Balchunas

AP CSP: Problem Set 3

Specification

Submit a .py file named `your_name_pset3.py`. Your file should **NOT** execute any functions. You are just filling them in. When I run your program there should be no errors and no output. You may choose to test your functions as you go, but you may delete the function calls after you test or comment them out. Comment your problems with the problem number above the function header. Remember to import the random module to use the library's functions.

Grader

For the following problems you will be writing functions inside of the grades_data.csv file provided. You are given a list of letters corresponding to multiple choice answers. Each row in the file corresponds to a singular person's exam, except for the first row which is the solutions to the exam. There is also a helper function that will help you get the grades into lists. Use the `grade_helper` module to load the data in easily. You will import it and use the `grade_loader` function which returns a tuple. The first element in the tuple is a list corresponding to the solutions for the exam, and the second element in the tuple is a 2D list of student answers. Each row is a list of multiple choice answers given for a particular student. The `grade_loader` function takes a single argument, the name of the csv file you'd like to load. An example program is shown below.

```
import grade_helper


all_grades = grade_helper.grade_loader("grades_data.csv")
solution = all_grades[0]
student_data = all_grades[1]

print(solution)
print(student_data)
```

Problem 1

Write a function that takes a 2D list of test data and a 1D list of solutions as input and returns a list of scores (not percentages, just number correct). The first element of the output list is the first person's grade, etc.

```
get_grades(data, solution)
```

Problem 2

Write a function that takes a 2D list of test data and a 1D list of solutions as input and returns a list of the number of people who got each question correct (not percentages, just number). The first element of the result list would be the number of people that got question 1 correct.

```
get_question_counts(data, solution)
```

Problem 3

Write a function that determines the highest, lowest, and average scores. Return these values as a tuple in that order. Hint: Use the result from problem 1 as input to this function.

```
get_score_statistics(scores)
```

Games

For the next set of problems, ensure that you are displaying the game boards in a way that makes sense and the current state of the game is clear to the player.

Problem 4

Connect Four (also known as Four Up, Plot Four, Find Four, Captain's Mistress, Four in a Row, Drop Four, and Gravitrips in the Soviet Union) is a two-player connection board game, in which the players choose a color and then take turns dropping colored discs into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs.

The game board is 6 rows of 7 columns. The first player (red) will specify in which column to drop his or her piece. You may use an 'R' character for this. The yellow player will then specify a column. You will then place the 'Y' inside of the appropriate column. If the row is full, do not allow the player to place their disc in that row.

Check for winners after each player's turn. Return a game winner, or tie, appropriately.

Write the function `connect_four()` that lets two users play a game of connect four. User input should be a single number, the row in which to drop a piece. The game ends with a congratulatory message to the winner, or informing the users of a tie.

Problem 5

Minesweeper is a single-player puzzle video game. The objective of the game is to clear a rectangular board containing hidden "mines" or bombs without detonating any of them, with help from clues about the number of neighbouring mines in each field. The game originates from the 1960s, and it has been written for many computing platforms in use today. It has many variations and offshoots. You will create a game of Minesweeper in the following way:

- Create two grids of width w and height h. One will be the display board and the other the actual solved game board.
- Initialize each square in the display board with a `#` to show the user that he or she has not yet selected the square.
- In the game board, randomly select `difficulty` percent of squares to have bombs `B`.
- Create an algorithm that runs over the game board and determines how many bombs each individual square is touching. Note that touching refers to both cardinal neighbors (north, south, east, west) and diagonal neighbors. Fill in each cell with the number of bombs it is touching, or a blank space for no bombs.
- Allow the user to play the game by providing a row, column pair.
- You may write a simplified algorithm in which a number or space is revealed. If the row, column pair is not a bomb, the player gets to play another round, and the player loses if a bomb is selected. The player wins if all non-bomb squares are selected.
- The game board must be shown to the person in a meaninful way at each step of the game loop.

You DO NOT need to write the functionality of the real game where if you select a square with a 0, the real game will automatically expand outwards.

To create a game of Minesweeper the function call requires a width, height, and percentage of squares to be bombs e.g. `minesweeper(10, 20, 0.25)` allows me to play a game of minesweeper where the board is 10×20 and a quarter of the spots are filled with bombs. In general, `minesweeper(rows, cols, difficulty)`.

