Notice

This homework requires you to create images, which can be difficult to check using `isequal` or the file comparison tool. To mitigate this, we have given you a function called `checkImage` that will compare two images. You can look at `help checkImage` to see how to use the function.

Also, all output images should be saved as `.png` images.

Happy coding,
~Homework Team

Function Name: `cartoonize`

Inputs:

1. (*char*) the filename of the image to convert to a cartoon, including the file extension
2. (*double*) the factor of cartoonization

Outputs:

None

Image File Outputs:

1. An image file containing the original and new "cartooned" image side-by-side, named `'cartoon_'<filename>`

Background:

Our neighbors at Turner Broadcasting need help from a MATLAB-wiz Georgia Tech student. They had completed all filming for a new sitcom but, at the last second, they decided to make it a cartoon instead of a live-action show! You will convert the input images to cartoons and create a side-by-side comparison image.

Function Description:

To make the cartoon image, all RGB values in the image array within the range of the input factor will be rounded to the same number. This is achieved by dividing all RGB values by the factor, rounding to a whole number, and scaling (multiplying) back up by that factor.

For example, the numbers `[ 10, 20, 40, 90 ]` brought to the closest number with a factor of `40` would be `[ 0, 40, 40, 80 ]`. This will reduce gradient patterns in the image and form a cartoon-like effect.

Concatenate the new cartoon image onto the right side of the original image. To name your image output, prepend `'cartoon_'` to the front of the original filename.

Notes:

- Remember to convert to double when conducting math on your array. After the calculations, convert your array back to `uint8`

Function Name: `popArt`

Inputs:

1. (*char*) the filename of an image to convert to pop art, including the file extension

Outputs:

none

Image Outputs:

1. A file containing the finished pop art image, named the original filename before the file extension with `'_pop.png'` appended

Banned Functions:

`rgb2gray(), mat2gray(), ind2gray(), flip(), fliplr()`

Background:

You've decided to expand on your skills to include some unconventional artistic ones. Taking after Andy Warhol, you take normal images and turn them into some funky pop art to provide your dorm room with some eccentric wall decorations.

Function Description:

The output image will have twice the number of columns and twice the number of rows as the original. The output image will contain these four versions of the original image:

1. Top right: the negative of the original image
2. Top left: the mirror image of the original image, with the red and blue layers swapped
3. Bottom left: the grayscale of the original image
4. Bottom right: the image rotated 90 degrees.

The name of the output image should be the original name before the file extensions with `'_pop.png'` appended.

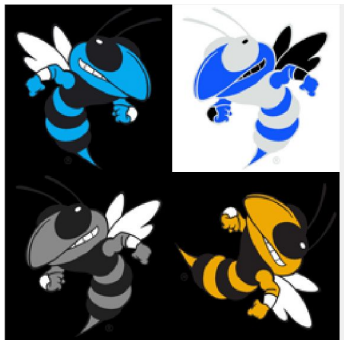
Continued...

Example:

`buzz.png`

Original

`buzz_pop.png`



Output

Notes:

- To find the grayscale of the image, take the average of the three color values for each pixel. Remember to convert from uint8 to double before doing calculations and back to uint8 after the calculations.
- The input images are guaranteed to be square.

Function Name: `narutoRun`

Inputs:

1. (*char*) the filename of an image containing the runner, including a file extension
2. (*char*) the filename of a location background image, including a file extension

Outputs:

None

Image Outputs:

1. A file containing the edited image, titled `'RunAcross'<location>'NarutoStyle.png'`

Background:

It's been a few weeks since the famous Run Across Tech Green Naruto Style event, and all this midterm pent-up energy has you itching to go again. The trouble is, not everyone sees your vision to have daily Naruto Runs. Therefore you must simulate those daily events by placing an image of someone running like Naruto on various campus background images. They will see; they will join you.

Function Description:

In order to place your running image on the campus image, you must replace the background color of the running image with the corresponding pixels in an image of campus. If the images are different sizes, you should resize the campus picture to be the same size as the running picture before performing the change.

To determine the background color, find the R,G, and B values of the top-left pixel of the running image. Once you know these values, you can replace the pixels of that color in the running image with the corresponding pixels from the campus image.

