CSCI 1133, Fall 2019
Programming Assignment 6
Due: 11:55pm, Wednesday October 16, 2019

Due Date: Submit your solutions to GitHub by 11:55 p.m., Wednesday, October 16th. We will do a pull from this time point. Do not upload anything to Canvas and PLEASE be sure to use proper naming conventions for the file, classes, and functions. We will NOT change anything to run it using our scripts.

Unlike the computer lab exercises, this is not a collaborative assignment. You must design, implement, and test your code on your own without the assistance of anyone other than the course instructor or TAs. In addition, you may not include solutions or portions of solutions obtained from any source other than those provided in class (so you are ONLY allowed to reuse examples from the textbook, lectures, or code you and your partner write to solve lab problems). Otherwise obtaining or providing solutions to any homework problem for this class is considered Academic Misconduct. See the syllabus and read section "Academic Dishonesty" for information concerning cheating. Always feel free to ask the instructor or the TAs if you are unsure of something. They will be more than glad to answer any questions that you have. We want you to be successful and learn so give us the chance to help you.

Instructions: This assignment consists of 4 problems, worth a total of 40 points. Solve the problems below by yourself (unlike the labs, where you work collaboratively), and put all functions in a single file called `hw6.py`. Use the signatures given for each class and function. We will be calling your functions with our test cases so you must use the information provided. If you have questions, ask!

Because your homework file is submitted and tested electronically, the following are very important:

- You follow all naming conventions mentioned in this homework description.
- You submit the correct file, hw6.py, through Github by the due date deadline.
- You follow the example input and output formats shown.
- Regardless of how or where you develop your solutions, your programs should execute using the python3 command on CSELabs computers running the Linux operating system.

Push your work into Github under your own repo. The specific hosting directory should be: repo-<username>/hw6, where you replace <username> with your U of M user name. For instance, if your email address is bondx007@umn.edu, you should push your hw6 to this directory: repo-bondx007/hw6

The following will result in a score reduction equal to a percentage of the total possible points:

- Incorrectly named/submitted source file, functions, or classes (20%)
- Constraints not followed (40%)
- Failure to execute due to syntax errors (30%)

Image Manipulation

Every image rendered on a computer can fundamentally be represented as a three dimensional matrix. An image can be broken down into two dimensional array of pixels (short for "picture element"), which appear to be tiny squares of a single color that, when combined, make up the images you see on a computer screen. However, this isn't quite accurate: each pixel is actually displayed as light produced by three tiny light-emitting diodes: one red, one green, and one blue.

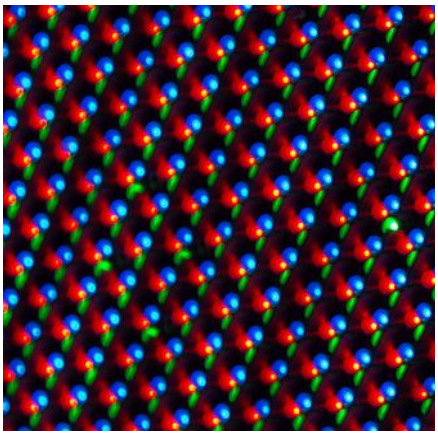


Image credit: https://en.wikipedia.org/wiki/LED_display

For this reason, pixel data is itself stored as a list of three elements, representing the relative strengths of the red, green, and blue diodes needed to produce the required color for the pixel; these are often given as numbers between 0 and 255. Since images are two dimensional arrays of pixels, and we can represent a pixel as a one-dimensional list of three integers, we can represent an image as a three dimensional matrix (that is, a list of lists of lists) of integers in Python. In the following problems, you will be manipulating matrices of this format in order to alter image files in various ways. You can see an example of how an image is represented as a 3D matrix below:

```
[[[0, 0, 255], [255, 255, 255]],
[[255, 0, 0], [0, 255, 0]],
[[0, 255, 255], [255, 128, 50]],
[[127, 127, 127], [0, 0, 0]]]
```

You may notice a few oddities in this representation. First, note that the pixels are listed starting with the bottom row, meaning that if you print out each row of the matrix on a different line as above, the layout almost matches the image, except that it's flipped vertically. Second, the channels (color components) for each pixel are listed in the order [blue, green, red] rather than what you may be familiar with, the standard RGB. The reason that we're choosing this representation is because it matches the order in which the data is listed within the .bmp files that we'll be using (see https://en.wikipedia.org/wiki/BMP_file_format for more info), but thankfully, for the tasks that you'll be completing, neither of these oddities really matter: none of the tasks you will be facing require you to know what order the pixels are listed in or what order the colors are within each pixel.

Download the template file `hw6.py` from Canvas, along with all of the example .bmp files (They are all in the compressed folder labelled `hw6.zip`). The template file has a function
`transform_image(fname, operation)`
which is designed to read in the bmp format image file at fname, convert the data into a pixel matrix as described above, perform an operation on it, and then convert that matrix back into .bmp file. In particular, the input file is assumed to be in the same directory as hw6.py, and the output file will be named the same as the input, except with `<operation>_` appended to the front. For example, `transform_image('cat.bmp', 'grayscale')` will produce a grayscale version of `cat.bmp`, called `grayscale_cat.bmp`.