Problem 6

Sudoku is a logic-based number placement puzzle. In classic sudoku the objective is to fill a 9×9 grid with digits so each column, row, and the nine 3×3 subgrids contain all of the digits 1 through 9. Below are three sample unsovled problems at different difficulties as well as an unsolved problem with a solution.

In this problem, write a function `sudoku(puzzle)` that creates a game of sudoku, given an initial puzzle. The user input will come as (row)(col) (number). So the input `21 4` would place a 4 in the third row and second column of the puzzle.

If the puzzle is solved correctly, a congratulatory message should be displayed and the program should finish. If the user fills the grid, but the puzzle is solved incorrectly, the program should prompt the user further for input.

Example Puzzles

```
# Easy
puzzle = [
  [7, 6, 2, 0, 0, 9, 0, 1, 0],
  [3, 0, 4, 0, 0, 0, 0, 0, 2],
  [0, 9, 7, 1, 0, 0, 0, 0, 0],
  [0, 0, 6, 0, 0, 8, 7, 0, 3],
  [5, 8, 0, 0, 6, 0, 2, 0, 1],
  [0, 7, 0, 9, 0, 0, 3, 0, 0],
  [9, 0, 1, 4, 2, 0, 0, 8, 6],
  [2, 4, 5, 0, 0, 3, 0, 0, 0]
]

# Medium
puzzle = [
  [0, 0, 1, 0, 0, 0, 0, 0, 8],
  [0, 4, 0, 0, 0, 0, 0, 0, 0],
  [8, 2, 7, 5, 4, 0, 0, 0, 0],
  [0, 0, 0, 4, 1, 0, 0, 8, 7],
  [0, 0, 0, 7, 0, 3, 0, 9, 2],
  [0, 7, 0, 0, 2, 8, 5, 0, 6],
  [4, 9, 0, 0, 0, 0, 0, 0, 3],
  [0, 0, 3, 0, 0, 0, 0, 6, 9],
  [2, 1, 8, 0, 0, 0, 0, 0, 0]
]

# Hard
puzzle = [
  [0, 0, 0, 5, 0, 1, 0, 7, 0],
  [0, 0, 7, 6, 4, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 4, 9],
  [1, 0, 0, 7, 0, 0, 0, 8, 0],
  [7, 4, 5, 0, 1, 0, 0, 0, 0],
  [0, 0, 0, 0, 9, 0, 1, 0, 0],
  [3, 0, 0, 9, 0, 0, 0, 0, 0],
  [4, 2, 0, 0, 8, 5, 0, 0, 0],
  [5, 0, 1, 0, 7, 4, 0, 0, 8]
]

# Solved
unsolved = [
  [5, 3, 0, 0, 7, 0, 0, 0, 0],
  [6, 0, 0, 1, 9, 5, 0, 0, 0],
  [0, 9, 8, 0, 0, 0, 0, 6, 0],
  [8, 0, 0, 0, 6, 0, 0, 0, 3],
  [4, 0, 0, 8, 0, 3, 0, 0, 1],
  [7, 0, 0, 0, 2, 0, 0, 0, 6],
  [0, 6, 0, 0, 0, 0, 2, 8, 0],
  [0, 0, 0, 4, 1, 9, 0, 0, 5],
  [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

solved = [
  [5, 3, 4, 6, 7, 8, 9, 1, 2],
  [6, 7, 2, 1, 9, 5, 3, 4, 8],
  [1, 9, 8, 3, 4, 2, 5, 6, 7],
  [8, 5, 9, 7, 6, 1, 4, 2, 3],
  [4, 2, 6, 8, 5, 3, 7, 9, 1],
  [7, 1, 3, 9, 2, 4, 8, 5, 6],
  [9, 6, 1, 5, 3, 7, 2, 8, 4],
  [2, 8, 7, 4, 1, 9, 6, 3, 5],
  [3, 4, 5, 2, 8, 6, 1, 7, 9]
]
```

Problem 7

Hitori is a logic-based number puzzle game played with a grid of squares, each containing a number. The game is played by eliminating squares (blocking the numbers out). The goal of the game is to end with a board that has the following three properties:

1. No one row or column can have more than one instance of any given number.
2. Black squares cannot be vertically or horizontally adjacent, although they can be diagonal to one another.
3. The remaining squares (containing numbers) must be all connected to each other horizontally, or vertically. They may be connected diagonally, but this does not count as being "connected".

