CS 112 – Project 4 Multi-dimensional lists

Due Date: Sunday, Nov 3rd, 11:59pm

no extra credit for early submissions in this project

Project basics file:

- https://cs.gmu.edu/~marks/112/projects/project_basics.pdf

Needed file. A zip file containing all the tester files

- https://cs.gmu.edu/~marks/112/projects/P4.zip

Optional file included in zip for visualizing the inputs/outputs as images: `P4_helper.py`

Background

The purpose of this assignment is to practice building, inspecting, and modifying multi-dimensional lists effectively. This often requires nested for-loops, but not always. It involves thinking of the structure like an MxN matrix of labeled spots, each with a row and column index, or a higher dimensional structure like a 3D matrix that has indexes for height, width and depth. Multi-dimensional lists aren't conceptually more difficult than single-dimension lists, but in practice the nested loops, aliasing, and more complex traversals and interactions merit some extra practice and thought.

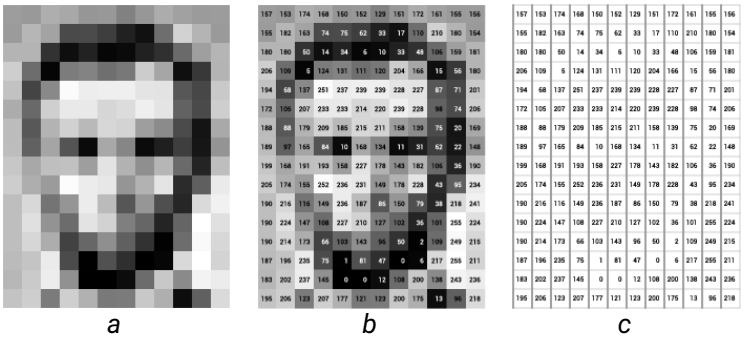
Scenario

We want to implement a set of functions that are very common in image processing tools and form the basis for all the advanced features one finds in tools like Photoshop, GIMP, etc.

Definitions

A digital image with dimensions *H*x*W* is represented with a matrix of pixels that has *H* rows and *W* columns. In Python we can store this matrix in a list of lists [[. . .], [. . .], [. . .] . . . . . [. . .]]. The length of the outer list equals the number of rows *H*, while the length of each nested list (it's the same for all of them) equals the number of columns *W*.

In grayscale images, each pixel is an integer that takes values from 0-255 (i.e. 1 byte). A value of zero corresponds to the black color and a value of 255 corresponds to the white color. Everything in between is a tone of gray; the darker the gray the closer to 0, the lighter the gray the closer to 255. Figure 1a depicts a zoom-in on a 16x12 grayscale image. In Figure 1b you can see the values of each pixel superimposed on the image. And in Figure 1c you can see the matrix representation of this grayscale image.



a *b* *c*

Figure 1 (grayscale image)

In color images, each pixel consists of a list of 3 integers that each takes values from 0-255. The first number corresponds to ***R**ed*, the second number corresponds to ***G**reen*, and the third number corresponds to ***B**lue*. Any color can be represented with the proper combination of the RGB values.

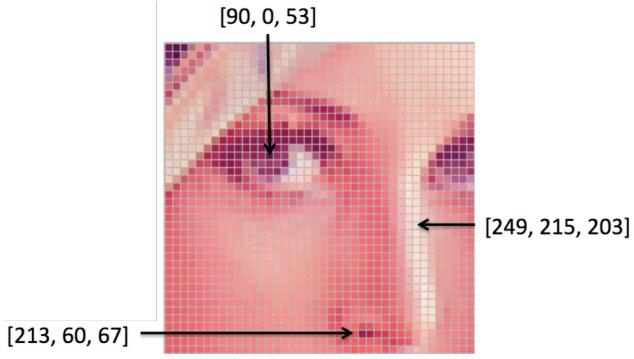


Figure 2 (color image)

Therefore, a color image in Python will be a list of lists of lists and will look like this:

[[. . . , [90,0,53], . . .] , . . . , [. . . , [249,215,203], . . .] , . . . , [. . . , [213,60,67], . . .], . . .]

