LAB 3 EXAM QUESTIONS

January 2021

Q1 - METU Registration

METU decided to make renovations on the registration system. Instead of old system, they want to change the selection ordering. Now, in order to make the renovation YOU have been chosen.

Instead of first clicks gets the lecture, now students should create a wishlist in order to take the lectures. You need to write a function named register() which takes two parameters:
offered: The list of the courses that METU offers.
wishlist: The wishlist of the students that students created.

The list offered has courses in the format (coursename, totalCapacity, usedCapacity). Each course that is stored in the offered is type of list. You can check the example offered list below:

```
    [['Ceng111', 155, 151], ['Ceng140', 150, 147], ['Ceng232', 143, 140]].
```

The wishlist is the total list of the students that contains the student name and followed by selection lists of the students. The list given in the priority order. An example wishlist:

```
    [['Student1','Ceng111', 'Ceng140'],
    ['Student2','Ceng140', 'Ceng232'],
    ['Student3','Ceng232', 'Ceng140']]
```

In order to be fair, first selections of the students matched if possible. The first student takes its first selection if possible (if there is enough capacity in the course) and we continue with second student (if there is not enough space just skip that turn for the student) and so on. The registration process continues until all the wishlist finishes somehow.

Your function should return a list that contains [studentName, courseName] pairs with the found matchings. The ordering is **IMPORTANT**. Your list should be ordered by how the registration process happened.

Sample IO:

```
IO1:
>>> offered = [['Ceng111', 10, 9]]
>>> wishlist = [['Student0', 'Ceng111']]
>>> register(offered, wishlist)
[['Student0', 'Ceng111']]

IO2:
>>> offered = [['Ceng111', 10, 9], ['Ceng140',10,9]]
>>> wishlist = [['Student0', 'Ceng111'],['Student1', 'Ceng140']]
>>> register(offered, wishlist)
[['Student0', 'Ceng111'], ['Student1', 'Ceng140']]

IO3:
>>> offered = [['Ceng111', 10, 9], ['Ceng140',10,8]]
>>> wishlist = [['Student0', 'Ceng111'],['Student1', 'Ceng111', 'Ceng140'], ['Student2', 'Ceng140']]
>>> register(offered, wishlist)
[['Student0', 'Ceng111'], ['Student2', 'Ceng140'], ['Student1', 'Ceng140']]

IO4:
>>> offered = [['Ceng111', 10, 9], ['Ceng140',10,8]]
>>> wishlist = [['Student0', 'Ceng111','Ceng140'],['Student1', 'Ceng111', 'Ceng140'], ['Student2', 'Ceng14
>>> register(offered, wishlist)
[['Student0', 'Ceng111'], ['Student2', 'Ceng140'], ['Student0', 'Ceng140']]

IO5:
>>> offered = [['Ceng111', 10, 9], ['Ceng140', 11, 9], ['Ceng232', 10, 3], ['Ceng443', 10, 1],
['Ceng351', 11, 3], ['Ceng790', 10,8], ['Ceng334',2,1]]
>>> wishlist = [['Student0', 'Ceng111', 'Ceng140','Ceng232'], ['Student1','Ceng111', 'Ceng140','Ceng232'],
['Student2', 'Ceng140', 'Ceng232'],
['Student3', 'Ceng232','Ceng351','Ceng111','Ceng443', 'Ceng334','Ceng790']]
>>> register(offered, wishlist)
[['Student0', 'Ceng111'], ['Student2', 'Ceng140'],
['Student3', 'Ceng232'], ['Student0', 'Ceng140'],
['Student2', 'Ceng232'], ['Student3', 'Ceng351'],
['Student0', 'Ceng232'], ['Student1', 'Ceng232'],
['Student3', 'Ceng443'], ['Student3', 'Ceng334'], ['Student3', 'Ceng790']]
```

Solution:

```
def register(offered,wishlist):
    registers = []
    i = 1 # counter to move forward in student wishes
    registered = True # flag to use for stopping when i > length of all student wishes
    while registered:
        registered = False
        for wish in wishlist: # go through student wish lists
            student = wish[0]
            if i < len(wish): # if the student still has course wishes that haven't been processed
                registered = True
                course = wish[i]
                for entry in offered: # look through the offered courses to match
                    if entry[0] == course: # courses match!
                        if entry[2] < entry[1]: # check the capacity, add the student if there is enough
                            registers.append([student, course])
                            entry[2] += 1 # entries are lists and thus mutable!
                        break # break out either way since we've processed the wish now
        i += 1 # advance one course in the student lists
    return registers
```

Q2 - Line111

Your task is implementing a very basic line editor design. The editor only contains a single string in one line; there is no vertical movement. It supports 5 text-based commands for modifying the string:

1. `pos i`: Move the cursor to the given non-negative zero-based index `i`. `i` can be longer than the contained text, in which case the cursor should move to the end of the line.
2. `ins str`: Insert string `str` under the cursor. Existing characters after the cursor are shifted right. e.g. if your text is `"abcd"` and your cursor is at index 2 (under `c`), then inserting `"efg"` yields `"abefgcd"`. Inserting at index 0 corresponds to prepending and inserting at the end of the line (`index == len(text)`) corresponds to appending. `str` is not quoted, and can contain only alphanumeric characters (a-z, A-Z, 0-9).
3. `del`: Delete the character under the cursor. Nothing happens if the cursor is at the end of the line.
4. `undo`: Undo the last insertion or deletion. Please note that just like in a real editor, moving the cursor is not an operation and is not affected by `undo` and `redo`. Only `ins` and `del` count.
5. `redo`: Redo the last undone operation. Just like in a real editor, doing an operation (`ins` or `del`) after undoing removes the possibility of redoing.

Two more details:

1. The cursor starts at position zero.
2. Inserting and deleting text **does not** move the cursor.
3. Getting an `undo` when there is nothing to be undone or a `redo` when there is nothing to be redone is possible. Nothing should happen in these cases.

You will write a function `line111(text, cmds)` taking the initial text contained in the editor as well as a list of commands. The function should return the last state of the text after all the commands have been run.

Hint: Use `str.split(" ")` for parsing and remember that strings are immutable.

Function signature:

```
def line111(text, cmds):
    """
    Apply the specified commands to the text (a single string) as the totally awesome line111
    editor. Remember that "pos" commands do not count for undo/redo! Return
    the final, new text!

    text:   A string. The text that will be modified by the commands.
    cmds:   A list of strings. Commands to be applied to the text sequentially.
            Explained in the description.
    """
    pass
```

Example runs:

```
>>> line111("cengaver", ["ins hello"])
"hellocengaver"
# ^ insert to initial pos 0
>>> line111("cengaver", ["del", "del", "del"])
"gaver"
# ^ delete thrice from pos 0
>>> line111("cengaver", ["pos 4", "ins iz"])
"cengizaver"
# ^ move to index 4 and insert
>>> line111("cengaver", ["pos 1000", "ins can"])
"cengavercan"
# ^ 1000 is large and moves to end of line
>>> line111("cengaver", ["pos 7", "del", "del", "del"])
```

```
"cengave"
# ^ only 1 deletion since cursor does not move
>>> line111("cengaver", ["pos 1000", "ins can", "undo"])
"cengaver"
# ^ undo the insertion
>>> line111("cengaver", ["pos 1000", "ins can", "pos 0", "pos 3", "undo"])
"cengaver"
# ^ pos has no effect on undo/redo
>>> line111("cengaver", ["pos 1000", "ins can", "undo", "redo"])
"cengavercan"
# ^ redo the undone op
>>> line111("cengaver", ["pos 1000", "ins can", "undo", "ins siniz", "redo"])
"cengaversiniz"
# ^ last redo does nothing since ins happens after undo, clearing redos
>>> line111("cengaver", ["undo", "undo", "redo"])
"cengaver"
# nothing happens since there is nothing to undo/redo
>>> line111("cengaver", ["ins has", "pos 1000", "ins ler", "pos 7", "del", "del"])
"hascengerler"
# multiple ins/del/pos operations
```

Solution:

```python
def line111(text, cmds):
    undo, redo = [], []  # stacks for undo and redo states
    ci = 0  # position
    for cmd in cmds:
        args = cmd.split(' ')
        if args[0] == 'pos':
            ci = int(args[1])
            if ci > len(text):  # set to end on overflow
                ci = len(text)
        elif args[0] in ('ins', 'del'):
            undo.append(text)  # save last text for undo
            redo = []  # clear the redo buffer
            if args[0] == 'ins': # insert using slicing
                ins_string = args[1]
                text = text[:ci] + ins_string + text[ci:]
            else: # delete using slicing
                text = text[:ci] + text[ci+1:]
        elif args[0] == 'undo' and undo:  # undo, only when undo is not empty
            redo.append(text) # put the undone text in the redo stack
            text = undo.pop()
        elif args[0] == 'redo' and redo:  # redo, only when redo is not empty
            undo.append(text) # put the redone text back in the undo stack
            text = redo.pop()
    return text
```

Q3 - Infection

Write a function named `infection()` which computes spreading of a disease at each time step.