Hitori puzzles come in different shapes, so use the dimensions of the input board to determine which numeric values are in the puzzle. For example, a 5×5 board would contain numbers 1 to 5 and a 9×9 board would contain numbers 1 to 9. Below are some sample input puzzles to use.

Write a function `hitori(puzzle)` which allows the user to play a Hitori puzzle. The input for the game is just a row and column pair of numbers to block or unblock a particular square. Although real hitori puzzles may be larger than 9×9, this makes displaying to the console a bit complicated since you now need two characters per square. If you're up for a challenge, make your hitori puzzle look nice for larger inputs, but otherwise you may assume that puzzles are 9×9 or smaller.

Suggestion: Use 0's in your board to represent blacked out squares. But the 0's are going to make the problem harder to read so use a hash `#` when you display them to the user.

Example Puzzles

```
# Easy
puzzle = [
  [4, 5, 3, 2, 5],
  [3, 5, 4, 2, 2],
  [5, 3, 1, 5, 1],
  [5, 4, 4, 3, 1],
  [1, 4, 5, 4, 5],
]

# Hard
puzzle = [
  [2, 5, 1, 5, 3],
  [3, 4, 3, 1, 2],
  [1, 5, 4, 2, 4],
  [3, 1, 2, 5, 4],
  [4, 5, 5, 4, 1],
]

# Medium
puzzle = [
  [8, 8, 3, 3, 5, 3, 1, 1],
  [2, 6, 1, 8, 7, 5, 3, 4],
  [3, 2, 5, 2, 1, 1, 8, 3],
  [6, 6, 7, 1, 3, 2, 8, 5],
    [4, 8, 4, 5, 3, 1, 2, 3],
    [2, 5, 6, 8, 2, 2, 4, 5],
    [7, 8, 8, 2, 6, 3, 4, 1],
    [1, 7, 2, 2, 8, 4, 5, 6]
]

# Solved
unsolved = [
    [4, 8, 1, 6, 3, 2, 5, 7],
    [3, 6, 7, 2, 1, 6, 5, 4],
    [2, 3, 4, 8, 2, 8, 6, 1],
    [4, 1, 6, 5, 7, 7, 3, 5],
    [7, 2, 3, 1, 8, 5, 1, 2],
    [3, 5, 6, 7, 3, 1, 8, 4],
    [6, 4, 2, 3, 5, 4, 7, 8],
    [8, 7, 1, 4, 2, 3, 5, 6]
]

solved = [
    [0, 8, 0, 6, 3, 2, 0, 7],
    [3, 6, 7, 2, 1, 0, 5, 4],
    [0, 3, 4, 0, 2, 8, 6, 1],
    [4, 1, 0, 5, 7, 0, 3, 0],
    [7, 0, 3, 0, 8, 5, 1, 2],
    [0, 5, 6, 7, 0, 1, 8, 0],
    [6, 0, 2, 3, 5, 4, 7, 8],
    [8, 7, 1, 4, 0, 3, 0, 6]
]
```

Images

See the README for the `images_csp.py` module for help on how to work with the following problems. Find an image, or images, that you'd like to work with from online for the following problems. You will be building the functionality of an image manipulation app! For each problem, demonstrate that it works by saving the output of your function using your original image.

Grayscale Image Problems

Problem 8

Grayscale pixel values range from 0 (black) to 255 (white). *Thresholding* an image means taking these values and setting each pixel to either black or white depending on a certain threshold value. Image processing with a thresholded image tends to be easier with 2-color images. Write a function that takes a 2D list representing a grayscale image, and a threshold. The function will return a 2D list representing the thresholded image with white pixels at or above the threshold and black pixels below.

```
threshold(image, threshold)
```

Problem 9

Adjusting the brightness of an image means making the pixel values higher (to brighten) or lower (to darken). Write a function that takes an image and an integer (positive or negative) and returns a 2D list representing the original image with the brightness adjusted. Keep in mind that the values must be between 0 and 255 for a valid grayscale image.

```
adj_brightness(image, adjustment)
```

