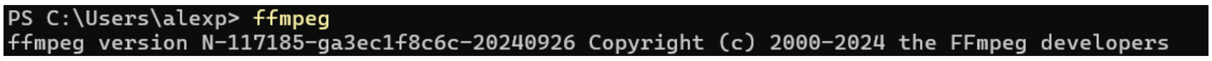
Georgina Hostench 240469
Àlex Pachón 240587
Clara López 240878

S1 - JPEG, JPEG2000, FFMPEG

1. **Install ffmpeg:**

We installed the windows version of ffmpeg and then ran it to check its version.



```
PS C:\Users\alexp> ffmpeg
ffmpeg version N-117185-ga3ec1f8c6c-20240926 Copyright (c) 2000-2024 the FFmpeg developers
```

2. **Start a script, create class and method**

We have created a class with 3 color values as attributes (value 1, 2 and 3 because we still don't know the color space). We then have created the two methods which convert RGB to YUV and otherwise, following the formulas provided in the lecture slides. These two functions return the new converted color object.

3. **Use ffmpeg to resize images into lower quality.**

We have our picture (olivia.jpg) in our folder. To resize the image using ffmpeg we need to run the command:

ffmpeg -i olivia.jpg -vf scale=320:240 output_320x240.png

The method which will call this command is called resize_image, and has as input parameters the input and output path of the images, and width and height to be resized. This is stored in a string variable called command, and then runned using subprocess.run(command) so we are able to resize the image from the terminal.

By using the terminal to resize the image, we have checked that this new method works for both windows and linux operating systems.

4. **Create a method called serpentine which should be able to read the bytes of a JPEG file in the serpentine way we saw.**

We have created a new function called *serpentine*. To be able to prove its correct functioning, we have first created a matrix with some number, and calculated manually what should be the expected output. This has been later changed for the functions which open the image and store its information in a matrix. After that, the method proceeds to initialize some variables which will allow us to iterate along all the pixels of the image, storing their values in a new array following the serpentine order. We basically identify two directions: up-right and down-left. At the very beginning, the first one is followed. For each direction, we check some conditions to change of row, column or direction.

5.1. Use FFMPEG to transform the previous image into b/w.

The command we must call in the terminal is the following:

ffmpeg -i input -vf format=gray output

We will implement this method following the logic from exercise 3.

5.2. Create a method which applies a run-lenght encoding from a series of bytes given.

We now create a new function called *rl_encoding*. This function, what it does is to compress data by storing a sequence of bytes with the same value as a unique value and its counting.

To do so, in the function we have created, we go byte by byte and if the current byte has the same value than the previous one we add one to a counter until it is different, then it is stored in the new array with the value of the counter.

6. Create a class which can convert, can decode (or both) an input using the DCT.

Now we have to create a class to convert and decode an image using DCT.

To do so, we have initialized the class with the path of the input image, and the quantization matrix. Inside this class, we have also defined functions that we will use later to compute the conversion and decoding of an image like *load_image_as_matrix* (to transform an image into a matrix), *select_q_matrix* (to choose which quantization matrix we want to use), *rl_decoding* (to undo the compression explained in 5.2). Moreover, we use the function *rl_encoding* previously explained..

The conversion is done in the *dct_compression* function. We create a 0 matrix with the same shape as the image and then we divide the image in a grid. We iterate through each block and then inside the block, through each pixel of it. We center the pixel values around zero by subtracting 128 to its value and then we apply the *dct* function from the *cv2* library. Then, we quantize it, apply the *serpentine* and *rl_encoding* functions.

Then we apply the decode in the *dct_decompression*, where we undo the procedures done in the the *dct_compression* function. We apply the *rl_decoding*, iterate through each block and pixel of the grid and dequantize it and apply the *idct* function from the *cv2* library.

Then we obtain the final output as a converted and decoded image, and as we can see, it looks exactly the same as the input one.

7. Create a class which can convert, can decode (or both) an input using the DWT.

Now we are asked to compute a new class to be able to convert and decode an input image using DWT. To do so, we use the function *dwt2 and idwt2* from the *pywt* library.

To convert the image, once we obtain the coefficients of the dwt, we can see that they are divided in four different arrays, each one for LL, LH, HL and HH as we can see in the image below:

```
                              -------------------
                              |        |        |
                              | cA(LL) | cH(LH) |
                              |        |        |
(cA, (cH, cV, cD))  <--->     -------------------
                              |        |        |
                              | cV(HL) | cD(HH) |
                              |        |        |
                              -------------------
```

Georgina Hostench 240469
Àlex Pachón 240587
Clara López 240878

And we convert these values into a single array that we will use to obtain the resulting image, in which we will see four images corresponding to the four sections above.

To decode, we use the coefficients we obtained when we converted the original image and when running the code we obtain a resulting image that looks exactly as the original image.

8. **Use any AI (YES, you can NOW, you lazy!) to create UNIT TESTS to your code, for each method and class**

In the code, below the classes we created we can see the different UNIT TESTS we used to check that the classes worked correctly.