Drill Problem: #1

Function Name: checkImage

Inputs:

1. (*char*) The name of an image file
2. (*char*) The name of a second image file

Outputs:

1. (*char*) A sentence comparing the two images

File Outputs (Potential):

1. A black and white image file of where the two input images differ, named: `'<image1>VS<image2>.png'`

Function Description:

Given two images, write a function called `checkImage()` that determines if the two images are the same, or if, and how, they are different.

- If the two images are completely identical, the output should read `'The images are the same.'`
- If the two images do not have the same dimensions, the output should read `'The images have different dimensions.'`
- If the two images have the same dimensions, but have different color values at the same pixel, the output should read `'The RGB values are different.'` Additionally, if the two images have different colors, you should write a new image that is white everywhere the two images have the same RGB values and black everywhere they are different. This new image should be called `'<image1>VS<image2>.png'`. For example, if you were comparing `'lilacs.png'` and `'roses.png'`, you would call your new image `'lilacsVSroses.png'`.

Notes:

- You may find the function `imwrite()` useful.
- It is not necessary to use iteration for pixel-by-pixel comparisons. This will cause your code to run very slowly and is inefficient.
- Use a 3-layer uint8 array to write the output image.
- This function should be useful for the rest of your homework.

Drill Problem: #2

Function Name: colorScreen

Inputs:

1. (*char*) A filename of a foreground image
2. (*char*) A filename of a background image
3. (*double*) A vector of length three of RGB values to replace

Outputs:

1. (*uint8*) A uint8 array with the `colorScreen` effect applied

Function Description:

Given two images, write a function called `colorScreen` that will replace all pixels in the first image that match a given RGB calue with the corresponding pixels in the second image. For example, if the following is called:

```
colorScreen('img1.png', 'img2.png', [0 255 0]);
```

A uint8 array should be returned that looks like `'img1.png'`, but with all of the pure green pixels replaced with the corresponding pixels in `'img2.png'`.

Notes:

- You should not have to use iteration for this problem; iteration is slow and inefficient for images due to the size of the images to be worked with. Consider logical indexing instead.
- The two input images will always be the same size.

Drill Problem: #3

Function Name: hashtagNoFilter

Inputs:

1. (*char*) A string identifying the “top” image file
2. (*char*) A string identifying the “bottom” image file
3. (*char*) A string describing the blend mode to apply to the images

Outputs:

(*none*)

File Outputs:

1. Image file of the colored-in input file

Function Description:

If you’ve ever edited a picture before, you’ve probably used blend modes, whether you knew it or not! Now you’re going to get up close and personal with blend modes and learn how software like Photoshop is able to combine images to make cool effects. If you have no idea what a blend mode is, never fear, here’s a web page with all the info on them you could ever want: http://photoblogstop.com/photoshop/photoshop-blend-modes-explained.
But the TL;DR version is: a blend mode is a method for combining the pixels of 2 images to produce a new image.

You’ll be coding 3 blend modes, namely Screen, Multiply and Overlay. For this assignment, we will use Photoshop’s equations for these modes. First, you need to “normalize” the image, which means changing the RGB values from integers between 0 and 255 to doubles between 0 and 1 (0 maps to 0, 255 maps to 1). You will use these normalized values to create the blends, then convert back to `uint8()` values between 0 and 255 at the end. Here’s how each mode works:

- `'multiply'`: This is the most straightforward mode; you simply take the corresponding normalized values for each color channel and multiply them together to get the output pixels.

```
outputChannel = topChannel.*bottomChannel
```

- `'screen'`: You will create each output color channel with the following equation:

```
outputChannel = 1 - ((1-topChannel).*(1-bottomChannel))
```

  In English, this blend mode says: “Invert the top and bottom channels, multiply them, and then invert that.”

- `'overlay'`: Overlay combines Multiply and Screen in the following way:

$$\texttt{outputChannel} = \begin{cases} 2.*\texttt{topChannel}.*\texttt{bottomChannel}, & \text{if luminance} < 0.5 \\ 1-(2.*(1-\texttt{topChannel}).*(1-\texttt{bottomChannel})), & \text{otherwise} \end{cases}$$

where the “luminance” is the normalized luminance of the pixels in the bottom channel. The luminance of a pixel is defined as:

```
luminance = 0.3*R + 0.59*G + 0.11*B
```

where R, G, and B are the red, green and blue layer values for that pixel. Note that these values are capped between 0 and 255; normalized luminance is these values mapped to numbers between 0 and 1.