Problems 10, 11

Images can often be reflected (flipped vertically/horizontally) using image manipulation software. Write two functions that perform this operation. Each function takes a 2D list representing an image as input and returns a 2D list as output which is the original image in the new orientation.

```
flip_vertical(image)
flip_horizontal(image)
```

Problem 12

Images can be "inverted" by making bright pixels dark, and dark pixels bright. Write a function that will invert a grayscale image by mapping white pixels to black (e.g. 0 becomes 255, 255 becomes 0) and the rest of the values follow similarly. 1 becomes 254, 2 becomes 253, etc. The function takes a 2D list representing an image as input, and returns a 2D list as output which is an inverted version of the input image.

```
invert_img(image)
```

Problem 13

Cropping refers to selecting only a region of interest of a particular image. Write a function that takes 3 parameters: a 2D list representing an image, a tuple containing the (x, y) coordinate of the top left of the rectangle to crop, and a tuple containing the width and height of the cropped rectangle. The function returns the cropped image. Note that x coordinates represent columns in 2D lists and y coordinates represent rows.

```
# Return a 10 by 20 rectangle where the top left pixel is at coordinate 10, 15
crop_img(image, (10, 15), (10, 20))
```

Problems 14, 15

Image apps often provide functionality to rotate images. Write a function that takes a 2D list representing an image as input and returns a 2D list representing a list as output. The returned list will be the image rotated 90 degrees counter-clockwise (clockwise).

```
rotate90ccw(image) # Rotates image 90 deg counter-clockwise
rotate90cw(image)  # Rotates image 90 deg clockwise
```

Problem 16

Masking refers to hiding certain pixel values by overlaying two images on top of each other. Write a function that performs a bit-wise mask (e.g. the mask is a 2D list full of 1's an 0's) on an image and returns the new image. Where the mask has a value of 1 the new image will have the value it has in image, and where the mask has a 0, the new image will have a black pixel.

```
mask_img(image, mask)
```

Color Images

Most image apps don't take grayscale images only but work with color images. Color images are 3D lists, but it's best to think of them as 2D lists of color values. Picture a grayscale image, but rather than each pixel being represented by a number 0 to 255, each pixel is represented by 3 numbers corresponding to the red, green, and blue color values in that order. Each of these values ranges from 0 to 255. For each of the following problems, demonstrate that it works by saving the output of your function using your original image.

Problem 17

Grayscale images are created from color images by taking the average of the red, green, and blue values for each pixel. Write a function that takes a 3D list representing a colored image, and return a 2D list representing the grayscale version of that image. Make sure that your averages are integers! Pixel values must be between 0 and 255 and cannot be floating point numbers.

```
cvt_grayscale(image)
```

Problem 18

Most image editing programs can apply a "sunset" filter in which the red channel gets a little brighter. Write a function that takes a 3D image and an adjustment value (integer) that will change the value of the red channel. Keep in mind that you must keep the values in a valid range (0 to 255). The function returns a 3D list with a "sunset" filter applied.

```
filter_sunset(image)
```

Problem 19

Most image editing programs support a "sepia" filter, which gives images an old-timey feel by making the whole image look a bit reddish-brown. There are a lot of algorithms to give this effect, but you can use the one following this prompt. Write a function that takes a 3D list representing an image and applies a sepia filter to it. The function returns a 3D list with the filter applied.

```
sepiaRed = .393 * originalRed + .769 * originalGreen + .189 * originalBlue
sepiaGreen = .349 * originalRed + .686 * originalGreen + .168 * originalBlue
sepiaBlue = .272 * originalRed + .534 * originalGreen + .131 * originalBlue
```

Note that these values might not be integers, or in a valid range for pixel values. Make sure you take this into account.

```
filter_sepia(image)
```

Problem 20

The Marilyn Diptych is a silkscreen painting by American pop artist Andy Warhol depicting Marilyn Monroe. The monumental work is one of the artist's most noted of the movie star. You've probably seen it before (https://www.masterworksfineart.com/artists/andy-warhol/marilyn-monroe).This problem requires you make a 2x2 image consisting of 4 copies of your input image placed in a square grid. Each image should have a different filter applied to it. One should be yellowed by having only blues and greens, one should be only red, one should be only blue, and one only green. Write a function that takes a 3D list representing a color image and returns a 3D list representing the new image.

```
warhol_effect(image)
```

Extra Credit

These two questions are extremely important for image analysis, so if you're up for the challenge, you'll be rewarded!

