Data, Metadata and APIs

Part 2: Photo Filter Review (Modifying Data)

Now that we know how bitmap uses bytes to encode image data, we can have some fun by manipulating those bytes. The results look like the types of filters you might apply to your photographs on Instagram, Snapchat, etc. We did this same thing first semester with HTML/JavaScript.

Taking an Image File as Input

We first start by writing the bytes of *flowers.bmp* to a byte array named *original_bytes*. In this case, *flowers.bmp* is the **input** for our algorithm:

In [1]:

```
# Create a byte array from the binary data in flowers.bmp
with open("flowers.bmp", 'rb') as original_image:
    original_data = original_image.read()
    original_bytes = bytearray(original_data)
```

It looks like nothing happened, but the image data is currently stored in the variable *original_bytes*, just waiting to be modified. Just in case you forgot from yesterday, here's what this image looks like:

In [2]:

```
# Display flowers.bmp

from PIL import Image
img = Image.open("flowers.bmp")
img.save("output/flowers.png",'png')

from IPython.display import Image
Image(filename="output/flowers.png")
```

Out[2]:

Abstraction to Manage the Complexity of your Algorithm

One example of abstraction is giving a name to an algorithm (AKA defining a function). For example, think about when your parents say, "Clean your room!" This is actually an abstraction for a multistep procedure: "Make your bed, fold & put away your laundry, throw away any garbage that is on your desk/floor, and return any dirty dishes to the kitchen." In pseudocode:

```
def clean_room():

    make_bed()

    fold_laundry()

    throw_away_garbage()

    return_dishes()
```

The benefit of *defining clean_room()* is that if you need to refer to these steps in future algorithms, you just have to write *clean_room()* instead of having to write a lot of additional code: *make_bed()*, *fold_laundry()*, *throw_away_garbage()*, and *return_dishes()*.

That is, **abstraction** allows us to *manage the complexity of a program*.

Here are three functions that we will use to help manage the complexity of our program.

Abstraction 1: A function that creates a list of pixels from the bytes of the .bmp file

We will use the function *bitmap_to_pixels(byte_array)* any time we want to convert a new *.bmp* file to a list of RGB pixels:

In [3]:

```
# Summary: Reads a bitmap byte array and return the file header and a list of pi
# Parameters: A byte array from a bitmap
# Return: A tuple in the form of (header, RGB triples list)

def bitmap_to_pixels(byte_array):
    pixels_list = []
    length_of_image_bytes = len(byte_array) - 54 # Read after the 54th byte
    number_of_pixels = length_of_image_bytes//3 # There are 3 bytes per pixel
    header = byte_array[:54] # This is where the metadata is stored
    for i in range(number_of_pixels):
        b = byte_array[54 + 3*i] # Read the blue byte the starts right after the
        g = byte_array[54 + 3*i + 1] # Read the green byte the starts right afte
        r = byte_array[54 + 3*i + 2] # Read the red byte the starts right after
        pixel = [r,g,b] # Store the three channels as an RGB list named 'pixel'
        pixels_list.append(pixel) # Append 'pixel' to pixel list
    return header, pixels_list # Return the file header (metadata) and list of p
```

Abstraction 2: A function that takes a list of pixels and converts it to the bytes of a .bmp file

This function, *pixels_to_bitmap(header, pixel_list)*, goes in the opposite direction. It takes a file header and a list of pixels to create a bitmap byte array:

In [4]:

```
# Summary: Reads a file header and list of pixels (RGB triples) and returns a bi
# Parameters: The 1st parameter is the bitmap file header and the 2nd parameter
# Return: A bitmap byte array

def pixels_to_bitmap(header,pixel_list):
    byte_array = header
    number_of_pixels = len(pixel_list)
    for i in range(number_of_pixels):
        r, g, b = pixel_list[i][0], pixel_list[i][1], pixel_list[i][2]
        byte_array.append(b)
        byte_array.append(g)
        byte_array.append(r)
    return byte_array
```

Abstraction 3: A function that writes out the bytes to a new .bmp file

This final function, *file_writer(byte_array, new_file_name)* allows you to create a bitmap (.bmp) file from a bitmap byte array. It will save it to whatever directory you have this notebook saved in. This function allows us to create a *.bmp* file as output.