The new image should be saved to `'RunAcross'<location>'NarutoStyle.png'` where `<location>` is the second input (filename) to the function, excluding the file extension.

Hints:

- The `imresize()` function will be useful.

Function Name: `looneyTunes`

Inputs:

1. (*char*) the name of the song
2. (*double*) the number of pages for each music piece
3. *(double)* the number of instrumental parts in each song

Outputs:

none

Image Outputs:

1. Files containing an image for the music of each instrumental part. The number of files outputted will be equal to the number of instrumental parts in each song, given by input 3

Background:

The Georgia Tech Orchestra needs help making copies for all their instruments from the conductor's score, which is a sheet of paper with all of the instruments' parts shown. By taking out each instrument's part from all the pages of the score and putting them together, you will consolidate all music for each instrument.

Function Description:

Each song has a certain number of pages (input 2) and number of instruments playing on that piece (input 3). The name of each song is given in the first input. To find the actual filenames of the pages of music, append '_page#', starting at '_page1' and continuing until you have the file names for pages 1 through the number of pages (input 2). After '_page#', append the file extension '.png'. The filenames will look like '<song>_page1.png', '<song>_page2.png', and so on.

As you go through all of the pages, you will need to break each page into the correct number of sections (input 3). The section for each instrument part on each page should be concatenated vertically with the the sections for that instrument part from the other pages. Finally, save each instrumental part in the order that they were on the page (top to bottom) as `'<songName> _partX.png'`. See the test cases for a visual example.

Notes:

- All pages of the same song are the exact same size (so you do not need imresize() in this problem).
- Each instrument will be (roughly) evenly spaced on the pages.
- When rounding rows to index out of each image, round as a last step after all other calculations and do not round intermediate steps.

Continued...

Hints:

- You will need to calculate the row at which to start indexing each section and the row at which to stop indexing each section. To find the index at which each section ends, divide the number of the current part by the total number of parts and multiply by the number of rows.
- Some test cases have many file outputs, so be sure to check them all against the solution file outputs.

Function Name: `pokemonSNAP`

Inputs:

1. *(char)* Filename of image with a pokémon (`'<filename>_pokemon.png'`)
2. *(char)* Filename of image without the pokémon (`'<filename>.png'`)
3. *(struct)* `nationalPokedex` structure array

Outputs:

none

File Outputs:

1. Image of the pokémon captured (`'<filename>_captured.png'`)

Background:

Oh snap! It's `pokemonSNAP`! Hopefully this will give you flashbacks to the greatest N64 game of all time.

Function Description:

Write a function called `pokemonSNAP` that takes in an image with a pokémon. The image input will always be a square image. You will also be provided with the image without the pokémon (which is same size at the image with the pokémon). Your function should be able to identify which pokemon is present, and determine the rarity of the pokemon in order to decide which pokéball to use to capture it!

You are also provided with a variable called `nationalPokedex` (a structure array) and the `pokedex()` function, which will help identify your pokémon. The inputs for the `pokedex()` function should be a square image stored as an `NxNx3 uint8` array and the `nationalPokedex` structure array. The output of `pokedex()` could be in one of 2 formats:

`'No pokemon detected.'` OR `'<pokemon name>, the <description> pokemon.'`

Use these two resources to figure out which pokémon exists in the image.

`nationalPokedex` is a `1x40` structure array (each structure contains a unique pokémon) with 4 fields: `pokemon` (name of the pokémon as a character vector), `image` (an `NxNx3 uint8` array of the pokémon image), `rarity` (a character vector representing the rarity) and `description` (used for the output of pokédex). Once you figure out the rarity of the pokemon in the image, determine which pokéball should be used.

