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, even if the input images are of another image type (such as .jpg).

Happy coding,
~Homework Team

Function Name: `maskingTape`

Inputs:

1. *(char)* Filename of the image of your unpainted wall
2. *(char)* Filename of the image showing what you want your wall to look like
3. *(double)* 1x3 vector representing the RGB values of the masking tape color

Outputs:

(none)

File Outputs:

1. Image of masking tape applied to original image

Function Description:

You are thinking of sprucing up your dorm room by painting your walls, and decide to use MATLAB to help you out! You are given an image of an unpainted wall, and an image of what you want the painted wall to look like. Based on these two images, determine where masking tape must be applied to the unpainted wall so that the appropriate areas remain unpainted.

Create a new image showing what the unpainted wall looks like with masking tape applied. The color of the masking tape is given by the third input, with the three values in the vector representing its red, green, and blue values, respectively. The new file name should be the same as the first input, with `'_tape'` appended before the file extension.

Notes:

- The painted wall may be painted with any assortment of colors.
- Any part of the painted wall that is the same color as the unpainted wall will be covered in masking tape.

Function Name: `hiddenSound`

Inputs:

1. *(char)* The filename of an image

Outputs:

1. *(double)* An Nx2 array of the sound found hidden in the image

Function Description:

A picture is worth a thousand words, and is also worth a snippet of sound! In this problem, you will be finding hidden sounds in images. The red, blue, and green sets of pixels, when linearized, each contain special information. Red pixels contain the left audio channel, green pixels contain the right audio channel, and blue pixels contain a scale factor to multiply to the corresponding entries in both audio channels. If a pixel has a zero scale factor, remove the corresponding entries in the right and left audio channels. To keep the dimension the same, you will also need to remove the zeros from the scale factor vector.

Audio signals use positive and negative values to make sound, whereas images only use positive values. To handle this, subtract 128 from the red and green pixels. Then divide those pixels by 128 to get the real right and left audio channels, and finally multiply by the scale factor divided by 256 to get the scaled right and left audio channels.

Notes:

- The left (red) audio channel should be the first column of the output and the right (green) channel should be the second.
- Linearize means to turn an array into a single column vector, starting with the first column and concatenating subsequent columns below.
- Round you audio channels to 4 decimal places.
- Try calling `sound(<output array>, 44100)` on the result.

Function Name: `memeGenerator`

Inputs:

1. *(char)* Top text
2. *(char)* Bottom text
3. *(char)* Name of the original image
4. *(double)* 1x3 RGB triplet representing text color

Outputs:

(none)

File Outputs:

1. A dank meme

Function Description:

Memes are the bread and butter for online communication and you've decided to expedite the process of creating your own memes by writing a MATLAB function to do it for you! Using the helper function `str2img` (provided) to convert a string into an image of the text, you will resize the text-image to fit the width of the original image and place it onto the original image. When resizing, round the new row dimension to the nearest integer.

The top text (first input) should be placed across the top of the image and the bottom text (second input) should be placed across the bottom of the image. The image produced by `str2img` will use the specified color as the text color and the background will be black. When putting the text on the image, the black background should be replaced with the corresponding pixels on the base image.

The output filename should be the same as the input image name with `'_meme'` appended to the end.

Notes:

- Both the top and bottom text should only be one line.
- You should maintain the original text image proportions when scaling to fit the image.
- You must use `imresize` to resize the text.
- When using `imresize`, use the "nearest" method (see `help imresize`).
- See `help str2img` for details on how to use this function.

Function Name: `microscope`

Inputs:

1. *(char)* The name of a microscope image
2. *(double)* The 'scale' represented by the scale bar

Outputs:

1. *(double)* A 1x2 vector for the dimensions of the microscope image

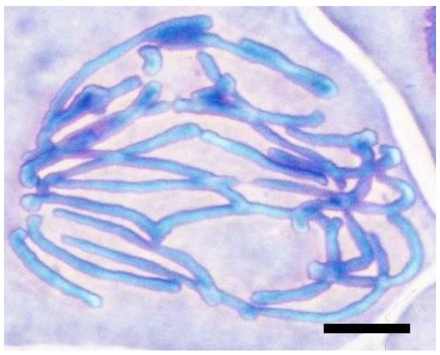
Banned Functions:

`rgb2gray`, `im2bw`

Function Description:

Published microscope images often have a black scale bar to indicate the scale of the image. Write a function called `microscope` which will determine the actual dimensions of the image based on the the relative size of the scale bar to the entire image. The second input to the function is the actual unitless distance represented by the scale bar.

For example, given the 400x500 image example.png with a scale bar length of 100 pixels,



example.png (400x500)

`[dim] = microscope('example.png', 50)` should return `[200 250]`, since the height of the image can fit 4 scale bars (200 units of length in total), and the width of the image can fit 5 scale bars (250 units of length in total). You should round your final output to the nearest integer.