This function is already working, so you do not need to edit it. However, the function itself doesn't actually perform any operations on the pixel matrix generated from the .bmp file; it instead calls on one of four unimplemented functions to do that work: `invert`, `grayscale`, `rotate`, and `edge_detect`. All of these functions take in a three dimensional matrix that represents an image, as described above, and should return another matrix representing the image with the specified operation performed. Your task in this assignment is to implement each operation.

Constraints:

- Do not import/use any Python modules except for `copy` (see part C hints).
- Do not alter the `transform_image` or `big_end_to_int` functions in the hw6.py template.
- Do not change the function names or arguments.
- Follow the instructions in the `hw6.py` template.
- Your submission should have no code outside of the function definitions (comments are fine).
- You are permitted to write helper functions outside of the ones provided in the template.
- You are not required to push any of the .bmp files to github: only the `hw6.py` file will be graded.

Problem A. (10 *points*) **Color Inversion**

First, you will be implementing the `invert` function. This performs a color inversion, as though you were producing a negative of a photo. Recall that each color component in every pixel is an integer between 0 and 255, inclusive. In every pixel, for each of the three color components, if the original value for the component was `x`, to invert the color you must set it to `255-x`.

Examples:

```
>>> invert([[[0, 0, 255], [255, 255, 255]],
[[255, 0, 0], [0, 255, 0]],
[[0, 255, 255], [255, 128, 50]],
[[127, 127, 127], [0, 0, 0]]])

[[[255, 255, 0], [0, 0, 0]],
[[0, 255, 255], [255, 0, 255]],
[[255, 0, 0], [0, 127, 205]],
[[128, 128, 128], [255, 255, 255]]]
```

(note that the highlighting and the new line for each row are for clarity purposes only and will not be present in the actual console output)

Below is the result of using `transform_image` with the `invert` operation on the following files found on Canvas, assuming that they are downloaded to the same folder as your `hw6.py` script:

```
>>> transform_image('dice.bmp','invert')
>>> transform_image('cat.bmp','invert')
>>> transform_image('connector.bmp','invert')
```

dice.bmp

(source: https://en.wikipedia.org/wiki/Dice)

invert_dice.bmp

cat.bmp

invert_cat.bmp

connector.bmp

invert_connector.bmp

Problem B. (10 *points*) **Grayscale**

Next, you will be implementing the `grayscale` function. This converts the image to grayscale (a black and white image). To do this, for every pixel, compute the average (mean) of the three color components, rounding down, and then set all three components to that value.

(Note: This isn't exactly the way real grayscale conversion works, but it's close enough. See https://en.wikipedia.org/wiki/Grayscale or https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion if you're interested in why this is technically wrong).

Examples:

```
>>> grayscale([[[0, 0, 255], [255, 255, 255]],
[[255, 0, 0], [0, 255, 0]],
[[0, 255, 255], [255, 128, 50]],
[[127, 127, 127], [0, 0, 0]]])

[[[85, 85, 85], [255, 255, 255]],
[[85, 85, 85], [85, 85, 85]],
[[170, 170, 170], [144, 144, 144]],
[[127, 127, 127], [0, 0, 0]]]
```

Below is the result of using `transform_image` with the `grayscale` operation on the following files found on Canvas, assuming that they are downloaded to the same folder as your `hw6.py` script:

```
>>> transform_image('dice.bmp','grayscale')
>>> transform_image('cat.bmp','grayscale')
>>> transform_image('connector.bmp','grayscale')
```

dice.bmp

grayscale_dice.bmp

cat.bmp

grayscale_cat.bmp

connector.bmp

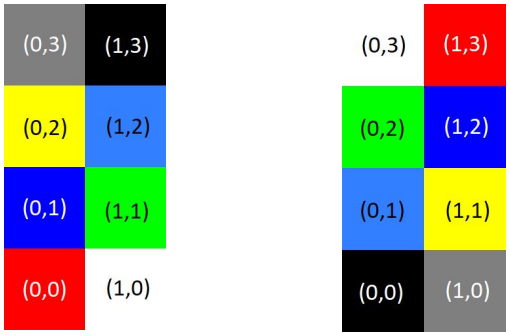
grayscale_connector.bmp

Problem C. (10 *points*) **Rotate**

Next, you will be implementing the `rotate` function. This rotates the image by 180 degrees (which is equivalent to reflecting it about both the horizontal axis and the vertical axis).

Hints:

- A pixel at location (x,y) in the original matrix should end up at location (width-x-1,height-y-1) in the output matrix, assuming that we start counting at 0 (see diagram below).
- For this problem and (especially) the next one, you may find it easier if you can create a copy of the pixel matrix to avoid the problem of overwriting pixel data in the original matrix before you're done using them to compute a pixel in the final matrix. Since this is a nested list structure, the easiest way to do this is to `import copy` and then use the `copy.deepcopy` method.