In [5]:

```
# Summary: Saves a bitmap byte array as a .bmp file with the specified filename
# Parameters: The 1st parameter is the bitmap byte array and the 2nd parameter i
# Return: Technically none, but this function does write out your bitmap file

def file_writer(byte_array, new_file_name):
    full_name = new_file_name
    new_file = open(new_file_name, 'wb')
    new_file.write(byte_array)
    new_file.close()
```

Image Filter #1: Turning 'Very Dark Pixels' to White

A pixel is "very dark" if is is a triple of the form [x, y, z] where x < 30, y < 30, and z < 30. Let's turn these pixels white:

In [6]:

```
header, pixel_list = bitmap_to_pixels(original_bytes) # Gather header and pixel

new_pixel_list = []

for pixel in pixel_list: # Loop through every pixel
    new_pixel = pixel # Copy the original pixel
    if pixel[0] < 30 and pixel[1] < 30 and pixel[2] < 30: # Check if R<30, G<30,
        new_pixel = [255, 255, 255] # Overwrite the original pixel saved
    new_pixel_list.append(new_pixel) # Add to list

new_file = pixels_to_bitmap(header, new_pixel_list) # Put file back together!

file_writer(new_file,"output/flowers_white_background.bmp") # Save the image in
```

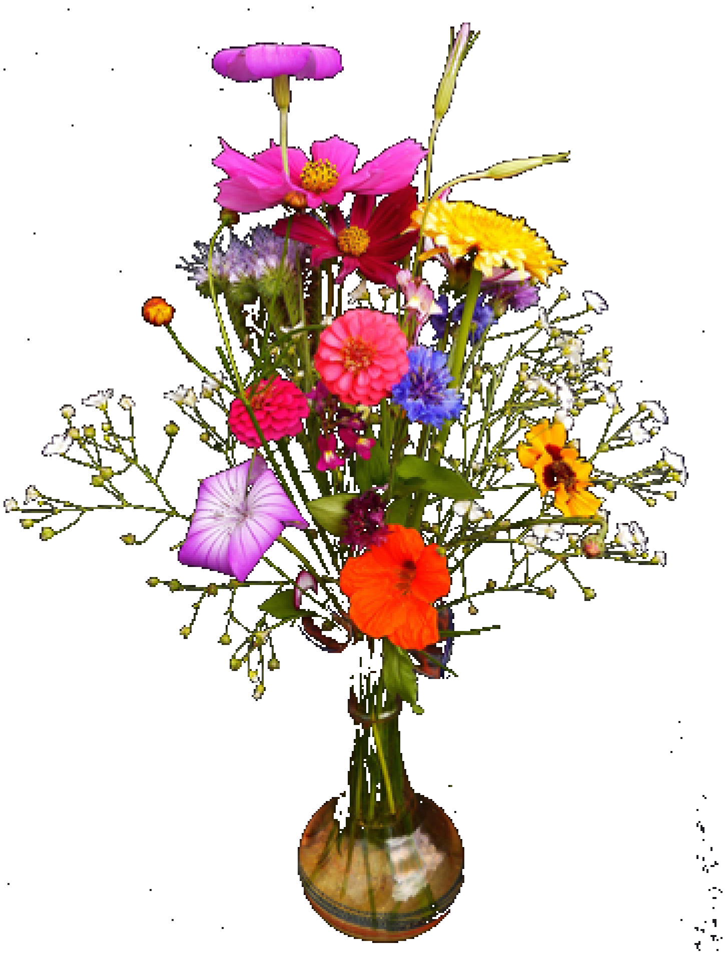
Let's take a look:

In [7]:

```
from PIL import Image
img = Image.open("output/flowers_white_background.bmp")
img.save("output/flowers_white_background.png",'png')

from IPython.display import Image
Image(filename="output/flowers_white_background.png")
```

Out[7]:



Not a bad result for just a few lines of code!

Image Filter #1 as an Abstraction

We can encapsulate the entire image filter procedure into a function:

In [8]:
```
def very_dark_to_white(file_name):

    with open(file_name, 'rb') as original_image:
        original_data = original_image.read()
        original_bytes = bytearray(original_data)

    header, pixel_list = bitmap_to_pixels(original_bytes)

    new_pixel_list = []

    for pixel in pixel_list:

        new_pixel = pixel

        #####################################################
        # image filter code goes in this section ############
        #####################################################
        if pixel[0] < 30 and pixel[1] < 30 and pixel[2] < 30:
            new_pixel = [255, 255, 255]
        #####################################################

        new_pixel_list.append(new_pixel)

    new_file = pixels_to_bitmap(header, new_pixel_list)

    original_file_name = file_name.split('.')[0]

    new_file_name = 'output/' + original_file_name + '_white_background.bmp'

    file_writer(new_file,new_file_name)

    return new_file_name
```

This function is an abstraction because very_dark_to_white(file_name) takes a *.bmp* file as input, writes it into an array of bytes, converts the bytes to a list of RGB pixels, applies the filter, converts the pixels back to bytes, then writes the file out as a new bitmap file. That's a lot of steps!

