CS112 Homework 7: – Groups Allowed – Recursion

Due: Sun, November 17th, 11:59pm

As this is a **Homework**, you can read any and all materials, ask us questions, talk with other students, and learn however you best learn in order to solve the task. Just create your own solution from those experiences, and turn in your work.

Notes

Recursion allows us solve complex problems that are defined in terms of a smaller version of the very same problem. The functions in this homework can all be written recursively, and it will require that you spend some time thinking about what a recursive solution looks like before you start writing code.

- **You must use recursion** to solve each problem. Remember, this means that as your code progresses through the problem the same function keeps calling itself on updated ("smaller") tasks until the base case is reached.
- You cannot import any module.
- A tester has been included with this assignment. Grading is based on the fraction of passing tests.

Turning It In

Add a comment at the top of the file that indicates your name, userID, G#, lab section, a description of your collaboration partners, as well as any other details you feel like sharing. Please also mention what was most helpful for you. Once you are done, run the testing script once more to make sure you didn't break things while adding these comments. If all is well, go ahead and turn in **just your .py file** you've been working on, named with our usual convention (see above), over on BlackBoard to the correct lab assignment. Please don't turn in the tester file or any other extra files, as it will just slow us down.

Grading Rubric

```
Pass shared test cases      25x4 (zero points for hard-coding)
-----------------------------------------------------------------
TOTAL:                      100
```

Tasks

---

fill(grid, old_value, new_value, seed)

Description: This is the same function you saw in Project 4, only this time you ***must*** use recursion. It fills in a **grid** (i.e. a list of lists of `ints`) by modifying it *in place*. The filling starts at location defined by **seed** and spreads to all neighbors that have the **old_value** which is replaced by the **new_value**.

Parameters: **grid** is a 2D `list` of `int`, **seed** is a `tuple` of `int` that holds a pair of coordinates (row, column), **old_value** (`int`) is the original value at **seed**, **new_value** (`int`) is the replacement value for the filling.

Return value: `None` (the grid is modified *in place*)

Example:

```
fill([[0,9,6,0,0,0],                          [[0,9,6,2,2,2],
      [0,7,4,0,0,0],                           [0,7,4,2,2,2],
      [1,3,0,7,8,0],                           [1,3,2,7,8,2],
      [0,9,5,1,0,1]], 0, 2, (0,3))      →      [0,9,5,1,2,1]]
```

---

descendants(family_tree, name, distance)

Description: **family_tree** is a `dictionary` where the key is a person's name (`string`) and the corresponding value is a `list` of his/her children's names (`string`). The function takes a **name** and returns all its descendants that are **distance** generations apart. Assume that all names (both *keys* and *values*) are unique. However, some names might not have any children.

Parameters: **family_tree** (`dictionary`), **name** (`string`), **distance** (positive `int`)

Return value: `list` of `strings` (order of names doesn't matter), and if there are no descendants an empty list

Example:

```
descendants({'alex':['bob','chloe','dan'],
             'bob':['ellen'],
             'chloe':['filip','greg'],
             'dan':['hector','iris','jason']
             'greg':['olive','paris']}, 'alex', 2)     →

                           ['ellen', 'filip', 'greg', 'hector', 'iris', 'jason']
```