It takes 3 arguments:

1. An `NxM` list which represents the neighbourhood of a group of people. For each `(i,j)`th position of the list, there exists a string defining the name of the person living there. If the string is equal to `'-'`, then it means there does not exist any human there.
2. A string giving the name of the person who has infected first in the neighbourhood.
3. An integer representing the time step at when you need to find which people are infected.

Spreading of the disease happens as in the following way:

- If a person P at `(i,j)`th position is infected at time step `t`, P infects the people in his/her `(i,j-1)`, `(i,j+1)`, `(i-1,j)` and `(i+1,j)` neighborhoods in order. At time step `t+1`, P gets better completely, yet his/her neighbors given above becomes ill.
- Each person can be infected only once. In other words, once a person has been infected at time `t`, s/he does not get infected at the next time steps anymore.
- The first infected person is assumed to be infected at time step `0`.

Your function is required to return the list of the people infected at time step given in the third parameter of the function.

If the spreading had terminated before the time step asked, then you are expected to return an empty list.

The ordering of the output is important and it should obey the rule given above.

HINT: You can use QUEUE.

```
EXAMPLE-1
>>> infection([['Fadime', 'Mert', 'Aylin', 'Seher', 'Murat', 'Yaren'],
        ['Gazanfer', 'Uygar', 'Derya', 'Bahri', 'Su', 'Keriman'],
        ['Rıza', 'Hale', 'Zeliha', 'Hakan', 'Merve', 'Tekin'],
        ['Bora', 'Mediha', 'Ceren', 'Kaya', 'Pelin', 'Zekeriya'],
        ['Leyla', 'Ferit', 'Zeynep', 'Duru', 'Alperen', 'Emine'],
        ['Yusuf', 'Asude', 'Batu', 'Vildan', 'Oya', 'Volkan'],
        ['Osman', 'Dilan', 'Jale', 'Turgut', 'Doruk', 'Vahap'],
        ['Neriman', 'Sarp', 'Halide', 'Gaye', 'Kunter', 'Enver'],
        ['Utku', 'Bahaddin', 'Nevin', 'Zuhal', 'Cenk', 'Erhan']], 'Duru', 4)

['Bora', 'Yusuf', 'Hale', 'Dilan', 'Derya', 'Halide', 'Tekin', 'Vahap', 'Su', 'Kunter', 'Seher', 'Zuhal']


EXAMPLE-2
>>> infection([['Fadime', 'Mert', 'Aylin', '-', '-', 'Yaren'],
        ['Gazanfer', 'Uygar', '-', 'Bahri', 'Su', 'Keriman'],
        ['Rıza', '-', 'Zeliha', '-', 'Merve', 'Tekin'],
        ['Bora', '-', 'Ceren', 'Kaya', '-', 'Zekeriya'],
        ['Leyla', 'Ferit', '-', 'Duru', 'Alperen', '-'],
        ['Yusuf', '-', 'Batu', 'Vildan', '-', 'Volkan'],
        ['-', 'Dilan', 'Jale', 'Turgut', 'Doruk', '-'],
        ['Neriman', '-', 'Halide', 'Gaye', '-', 'Enver'],
        ['-', 'Bahaddin', '-', 'Zuhal', '-', 'Erhan']], 'Duru', 3)

['Zeliha', 'Jale', 'Doruk', 'Gaye']
```

Solution:

```python
def infection(group, first_infected, query_time):
        for i in range(len(group)):
                if first_infected in group[i]:
                        j = group[i].index(first_infected)
                        current_time = 0
                        queue_infecteds = [first_infected]      # hold the names
                        queue_details = [(current_time, i, j)]  # hold the time step and positions
                        group[i][j] = '-'
                        break
        while queue_infecteds:
                if queue_details[0][0] == query_time:
                        return queue_infecteds
                infected = queue_infecteds.pop(0)
                current_time, i, j = queue_details.pop(0)
                if (j > 0 and group[i][j-1] != '-'):
                        queue_infecteds.append(group[i][j-1])
                        queue_details.append((current_time+1, i, j-1))
                        group[i][j-1] = '-'
                if (j < len(group[i])-1 and group[i][j+1] != '-'):
                        queue_infecteds.append(group[i][j+1])
                        queue_details.append((current_time+1, i, j+1))
                        group[i][j+1] = '-'
                if (i > 0 and group[i-1][j] != '-'):
                        queue_infecteds.append(group[i-1][j])
                        queue_details.append((current_time+1, i-1, j))
                        group[i-1][j] = '-'
                if (i < len(group)-1 and group[i+1][j] != '-'):
                        queue_infecteds.append(group[i+1][j])
                        queue_details.append((current_time+1, i+1, j))
                        group[i+1][j] = '-'
        return queue_infecteds
```