A function like is **an abstraction that manages the complexity of a program**, because applying this procedure to another file is now a piece of cake. Let's try. Start with a new image with a lot of "very dark" pixels:

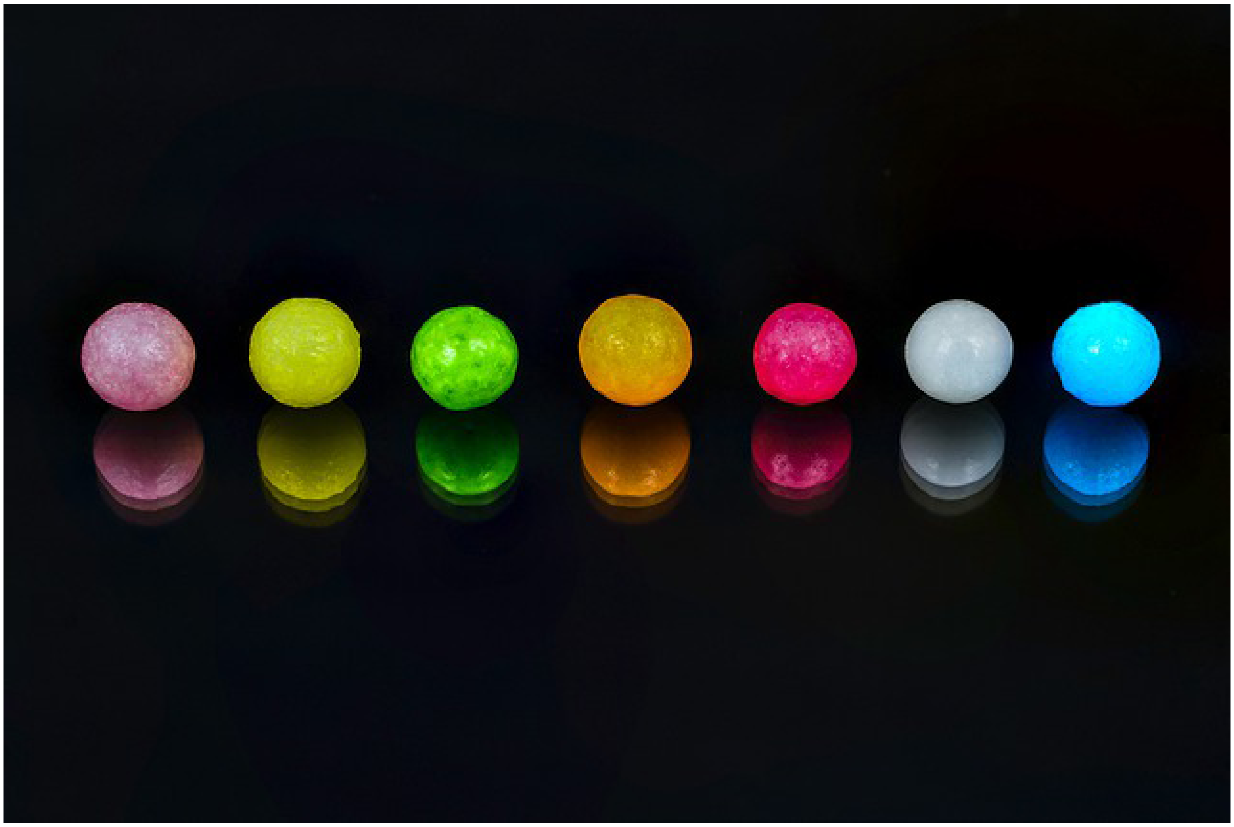
In [9]:

```
# Image Source: https://pixabay.com/en/nonpareils-balls-beads-sweetness-3128506/

from PIL import Image
img = Image.open("candy.bmp")
img.save("output/candy.png",'png')

from IPython.display import Image
Image(filename="output/candy.png")
```

Out[9]:



Now apply the filter:

In [10]:

```
very_dark_to_white('candy.bmp')
```

Out[10]:

```
'output/candy_white_background.bmp'
```

A complicated, multi-step procedure only required us to write a single line of code.