So Overlay says: wherever the normalized luminance of the pixels in the bottom channel is less than 0.5, then apply 2 times multiply to those pixels. Wherever the normalized luminance of these pixels is not less than 0.5, apply screen, with an additional factor of 2 thrown in.

The new image should be output to a file named:
`'<top image name>_<bottom image name>_<blend mode>.png'`

Notes:

- You should not have to iterate though the images at all; instead, use masking and `.*`
- Because Overlay combines Screen and Multiply, you may want to implement those modes in helper functions for easy reuse.
- The input blend mode is case-sensitive.
- You are guaranteed that both input images will be the same size.

Hints:

- Calculate the luminance with RGB values between 0 and 255 and then normalize the values to between 0 and 1.

Drill Problem: #4

Function Name: coloringBook

Inputs:

1. (*char*) A string identifying the name of an image file
2. (*char*) A string identifying the name of an MS Excel file

Outputs:

(*none*)

File Outputs:

- A image file created using the instructions below

Function Description:

As a kid (or adult), have you tried to color in drawings before, but all you have is an unsatisfying box of 10-color crayons? Like, you know that tree is not really blue, but blue is the closest color that you got?! Those were the good ol' days! Let's recreate those good ol' days with MATLAB, shall we?

Given an image file, match each pixel of the image file to the closest color we have in our unsatisfying box of crayons. Then, replace that pixel with the color that is determined to be the closest color. You should append `'_retro'` to the output image file name, while retaining the correct extension, so that the new image file name is formatted like `'<filename>_retro.<originalExtension>'`. The size of the image should remain unchanged.

Determining the closest color match:

For this problem, we will take the average color of a 2x2 pixel block, then compute the distance between the colors in the image and the colors available for our use. The formula is as follows (distance formula in 3-dimension):

$$Dist = \sqrt{(Rc - Ri)^2 + (Gc - Gi)^2 + (Bc - Bi)^2}$$

- ***Rc*** – Red value of the crayon color
- ***Ri*** – Red value of the image pixel
- ***Gc*** – Green value of the crayon color
- ***Gi*** – Green value of the image pixel
- ***Bc*** – Blue value of the crayon color
- ***Bi*** – Blue value of the image pixel

Then, replace the color of the 2x2 pixel block with the available color in the excel file that would minimize the distance.

Drill Problem: #4

Example:

- Inputs:
  1. `'20pack.xlsx'`
  2. `'mufasaAndSimba.png'`

- Output:
  1. `'mufasaAndSimba_retro.png'`

Notes:

- The image is guaranteed to have an even number of rows and columns.
- The first column in the excel file is guaranteed to be the name of the color. The second, third, and fourth columns are guaranteed to be the red, green, and blue RGB values of that color, respectively. You are not guaranteed to have the exact colors given in the excel file test case, or a given number of crayon colors in the excel file.

Drill Problem: #5

Function Name: mosaic

Inputs:

1. (*char*) A string identifying an image file name
2. (*double*) The number, N, of rows and columns into which to divide the output image

Outputs:

(*none*)

File Outputs:

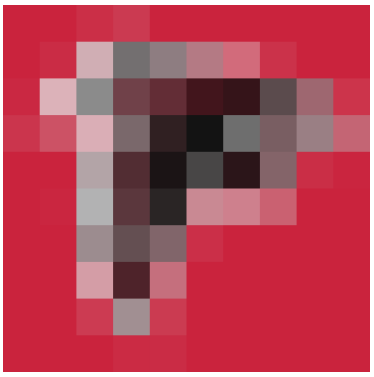
1. An image file of the mosaic of the input file, named:
   <original image name><second input>x<second input>.<original image extension>

Function Description:

Pictures of pictures, averages of averages—its' like image-ception! Write a function called `mosaic` that takes in an image file and outputs a modified image file, a mosaic of the original image, where each 'tile' is a smaller, shaded version of the original image. The shading of each tile comes from averaging the tile-sized (re-sized) original image with the average RGB value of the section of the original (originally sized) image that tile will replace. The number of tiles in the output image is defined by the second input, N, such that there are NxN total tiles.