Examples:

```
>>> rotate([[[0, 0, 255], [255, 255, 255]],
[[255, 0, 0], [0, 255, 0]],
[[0, 255, 255], [255, 128, 50]],
[[127, 127, 127], [0, 0, 0]]])

[[[0, 0, 0], [127, 127, 127]],
[[255, 128, 50], [0, 255, 255]],
[[0, 255, 0], [255, 0, 0]],
[[255, 255, 255], [0, 0, 255]]]
```

Below is the result of using `transform_image` with the `rotate` operation on the following files found on Canvas, assuming that they are downloaded to the same folder as your `hw6.py` script:

```
>>> transform_image('dice.bmp','rotate')
>>> transform_image('cat.bmp','rotate')
>>> transform_image('connector.bmp','rotate')
```

dice.bmp

rotate_dice.bmp

cat.bmp

rotate_cat.bmp

connector.bmp

rotate_connector.bmp

Problem D. (10 *points*) **Edge Detection Filter**

Finally, you will be implementing the `edge_detect` function. This task is a bit more involved than any of the previous ones, so you will definitely want to write the function incrementally (that is, write some of the functionality, test that, and then only move on when you're sure the first part works).

We'll be using a pretty simple scheme for edge detection; we want to highlight pixels that are significantly brighter (in at least one of the color components) than their surrounding pixels. A sudden shift in brightness like this is bound to occur along any sharp visible edge. To accomplish this, we'll take each pixel, multiply its color values by 8, and then subtract the color values of the 8 pixels that surround it on the grid. Pixels that are very similar to their surrounding pixels will then end up with a color close to black, since 8 times the color component of that pixel minus the color components of its 8 neighbors should end up around 0.

This means that for a pixel located at (x,y) in the original image with original red color value r(x,y), you can compute the red component for the pixel the new image r'(x,y) with the following formula:

r'(x,y) = 8*r(x,y) - r(x-1,y-1) - r(x,y-1) - r(x+1,y-1) - r(x-1,y) - r(x+1,y) - r(x-1,y+1) - r(x,y+1) - r(x+1,y+1)

A similar formula applies for the green and blue components. Note that this may produce a value that is above the maximum color component value of 255: if this occurs then you should just set the value to 255. Similarly, if the formula generates a negative number, just set the value to 0. For pixels along the boundaries of the picture that do not have 8 neighbors, ignore the formula and set their pixel values to black ([0, 0, 0]), to avoid the issue of going out of bounds.

Hints:

- See the previous problem's hint about creating a copy of pixel matrix so that you don't have to worry about overwriting values before you can use them to compute their neighbors.
- You may want to consider writing a helper function that computes the new pixel value for a single pixel, and then use that function in a nested loop inside `edge_detect`. This will make it easier to test whether that part of your implementation works before moving on.

Examples:

```
>>> edge_detect([[[0, 0, 255], [255, 255, 255]],
[[255, 0, 0], [0, 255, 0]],
[[0, 255, 255], [255, 128, 50]],
[[127, 127, 127], [0, 0, 0]]])

[[[0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [0, 0, 0]]]
```

(Note: none of the above pixels have 8 neighbors, so they are all set to black)

```
>>> edge_detect(
              [[[47,198,255],[47,198,255],[47,198,255],[47,198,255]],
               [[47,198,255],[47,198,255],[47,198,255],[47,198,255]],
               [[47,198,255],[47,198,255],[47,198,255],[131,38,79]],
               [[47,198,255],[131,38,79],[131,38,79],[131,38,79]],
               [[131,38,79],[131,38,79],[131,38,79],[131,38,79]],
               [[131,38,79],[131,38,79],[131,38,79],[131,38,79]]])

[[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [0, 0, 0], [0, 160, 176], [0, 0, 0]],
[[0, 0, 0], [0, 255, 255], [0, 255, 255], [0, 0, 0]],
[[0, 0, 0], [255, 0, 0], [168, 0, 0], [0, 0, 0]],
[[0, 0, 0], [84, 0, 0], [0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]]


>>> edge_detect([[[0,0,10], [0,0,20], [0,0,30], [0,0,200]],
     [[255,120,20], [70,120,30], [0,120,40], [0,0,60]],
     [[0,120,30], [255,120,40], [0,120,60], [0,120,50]],
     [[255,120,40], [255,120,60], [255,120,50], [255,120,40]]])


[[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [50, 255, 0], [0, 255, 0], [0, 0, 0]],
[[0, 0, 0], [255, 0, 0], [0, 120, 110], [0, 0, 0]],
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]]
```

Below is the result of using `transform_image` with the `edge_detect` operation on the following files found on Canvas, assuming that they are downloaded to the same folder as your `hw6.py` script. Note that these may take significantly longer to run than the previous functions, due to the complexity of the operation (for reference, each required about 15 seconds on my laptop).

```
>>> transform_image('dice.bmp','edge_detect')
>>> transform_image('cat.bmp','edge_detect')
>>> transform_image('connector.bmp','edge_detect')
```

dice.bmp

edge_detect_dice.bmp

cat.bmp

edge_detect_cat.bmp

connector.bmp

edge_detect_connector.bmp