Now view the result:

In [11]:
```
from PIL import Image
img = Image.open("output/candy_white_background.bmp")
img.save("output/candy_white_background.png",'png')

from IPython.display import Image
Image(filename="output/candy_white_background.png")
```

Out[11]:

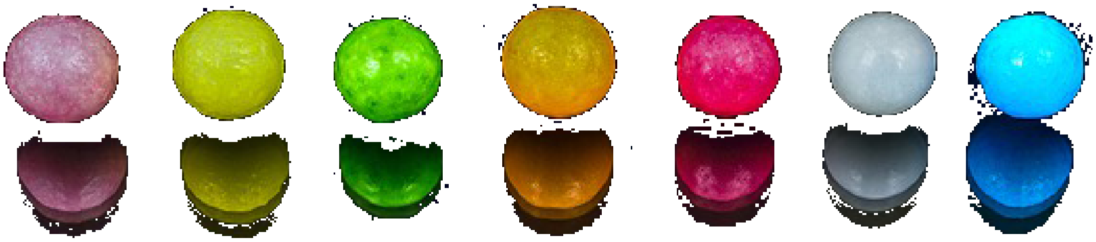


Image Filter #2: Monochrome Red

Now that you have the framework for an image filter, you don't need to write much code to create a "Monochrome Red" filter.

The filter needs to zero out all of the color information except for red:

In [12]:

```python
def red_monochrome(file_name):

    with open(file_name, 'rb') as original_image:
        original_data = original_image.read()
        original_bytes = bytearray(original_data)

    header, pixel_list = bitmap_to_pixels(original_bytes)

    new_pixel_list = []

    for pixel in pixel_list:

        new_pixel = pixel

        #####################################################
        # image filter code goes in this section ############
        #####################################################
        new_pixel[1] = 0 # set the green channel to 0
        new_pixel[2] = 0 # set the blue channel to 0
        #####################################################

        new_pixel_list.append(new_pixel)

    new_file = pixels_to_bitmap(header, new_pixel_list)

    original_file_name = file_name.split('.')[0]

    new_file_name = 'output/' + original_file_name + '_red_monochrome.bmp'

    file_writer(new_file,new_file_name)

    return new_file_name
```

Apply the filter to the flower picture:

In [13]:

```python
red_monochrome("flowers.bmp")
```

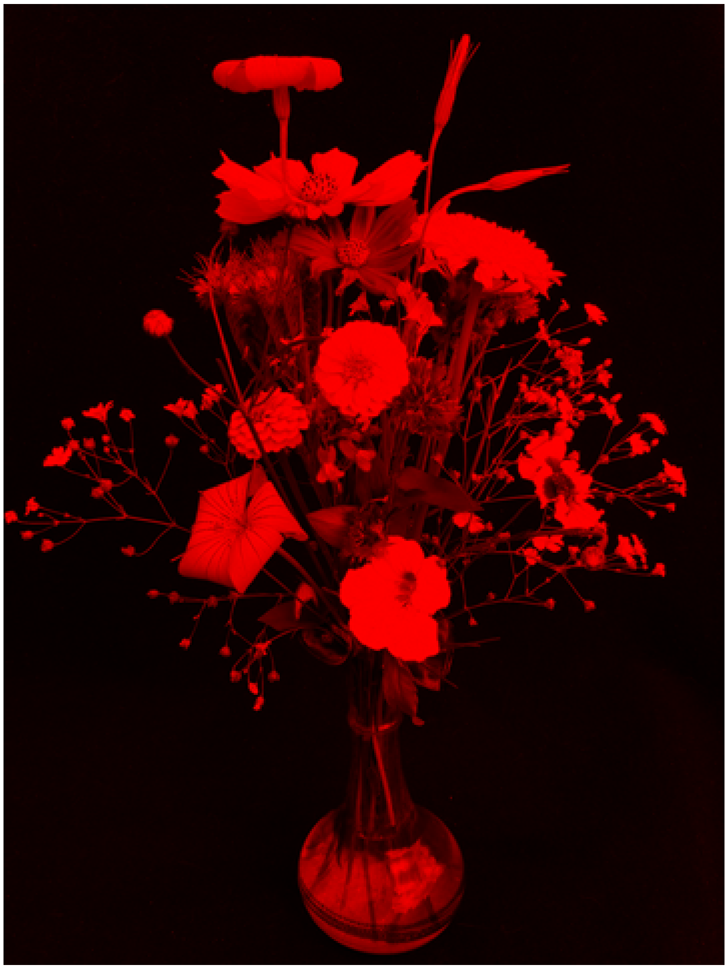
Out[13]:

```
'output/flowers_red_monochrome.bmp'
```

In [14]:
```
from PIL import Image
img = Image.open("output/flowers_red_monochrome.bmp")
img.save("output/flowers_red_monochrome.png",'png')

from IPython.display import Image
Image(filename="output/flowers_red_monochrome.png")
```

Out[14]:



It looks like a bouquet of flowers in an old photography darkroom.