As an example, consider the Falcons logo, `'falcons.png'` of size 500x500 pixels. For a second input of 4, each tile would be the `'falcons.png'` image scaled down to 125x125 pixels and tinted according to its location. The upper left tile would be tinted red from the RGB values of the tile-sized image being averaged with the mean R, G, and B values of the top-left 125x125 section of the original image: in this case a red color. The tile in the second row, third column would be tinted dark gray from the tile-sized image being averaged with the corresponding 125x125 segment of the original image: in this case a combination of predominantly black pixels with some white, gray, and red. The rest of the image tiles would continue along this algorithm. The new image file name would be: `'falcons4x4.png'`. Note that as the number of rows and columns of tiles is increased from 4 to 10 to 20 to 50 in the examples provided, the overall mosaic gains definition, while the individual tiles become less recognizable.

Falcons Image



10x10 Shading

10x10 Shading with Tiles

Drill Problem: #5

Notes:

- You are guaranteed to have an image of a size evenly divisible by the target number of rows and columns, though the image may not always be square.
- You can check your output image against the solution image using your `checkImage()` function.

Hints:

- Converting your image(s) to class `'double'` (and then converting back to class `'uint8'` before writing with `imwrite()`) may be helpful in calculating averages.
- Iterating through each section of the image to be replaced with a tile may be helpful.

Drill Problem: #6

Function Name: wheresWaldo

Inputs:

1. (*char*) A string identifying an image file
2. (*char*) A string identifying the image file containing Waldo

Outputs:

(*none*)

File Outputs:

1. A image file of the in input file with Waldo in color and the rest in grey

Function Description:

Remember Waldo? He's that guy who you found throughout your childhood and a couple of weeks ago in that awesome homework problem. Well your favorite friend is back again, but this time you are going to look through an actual image for Waldo!

You will be given a large image that has Waldo somewhere in it (first input), and a small sub-image of Waldo cut out from the larger image (second input). You need to find where the second image occurs within the first. Essentially this means you need to iterate through the larger image and look for where your smaller photo of Waldo fits. After you find where Waldo fits into the picture, make the rest of the image greyscale. This means that the only part of the larger image that should be in color is the sub-image containing Waldo.

There's a catch, however. Due to image compression artifacts, the sub image doesn't *exactly* appear in the larger image. Instead, you will have to find a close match. What is a close match, you ask? For this problem, a close match is when the maximum absolute difference in grayscale value between the sub-image and image is less than or equal to 100.

The name of the image file you output should be `'<filename>_foundhim.png'` which means if the larger image was named `'themepark.png'` your output image should be `'themepark_foundhim.png'`.

Notes:

- The solution code will take 15-30s to run since it (potentially) has to iterate through the entire image.
- A helper function may be useful to speed up the process or greyscale the image.
- Be efficient in how you code. Iterating through an image is already a computationally taxing process, so try not to do any unnecessary steps. For example, once you find a match in the image, do not keep iterating!

Drill Problem: EXTRA CREDIT

-THIS PROBLEM IS EXTRA CREDIT!-

Function Name: colorByNum

Inputs:

1. (*char*) A string identifying an image file
2. (*cell array*) A cell array of colors. Each index will contain a 1x3 vector of RGB values

Outputs:

(*none*)

File Outputs:

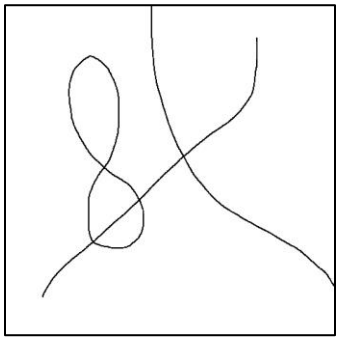
1. A image file of the colored in input file

Function Description:

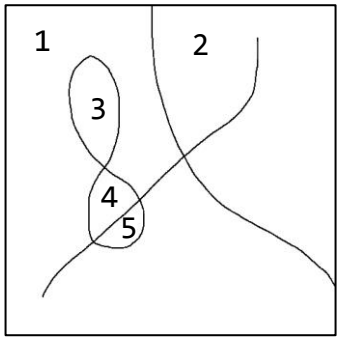
Remember in Kindergarten when you had to color in a picture by number? Yeah, turns out younger you was actually doing some complex computations to figure out what regions to color in with which color, as you will quickly see when you try to tell a computer how to do it. You will be making a function that takes a grayscale image that is mostly white with black lines and colors each region with the corresponding colors in the cell array of colors provided in the second input. A grayscale image is simply an image where the R, G and B layers are identical (i.e. you can ignore all but the first layer).

- **You should assign colors to each region in "reading order." That is, left-to-right, top-to-bottom (see images below).**
- A pixel is considered white if its grayscale value is greater than or equal to 240. Everything else is considered a line.
- You may have more regions than colors, in which case you should wrap back around to the first color.

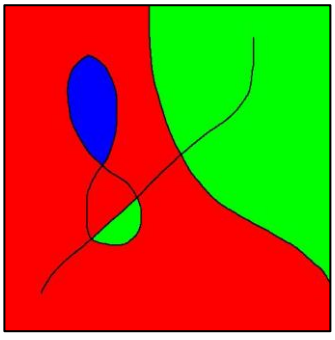
Given this image: ('bigTest.jpg')



The 'numbers' that should be assigned to each region are:



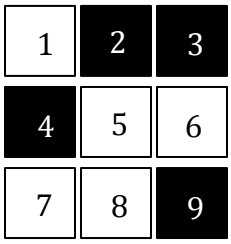
And the output image should be:



If the input colors were red, green, blue, respectively.

Drill Problem: EXTRA CREDIT

You can also open up the actual image files or run the solution code to take a closer look at what's going on. Also, note that you don't have to worry about the same color filling two adjacent regions.
Your output filename should be the same as the input file, with '_colored.jpg' appended to the end.
A white pixel is defined to be in the same region as another white pixel if they are adjacent along an edge.



If the input image was the 9 pixels to the left, pixels 2,3,4 and 9 would be lines. Pixel 1 would compose region 1. Pixels 5,6,7, and 8 would compose region 2. The fact that pixel 1 and 5 are touching on a corner does not make them part of the same region.

Notes:

- You will probably want to spend some time thinking through different strategies for solving the problem before you start coding.
- Please, please, please use helper functions!
- smallTest.jpg is a 25x25 pixel image. This is small enough that you can look at the image array as you go along to see exactly what your function is doing (helpful for debugging).
- You are guaranteed to have .jpg image inputs.
- If you look closely at the solution images, you will see slight variations in color within regions. These are just .jpg compression artifacts and you don't need to worry about them. If you run the solution functions and look at the image output, those artifacts aren't there.
- Be as efficient as possible when writing your function. If you aren't it could take a very long time to run.
- You may **not** use the `bwconncomp()`, `conv()`, `conv2()`, `bwlabel()`, `bwlabeln()`, `label2rgb()`, `labelmatrix()` `filter()`, `filter2()` or `imfilter()` functions.

Hints:

These hints start off mild and get progressively more hinty. So read as far as you want depending on how much you want to figure out for yourself. Also, there are many ways to solve this problem and these hints are only for one way.

- Think about ways to determine when you move from one region to the next. And think about ways to determine if you have already been in a region.
- Think about ways to combine iteration and masking to cut down on the amount of iteration you have to do. (If you ever find yourself iterating through the entire image multiple times, you've taken a wrong turn)
- You may want to just keep track of region numbers (1, 2, 3 …) at first, and then replace the numbers with their corresponding colors at the end.

Drill Problem: EXTRA CREDIT

- Given a single pixel in the image, how could you find all other pixels in that region?

Super Hints:

- Given a single pixel, iteratively look at adjacent pixels to find the rest of the region
- If you can do that, then you only need to find one pixel in each region (in order) and you can identify all the pixels in each region