Notes:

- The scale bar may be oriented horizontally or vertically; the longer side corresponds to the length input.
- The scale bar will be the only pure black pixels on the image.
- The output dimensions should be [height width].

Function Name: `pokemonGO`

Inputs:

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

Outputs:

(none)

File Outputs:

1. Image with pokémon replaced with pokéballs (`'<filename>_captured.png'`)
2. Image of your personal pokédex (`'<filename>_pokedex.png'`)

Function Description:

Just when Pokémon GO stopped taking over the internet, it took over MATLAB. Write a function called `pokemonGO` that takes in two images. The first will be a landscape with pokémon and the other will be the same landscape without. The image inputs will always be square images with the same sizes. Pokémon can be placed anywhere on a 5x5 square grid on the image. Your function should be able to go through the image and replace the pokémon with the appropriate pokéball to show that they have been caught. The type of pokéball used will depend on the rarity of the pokémon. Pokémon can have a rarity of `'common'`, `'uncommon'`, `'rare'`, or `'legendary'`. Common pokémon should be replaced with a regular pokéball, uncommon with a greatball, rare with an ultraball, and legendary with a masterball. Each of these pokéball images are provided for you as square images (`pokeball.png`, `greatball.png`, `ultraball.png`, `masterball.png`). You are also provided the national pokédex (structure array) and the `pokedex()` function which will identify your pokémon. The inputs for `pokedex()` 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:

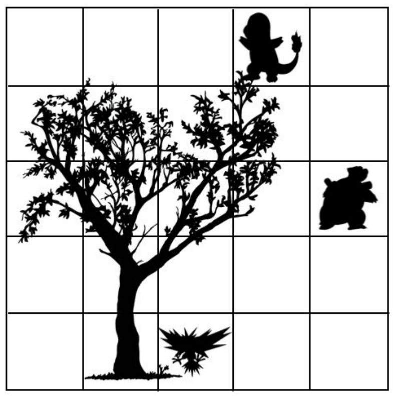
`'No pokemon detected.'` OR `'<pokemon name>, the <description> pokemon.'`

Use these two resources to figure out which pokémon exist in the image (if any do), replace them with the appropriate pokéballs, and output the updated image with `'_captured.png'` appended at the the end of the filename. You should also use the `nationalPokedex` structure to output an image of your personal pokédex, with `'_pokedex.png'` appended at the end of the filename, showing the pokémon you've caught in a single row. The personal pokédex image should always be of size `100x(n*100)`, where `n` is the number of pokémon you caught.

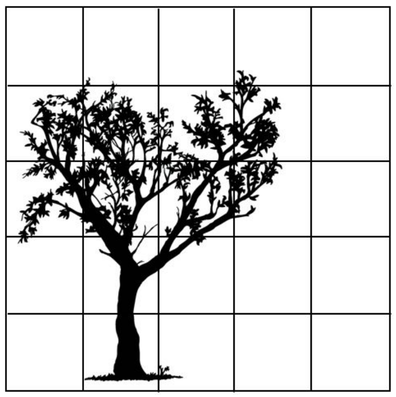
`nationalPokedex` is a `1x40` structure array (each structure contains a unique pokémon) with 4 fields: `pokemon` (name of the pokémon as a string), `image` (an `NxNx3 uint8` array of the pokémon image), `rarity` (a string representing the rarity) and `description` (used for the output of pokédex).

Example:
You should think about the input images as the following (except in color and without the grid).

`Tree_pokemon.png (MxM)`



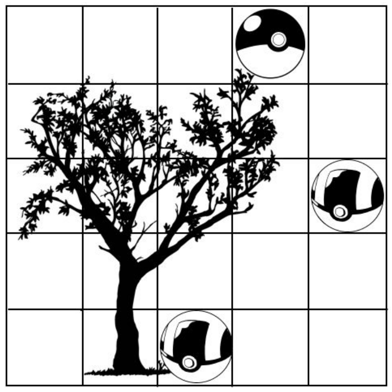
`Tree.png (MxM)`



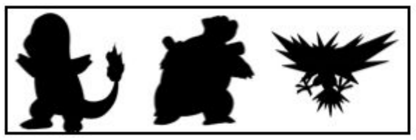
```
>> pokemonGO('tree_pokemon.png', 'tree.png', nationalPokedex)
```

Should produce the following images:

`tree_captured.png (MxM)`



`tree_pokedex.png (100x300)`



When searching through your image, you should always start in the top left corner and search from left to right.

Notes:

- The two input images will always be square images of the same size, with each of their dimensions divisible by 5.
- Each of the pokéball images are square images with perfectly green backgrounds.
- The pokémon uint8 3D arrays stored in `nationalPokedex.image` are always square images (`PxPx3`), but do not all have the same dimensions.

Hints:

- The `imresize` function will be useful.
- Take a look the `nationalPokedex` structure array to see how everything is stored.