Task #1: Grayscale (Black and White) Filter

Create a grayscale (black and white) filter. Display your results and explain how your code works.

Hint: Pure gray requires all three color channels to have the same value. Also, your filter must be in the form of a function.

In [15]:

```
# Your code here
def grayscale(file_name):

    with open(file_name, 'rb') as original_image:
        original_data = original_image.read()
        original_bytes = bytearray(original_data)

    header, pixel_list = bitmap_to_pixels(original_bytes)

    new_pixel_list = []

    for pixel in pixel_list:

        new_pixel = pixel

        #####################################################
        # image filter code goes in this section ############
        #####################################################
        new_pixel[1] = new_pixel[2] # set the green channel to 0
        new_pixel[0] = new_pixel[2]# set the blue channel to 0
        #####################################################

        new_pixel_list.append(new_pixel)

    new_file = pixels_to_bitmap(header, new_pixel_list)

    original_file_name = file_name.split('.')[0]

    new_file_name = 'output/' + original_file_name + '_red_monochrome.bmp'

    file_writer(new_file,new_file_name)

    return new_file_name
```

In [16]:

```
grayscale("flowers.bmp")
```

Out[16]:

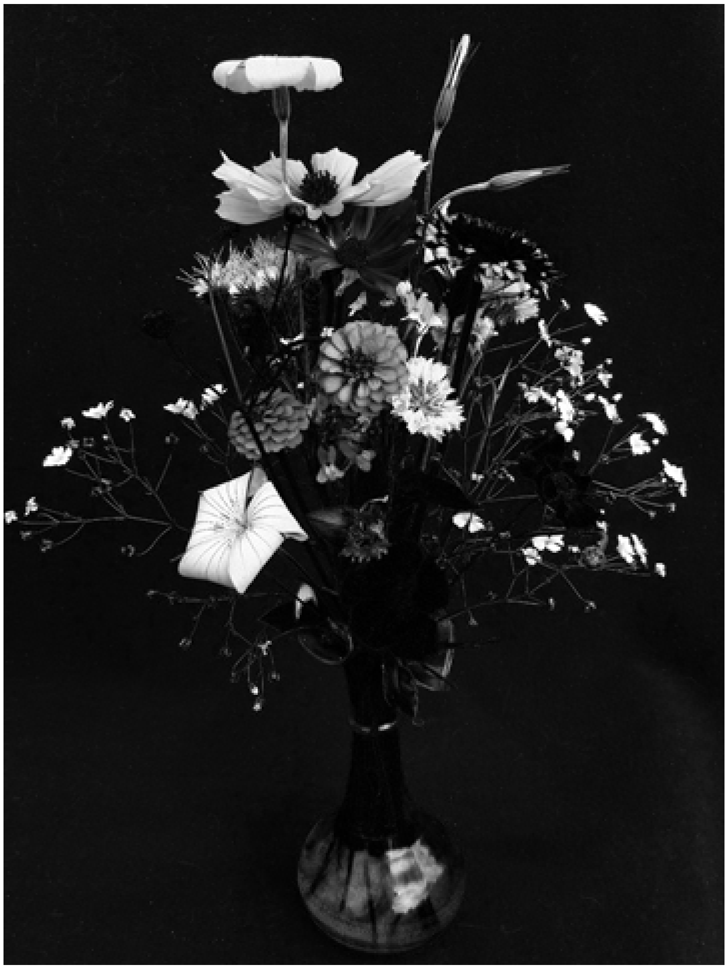
```
'output/flowers_red_monochrome.bmp'
```

In [17]:

```
from PIL import Image
img = Image.open("output/flowers_red_monochrome.bmp")
img.save("output/flowers_grayscale.png",'png')

from IPython.display import Image
Image(filename="output/flowers_grayscale.png")
```

Out[17]:



Task #2: Mystery Filter

Below is some code to insert into a function, *mystery_filter(file_name)*. Based on what you see in the code, what do you think it will do to the image? Explain, and also test your hypothesis on *red1.bmp* and *red2.bmp*.

Your Answer: It will make it gray, but the gray wll be the average of all the colors in the RGB.

