`.

The following is an `update_db` that contains two updates (both made to week 15).

```
[
  ["ADD", "6.S04", "12", "Thursday", "Walker", "15"],
  ["DELETE", "6.S04", "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.S04"` meets at `"15"` on `"Monday"` s in `"W36"` on every week *except* Week `"15"`.
2. `"6.S04"` meets at `"12"` on `"Thursday"` in `"Walker"` during Week `"15"`.
3. `"6.S04"` meets at `"14"` on `"Friday"` in `"Koch"` every week.
4. `"6.01"` meets at `"10"` on `"Tuesday"` in `"Stata"` every week.

Implement `earliest_meeting` a method to retrieve 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. `default_db`, as defined above
4. `update_db`, as defined above

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`.

Building an effective data structure from the databases should be very helpful. Consider using objects to encapsulate the database contents. We do not recommend printing the contents of an entire database, since these are quite large. You may find it useful to convert some strings to integers. To this end, `int("123")` returns the integer `123`.