Restrictions

- You are **not** allowed to `import` anything. It's ok to import modules for debugging purposes and/or for experimenting with the provided helper code but make sure you remove all imports when you submit your work.
- You are **not** allowed to use recursion, classes, exceptions, and list comprehensions
- From the built-in functions, you ***are*** allowed to use: `range()`, `len()`, `sorted()`, `int()`, `sum()`
- From list methods, you ***are*** allowed to use: `append()`, `extend()`, `insert()`, `copy()`, `sort()`, `remove()`, `pop()`
- You ***are*** allowed to use the `del` keyword

Assumptions

You may assume that:

- The types of the function arguments are the proper ones, you don't have to validate them.
- Image dimensions and the range of tones/colors are according to the specification, no validation is required

---

Testing

All the necessary files for testing your project have been **zipped** in one file. Once you download it, you must unzip it, and then work in it without moving or renaming its contents, otherwise the tester will not work. Tester inputs and outputs are stored in separate sub-directories, do not move or rename them either. Files in these sub-directories are numbered from 1 to 9. The larger the number the larger the image size you're testing. Keep in mind that debugging your code for larger inputs will not be trivial as the files are quite large and hard to visually inspect. Therefore, we have included some helper code that you can use to visualize as images the inputs and/or the outputs. You don't have to use it but it might be helpful if you have a hard time debugging the large tests. It is recommended that you **test each function independently** because some tests might be computationally intensive.

Other than that, this tester is still very similar to the previous ones. Be reminded that some of the test cases we'll be using for grading are omitted from the tester. This means that there might be errors in your code even if the tester is giving you no errors. You must do your own checks to make sure that you haven't missed anything and your code is correct. You do ***not*** need to modify the tester, just test on your own any way you like.

Grading Rubric

Correct submission & adequate commenting:	10	# see project basics file for more info
Tester cases:	75	# see project basics file for how to test!
Unknown test cases:	15	# you have to manually do your own checks
Extra credit:	5	# function `fill()` is extra credit work
TOTAL:	105	

Functions

The following are the functions you must implement. The signature of each one is provided, do ***not*** make any changes to them otherwise the tester will not work properly. Keep in mind that you do ***not*** have to write a main body for your program in this project.

`histogram(image)`

Description: It creates and returns a histogram of a grayscale image. A histogram provides a summary of the image by counting the number of pixels for each tonal value. The list of counters are ordered based on the pixel value (i.e. from 0 to 255). Input image should not be altered in any way.

Parameters: **`image`** (2D list of `int`) is the grayscale image

Return value: A list of `int` with 256 values

Example:

```
histogram([[1,1,2,2,5,2],
           [3,4,4,5,9,5]]) →    [0, 2, 3, 1, 2, 3, 0, 0, 0, 1, 0 . . . 0]
```

`flip(image, orientation)`

Description: It takes a grayscale image and flips its content either horizontally or vertically. No image is returned, all changes occur *in place*.

Parameters: **`image`** (a 2D list of `int`) is the input image, **`orientation`** (`string`) is either `'horizontal'` or `'vertical'`

Return value: `None`

Examples:

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

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

`rotate(image)`

Description: It takes a **square** grayscale image and rotates its content by 90 degrees clockwise. No image is returned, all changes occur *in place*.

Parameters: **`image`** (a 2D list of `int`) is the input image

Return value: `None`

Example:

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

`crop(image, origin, height, width)`

Description: It returns a cropped tile out of a color image without altering the input in any way.

Parameters: **`image`** (a 3D list of `int`) is the original image, **`origin`** (tuple of `int`) is a pair of coordinates (row, column) for top left corner where the cropping will begin, **`height`** (`int`) is the height of the cropped tile, **`width`** (`int`) is the width of the cropped tile

Return value: The cropped image that is a 3D list of `int`

Example:

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

`color2gray(image)`

Description: It takes a color image (3D list) and returns a new grayscale image (2D list) where each pixel will take the average value of the three color components truncated to integer. Input image should not be altered in any way.