Here is the code you need to create your mystery filter:

In [18]:

```python
def grayscale_two(file_name):

    with open(file_name, 'rb') as original_image:
        original_data = original_image.read()
        original_bytes = bytearray(original_data)

    header, pixel_list = bitmap_to_pixels(original_bytes)

    new_pixel_list = []

    for pixel in pixel_list:

        new_pixel = pixel

        average = (pixel[0]+pixel[1]+pixel[2])//3
        new_pixel[0] = average
        new_pixel[1] = average
        new_pixel[2] = average

        new_pixel_list.append(new_pixel)

    new_file = pixels_to_bitmap(header, new_pixel_list)

    original_file_name = file_name.split('.')[0]

    new_file_name = 'output/' + original_file_name + '_grayscale.bmp'

    file_writer(new_file,new_file_name)

    return new_file_name
```

In [19]:

```python
grayscale_two("red1.bmp")
```

Out[19]:

```
'output/red1_grayscale.bmp'
```

Next, here is *red1.bmp*, one of the pictures to which you must apply your mystery filter:

In [20]:
```
# Here is red1.bmp:

from PIL import Image
img = Image.open("red1.bmp")
img.save("output/red1.png",'png')

from IPython.display import Image
Image(filename="output/red1.png")
```

Out[20]:

In [21]:
```
grayscale_two("red2.bmp")
```

Out[21]:
```
'output/red2_grayscale.bmp'
```

Finally, here is *red2.bmp*, the other picture to which you must apply your mystery filter:

In [22]:

```
# Here is red2.bmp:

from PIL import Image
img = Image.open("red2.bmp")
img.save("output/red2.png",'png')

from IPython.display import Image
Image(filename="output/red2.png")
```

Out[22]:

Task #3: The Secret Message

Hidden in this bitmap's red noise is a secret message. Find it!

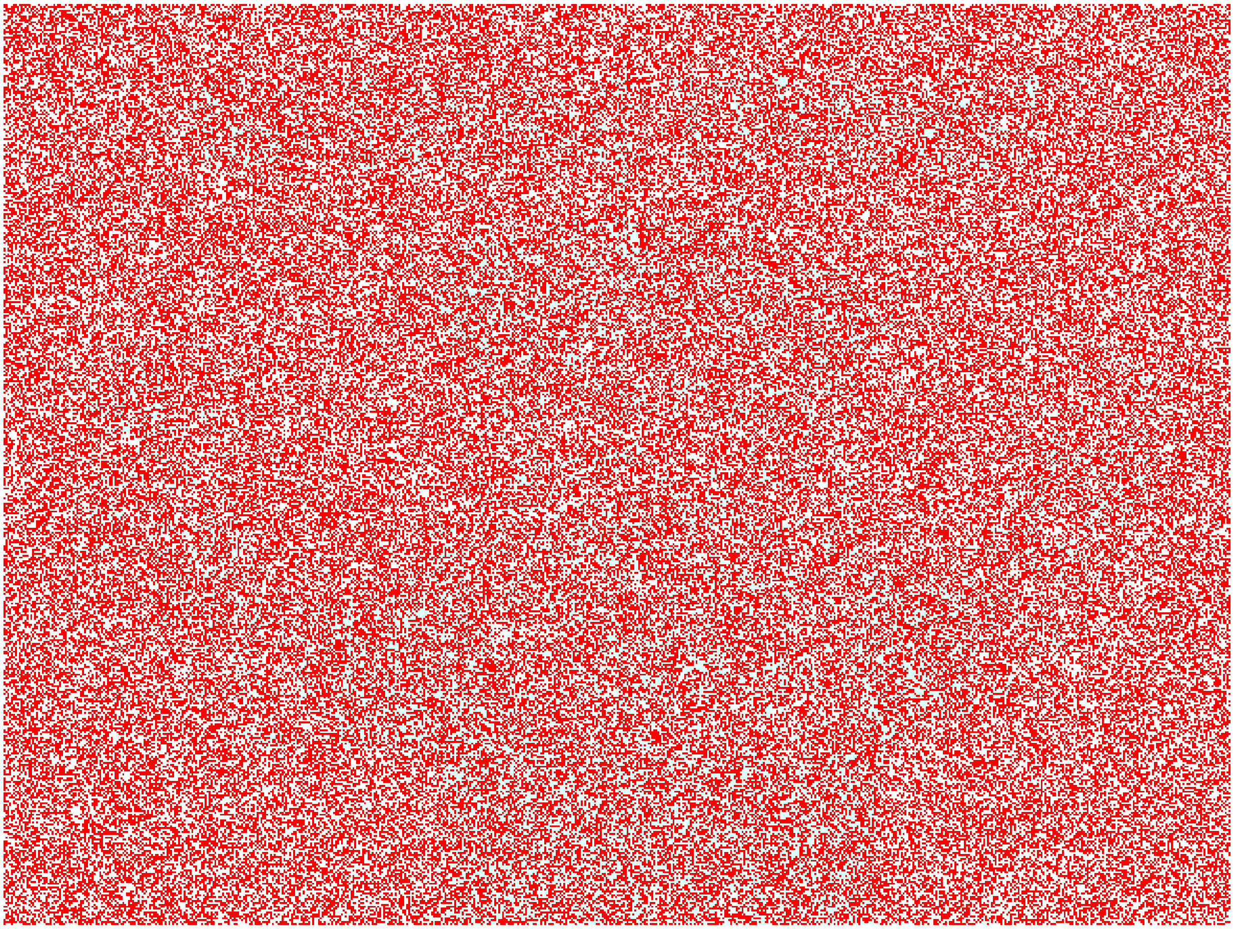
Hint: Many of you solved this puzzle during first semester. You can look back at old code if you'd like, but this time solve it using Python.