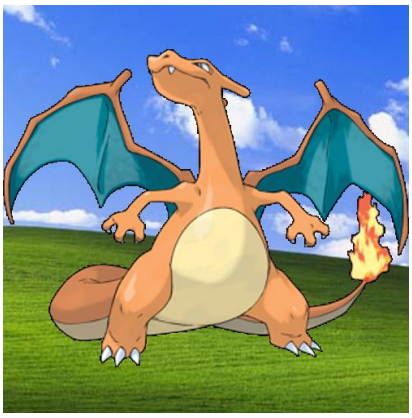
You will be provided with the images of 4 different pokeballs: `'pokeball.png'`, `'greatball.png'`, `'ultraball.png'` and `'masterball.png'`; `'common'` pokemon should be caught with a pokeball, `'uncommon'` with a greatball, `'rare'` with an ultraball and `'legendary'` with a masterball. Output the original image with the pokemon captured in the pokeball you determined best to use. The name of the output image should be formatted as:

`'<filename>_captured.png'`

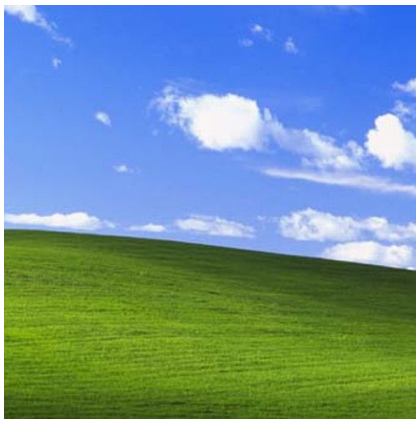
Continued...

Example:
Given the following input images:

`field_pokemon.png (500x500)`



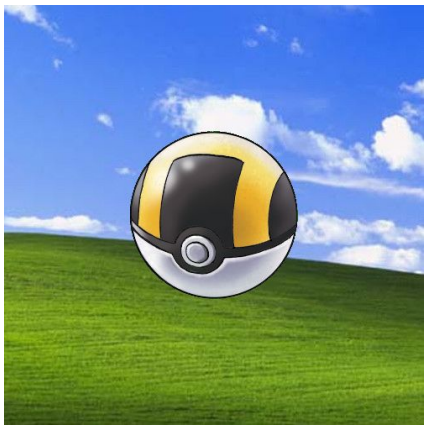
`field.png (500x500)`



```
>> pokemonSNAP('field_pokemon.png', 'field.png', nationalPokedex)
```

The following output image should be produced:

`field_captured.png (500x500)`



Notes:

- The third input to your function should always be the variable `nationalPokedex` (provided in the `nationalPokedex.mat` file).
- The two input images will always be square images of the same size.
- Pokeball images are all square and will always have a perfectly green background.
- You will need to use the `imresize()` function. When you do, use `'nearest'` as the third input.
- All first input images will contain a pokemon.

Hints:

- Take a look the `nationalPokedex` structure array to see how everything is stored.
- What are the RGB values of a perfectly green pixel?

Extra Credit

Function Name: shatteredScreen

Inputs:

1. (*cell*) A 1xM cell array containing the fragments of the images
2. (*char*) The name of the new image

Outputs:

(none)

Image Outputs:

1. A picture with the name given as an input with '.png' appended

Background:

It is friday night. You were done with all your work and were just laying back on your bed to watch some cartoons on your computer. All of a sudden your roommate comes into your room with three brooms strapped to one arm and a two-by-four bandaged to the other! *THWACK*

Now your screen is broken. Luckily, you were able to catch every single rectangular piece into a cell array. If only there were a handy MATLAB function that could put it back together…

Function Description:

Each cell contains an NxPx3 uint8 array that is a piece of the final image. There will always be at least two elements in the cell array that can be concatenated together to form a larger element in the array. In order to see if two elements can be put together compare a side on piece one to the corresponding side on piece two, if they are the same, the two can be be put together in the correct arrangement. You do NOT have to rotate any pieces to fit them together, they are all in the same orientation. The possible side comparisons you can do are as follows:

Piece	Legal comparisons			
Piece 1's side	Left	Top	Right	Bottom
Piece 2's side	Right	Bottom	Left	Top

Once you have created the full picture, write it to the given name concatenated with `'.png'`.

Notes:

- Think about writing this with multiple nested loops.
- You do not have to put every piece together your first loop through.
- If the sides of two image elements match up, the RGB values along that edge will be equal to the RGB values along the other edge. For example, if left side of piece 1

matches with the right side of piece 2, then the first column of RGB values in piece 1 will be equal to the last column of RGB values of piece 2.