Parameters: **`image`** (3D list of `int`) is the color image

Return value: The grayscale image that is a 2D list of `int`

Example:

```
color2gray([ [[1,1,2], [2,5,2]],              [[1, 3],
             [[3,4,4], [5,9,5]] ])   →          [3, 6]]
```

`extract_layer(image, color)`

Description: It extracts a single color out of the RGB values of the color image and returns this layer as a grayscale image. Input image should not be altered in any way.

Parameters: **`image`** (3D list of `int`) is the color image, **`color`** (`string`) is either `'red'` or `'green'` or `'blue'`

Return value: A 2D list of `int`

Example:

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

scale(image, factor)

Description: It scales a grayscale image by some **factor**. If the factor is positive it scales the image up by filling in all the extra pixels with the same value as the original. If the factor is negative it scales the image down by just deleting the extra pixels (in this case assume that the dimensions of the image are divided exactly by the **factor**). The scaling happens in place, nothing is returned.

Parameters: **image** (2D list of `int`) is the grayscale image, **factor** (`int`) is a number either greater than 1 or less than -1

Return value: `None`

Examples:

```
scale([ [1,7,2,5],              [ [1,1,7,7,2,2,5,5],
        [3,4,4,9] ], 2)    →      [1,1,7,7,2,2,5,5],
                                  [3,3,4,4,4,4,9,9],
                                  [3,3,4,4,4,4,9,9] ]

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

compress(image)

Description: It compresses a grayscale image and returns a data structure (list of lists) that holds a summary of the content, based on the following algorithm: First, we construct from the original image a black and white bitmap – values less than 128 are mapped to black and the rest are mapped to white. Then, we encode each row of this bitmap as a list containing the lengths of the alternating black and white sequences. Last, we package all these lists in an outer list. You can think of the encoding of each row as the process of counting the length of the consecutive black and white pixels (starting always with the black pixels) and adding these values in a list in the same order as we encounter the alternating black and white sequences. Obviously, each row, depending on its content, will have a different number of alternating sequences, so the full data structure will most probably be a list of variable-size sublists.

Parameters: **image** (2D list of `int`) is the grayscale image

Return value: A list of lists that summarizes the image

Example:

```
compress([[100,170,200,111,250,128],
          [223,244,255,115,127,105],
          [100,100,100,120,120,120]])    →    [ [1, 2, 1, 2], [0, 3, 3], [6] ]
```

`median_filter(image)`

Description: It applies a 3x3 median filter on a grayscale image and returns the filtered image as a new grayscale image. A median filter replaces the value of a certain pixel with the median value of the 9 pixels that belong to its 3x3 neighborhood. Because the border pixels don't have a full 3x3 neighborhood, the filtered image will have 2 fewer pixels in each dimension.

Parameters: **`image`** (2D list of `int`) is the grayscale image

Return value: A grayscale image (2D list of `int`)

Example:

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

Extra credit – the following function is optional and gives 5pts extra credit

This function is more challenging, but still only uses the same skills as before.

`fill(image, tone, seed)`

Description: It fills in a grayscale **`image`** with a certain **`tone`** of gray and returns the result as a new image. The input image should ***not*** be modified in any way. The filling starts at location defined by **`seed`** and spreads all around (check all 8 neighbors) until a different/boundary tone is hit. All the pixels that are replaced (i.e. filled-in) must have the same value as the initial pixel at **`seed`**, any other value is considered a boundary. The **`tone`** is a color that does ***not*** exist in the image before filling. You're ***not*** allowed to use recursion.

Parameters: **`image`** (2D list of `int`) is the grayscale image, **`tone`** (`int`) is the replacement tone of gray, **`seed`** (tuple of `int`) is a pair of coordinates (row, column)

Return value: A grayscale image (2D list of `int`)

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],], 2, (0,3))     →       [0,9,5,1,2,1]]
```