In [23]:

```
from PIL import Image
img = Image.open("clue.bmp")
img.save("output/clue.png",'png')

from IPython.display import Image
Image(filename="output/clue.png")
```

Out[23]:



In [24]:

```
# Your code here
def image_filter(file_name):

    with open(file_name, 'rb') as original_image:
        original_data = original_image.read()
        original_bytes = bytearray(original_data)

    header, pixel_list = bitmap_to_pixels(original_bytes)

    new_pixel_list = []

    for pixel in pixel_list:

        new_pixel = pixel

        if pixel[0] == 255:
            pixel[0] = 0
            pixel[1] = 0
            pixel[2] = 0
        elif pixel[1] == 255 and pixel[2] == 255:
            pixel[0] = 255
            pixel[1] = 255
            pixel[2] = 255

        new_pixel_list.append(new_pixel)

    new_file = pixels_to_bitmap(header, new_pixel_list)

    original_file_name = file_name.split('.')[0]

    new_file_name = 'output/' + original_file_name + '_filtered.bmp'

    file_writer(new_file,new_file_name)

    return new_file_name
```

In [25]:

```
image_filter("clue.bmp")
```

Out[25]:

```
'output/clue_filtered.bmp'
```

What is the secret message, and how did you find it?

Your Answer: Go FREMD VIKINGS

I found it by copy and pasting the garyscale code, and then looking at my old code in javascript to see what the point of the filter is. I saw that if there ios a red pixel, make is white, and if there is a cyan pixel, make it black.

Task #4: The French-to-Irish Filter

Here's a French flag. Write a filter that transforms it into an Irish flag.