Problem 21

There are a number of ways to create the effect of blurring or softening an image. For this problem, you'll use the "box blur," which works by taking each pixel and, for each color value, giving it a new value by averaging the color values of neighboring pixels.

Consider the following grid of pixels, where each pixel is numbered.

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

The new value of each pixel would be the average of the values of all of the pixels that are within 1 row and column of the original pixel (forming a 3x3 box). For example, each of the color values for pixel 6 would be obtained by averaging the original color values of pixels 1, 2, 3, 5, 6, 7, 9, 10, and 11 (note that pixel 6 itself is included in the average). Likewise, the color values for pixel 11 would be be obtained by averaging the color values of pixels 6, 7, 8, 10, 11, 12, 14, 15 and 16.

For a pixel along the edge or corner, like pixel 15, we would still look for all pixels within 1 row and column: in this case, pixels 10, 11, 12, 14, 15, and 16.

Hint: The edges can be tricky, so be careful! Sometimes it can make sense to "pad" the image by adding rows/columns of 0 values to the top, bottom, and sides.

```
0  0  0  0  0 0
0  1  2  3  4 0
0  5  6  7  8 0
0  9 10 11 12 0
0 13 14 15 16 0
0  0  0  0  0 0
```

Write a function that takes a 3D image and returns a blurred version of the image.

```
filter_boxblur(image)
```

Problem 22

In artificial intelligence algorithms for image processing, it is often useful to detect edges in an image: lines in the image that create a boundary between one object and another. One way to achieve this effect is by applying the Sobel (https://en.wikipedia.org/wiki/Sobel_operator) operator to the image.

Like image blurring, edge detection also works by taking each pixel, and modifying it based on the 3x3 grid of pixels that surrounds that pixel. But instead of just taking the average of the nine pixels, the Sobel operator computes the new value of each pixel by taking a weighted sum of the values for the surrounding pixels. And since edges between objects could take place in both a vertical and a horizontal direction, you'll actually compute two weighted sums: one for detecting edges in the x direction, and one for detecting edges in the y direction. In particular, you'll use the following two "kernels":

Gx:

```
-1 0 1
-2 0 2
-1 0 1
```

Gy:

```
-1 -2 -1
 0  0  0
 1  2  1
```

How to interpret these kernels? In short, for each of the three color values for each pixel, you'll compute two values Gx and Gy. To compute Gx for the red channel value of a pixel, for instance, we'll take the original red values for the nine pixels that form a 3x3 box around the pixel, multiply them each by the corresponding value in the Gx kernel, and take the sum of the resulting values.

Why these particular values for the kernel? In the Gx direction, for instance, we're multiplying the pixels to the right of the target pixel by a positive number, and multiplying the pixels to the left of the target pixel by a negative number. When we take the sum, if the pixels on the right are a similar color to the pixels on the left, the result will be close to 0 (the numbers cancel out). But if the pixels on the right are very different from the pixels on the left, then the resulting value will be very positive or very negative, indicating a change in color that likely is the result of a boundary between objects. And a similar argument holds true for calculating edges in the y direction.

Using these kernels, we can generate a Gx and Gy value for each of the red, green, and blue channels for a pixel. But each channel can only take on one value, not two: so we need some way to combine Gx and Gy into a single value. The Sobel filter algorithm combines Gx and Gy into a final value by calculating the square root of $Gx^2 + Gy^2$. And since channel values can only take on integer values from 0 to 255, be sure the resulting value is rounded to the nearest integer and capped at 255!

And what about handling pixels at the edge, or in the corner of the image? There are many ways to handle pixels at the edge, but for the purposes of this problem, you can to treat the image as if there was a 1 pixel solid black border around the edge of the image: therefore, trying to access a pixel past the edge of the image should be treated as a solid black pixel (values of 0 for each of red, green, and blue). This will effectively ignore those pixels from our calculations of Gx and Gy.

Hint: If you are having trouble with colored images, try to use a grayscale one first.

```
find_edges(image)
```