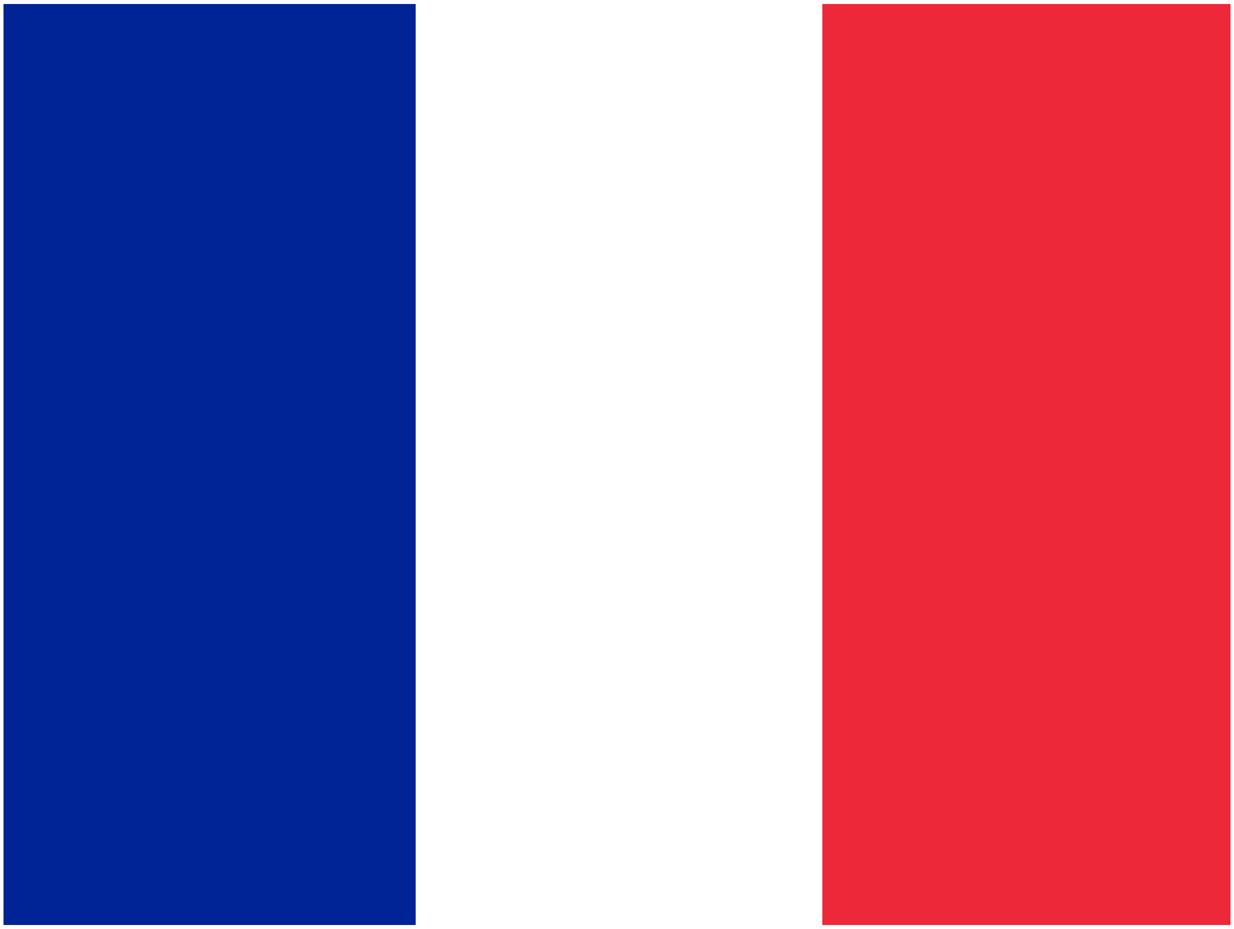
Hint: You may want to write some code to find out the RGB values of the three colors in this flag. Your filter must be in the form of a function.

In [26]:

```python
from PIL import Image
img = Image.open("france.bmp")
img.save("output/france.png",'png')

from IPython.display import Image
Image(filename="output/france.png")
```

Out[26]:



In [27]:

```
# Your code here
def flag_filter(file_name):

    with open(file_name, 'rb') as original_image:
        original_data = original_image.read()
        original_bytes = bytearray(original_data)

    header, pixel_list = bitmap_to_pixels(original_bytes)

    new_pixel_list = []

    for pixel in pixel_list:

        new_pixel = pixel

        if pixel[2] >= 100 and pixel[1] != 255:
            pixel[0] = 0
            pixel[1] = 255
            pixel[2] = 0
        if pixel[0] >= 200 and pixel[1] != 255 :
            pixel[0] = 255
            pixel[1] = 165
            pixel[2] = 0

        new_pixel_list.append(new_pixel)

    new_file = pixels_to_bitmap(header, new_pixel_list)

    original_file_name = file_name.split('.')[0]

    new_file_name = 'output/' + original_file_name + '_to_irish.bmp'

    file_writer(new_file,new_file_name)

    return new_file_name
```

In [28]:

```
flag_filter("france.bmp")
```

Out[28]:

```
'output/france_to_irish.bmp'
```

Task #5: Exploring Types of Output

Run the following code cell below. Then explain what the code does and why it is useful.

Hint: If you run the cell but can't tell what it did, look in the folder where this notebook is located. You might find something interesting.

Your Answer:It creates a website that coverts the file flower.bmp from red_monochrome to its original state.

In [29]:

```
output_string = """
<html>
<head>
<style>
    body {
        background-color: #BBBBBB;
        text-align: center;
    }
</style>
<script>
    function changePic(){
        document.getElementById('idPic').src = 'flowers.bmp';
        document.getElementById('idHeader').innerHTML = 'Original Image:';
        document.body.style.backgroundColor = '#FFAA00';
    }
</script>

</head>

<body>
<h1 id='idHeader'>Filtered Image:</h1>
"""

output_string += "<img id='idPic' src='" + red_monochrome("flowers.bmp") + "'>"

output_string += """

<br>
<input type='button' value='View Original' onClick=changePic()>

</body>
</html>
"""

html_file= open("writeout.html","w")
html_file.write(output_string)
html_file.close()
```

In [ ]: