Image Processing: Assignment #1

Obaida Khateeb , ID:201278066

Maya Atwan , ID:314813494

Problem 1

a) Considerations to take into account when deciding about the megapixels amount could include:

- **Storage Size**: higher number of pixels in the image requires more storage space.
- **Processor Power**: Handling higher resolution images need more power.
- **Image Format**: Image format determines how the resolution and colors be handled. For example, using a GIF format, which have limited number of colors (256), will not enable representing the additional details that increasing the resolution adds.

In addition, there are the considerations that are related to the desired use of the image:

- **Screen Capability:** If the image intended to be displayed in a screen, it's important that the image resolution match the screen maximum resolution capability. High resolution beyond the screen limitations will be a waste and will not be reflected into a better visualization.
- **Paper Size:** If the image going to be printed, it's important to choose a resolution that match the paper size. Larger paper size requires higher resolution in order to display the image clearly and sharply.
- **Human Perception:** As of the screen, it's important to take into consideration the human eye limitations. Adding a resolution beyond the eye differentiation ability will not yield noticeable improvement.
- **Transfer Time**: If the image intended to be uploaded or streamed, it's important to consider the time takes to transfer it. This stems from the storage size, which increase as resolution increase, leading to increased transfer time.

b) The following factors are important and should be considered when deciding about the extent to which the quantization of the image should be made:

- **Storage Size:** Stronger quantization results in smaller image size. Since the image represented as a matrix, stronger quantization leads to reduced amount of colors which requires a smaller number of bits for representing each matrix element.
- **Transfer Time:** Related to storage size, larger size images resulted from weaker quantization need more time to be transferred or streamed.
- **Screen Display ability:** The number of colors a screen can display. A quantization results in more colors than the screen ability to handle will make the screen not display part of them accurately, instead it will map them to other close colors.
- **Processor Number Representation Ability (Word Length):** The quantization should result in amount of colors that the processor can represent.

In addition, there could be considerations that are image-specific:

- **Image Nature of Details:** Strong quantization can cause image high-frequency details (like hair or thin adjacent lines) and sharp edges to disappear or to be blurred.
- **Color Transition Rate:** Strong quantization could lead two close colors to be merged into one, which could lead, especially if the two colors are adjacent, in loss of details and blurring in the boundaries.

Problem 2 - Nyquist

a) As we learnt, the sine wave $I(x,y) = \sin(2\pi x)$ has a frequency of 1. Also, the sine wave $I(x,y) = \sin(2\pi kx)$ has a frequency of k. Based on this, the frequency of the sinewave $I(x,y) = \sin(\pi kx)$ is:

$$f = \frac{k}{2}$$

Since the wavelength is 1/f. the wavelength of the given sine wave for both {2,0.25} is:

$$\boldsymbol{\lambda = \frac{1}{f} = \frac{2}{k}}$$

b) According to Nyquist theorem, to restore the image, the sampling frequency should be more than twice the largest signal frequency, mathematically:

$$\boldsymbol{f_{sample} \geq 2f_{max}}$$

Since the frequencies are uniform across the image, the largest signal frequency as seen in section (a) is:

$$\boldsymbol{f_{max} = \frac{k}{2}}$$

In addition, given that the width of each column is A, and the space between each two consecutive columns is A, which means that the distance between two adjacent samples which covers the full sin wave is 2A. Thus, the sampling frequency can be expressed as:

$$f_{sample} = \frac{1}{2A}$$

Based on the above, we conclude that:

$$\frac{1}{2A} = f_{sample} \geq 2f_{max} = 2 * \frac{k}{2} = k$$

Thus, the k that guarantees proper restoring of the image can be expressed by:

$$\rightarrow k \leq \frac{1}{2A}$$

If $A = 0.25$:

$$\boldsymbol{k \leq \frac{1}{2 * 0.25} = 2}$$

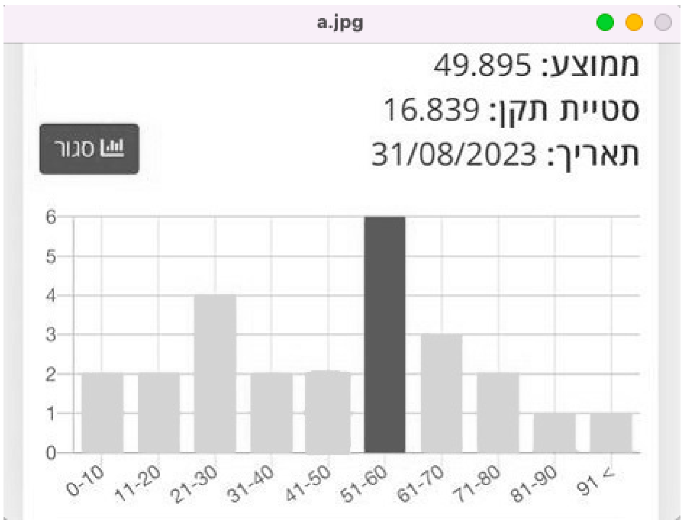
If $A = 2$:

$$\boldsymbol{k \leq \frac{1}{2 * 2} = 0.25}$$

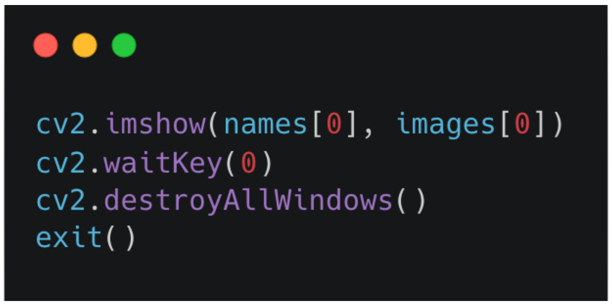
Problem 3 - Histograms, Matching, Quantization

a) After installing the required libraries cv2, matplotlib, sklearn. we called the function 'read_dir' to read images from the data folder, converts them to grayscale images and stores them in an array. the first image was successfully displayed by using the function 'imshow' from cv2 library.

below the output image, 'a.jpg', in grayscale:

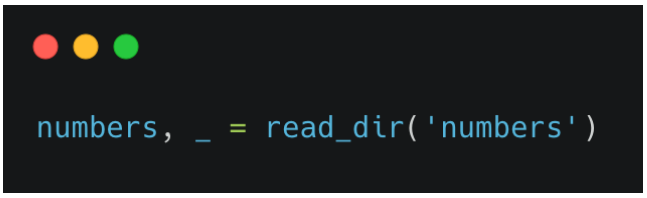


The code used for displaying the image is the following piece of code which is already provided:



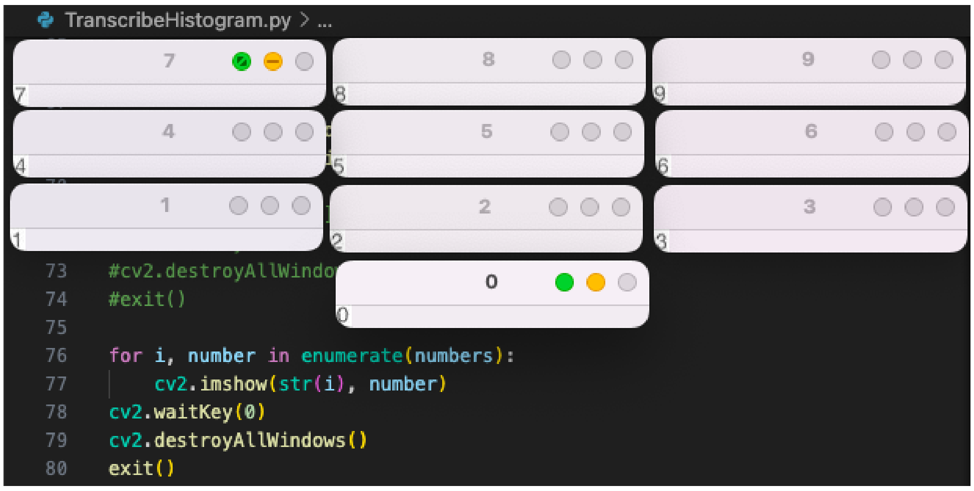
```
cv2.imshow(names[0], images[0])
cv2.waitKey(0)
cv2.destroyAllWindows()
exit()
```

b) The digit images already read by the line:



```
numbers, _ = read_dir('numbers')
```

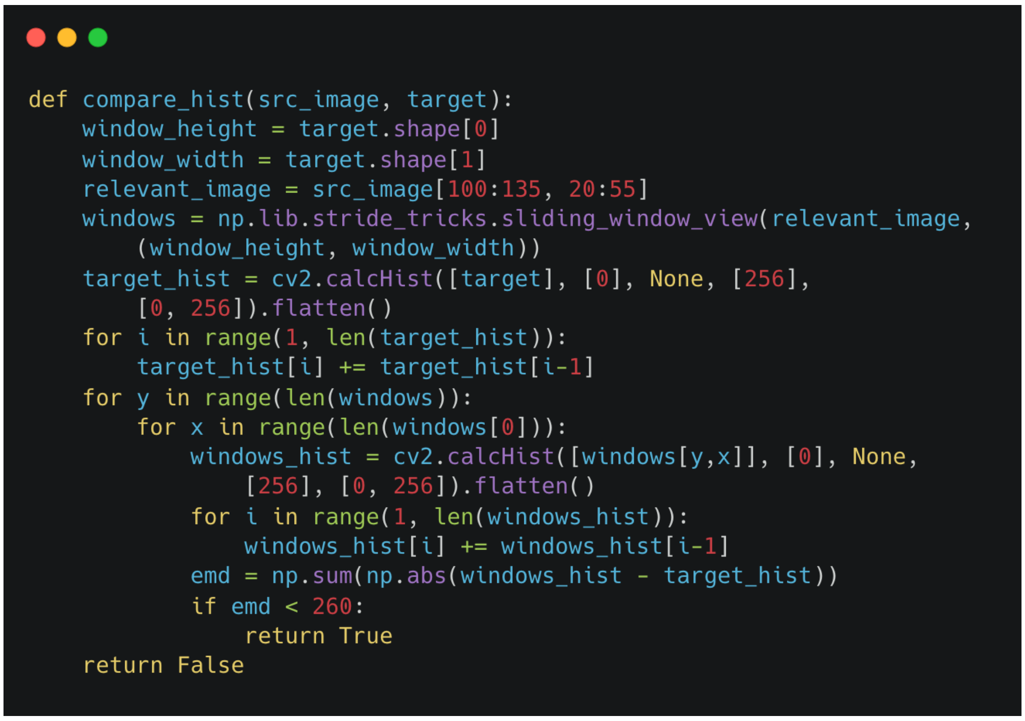
I verified this by displaying the different elements in the 'numbers' list:



TranscribeHistogram.py > ...

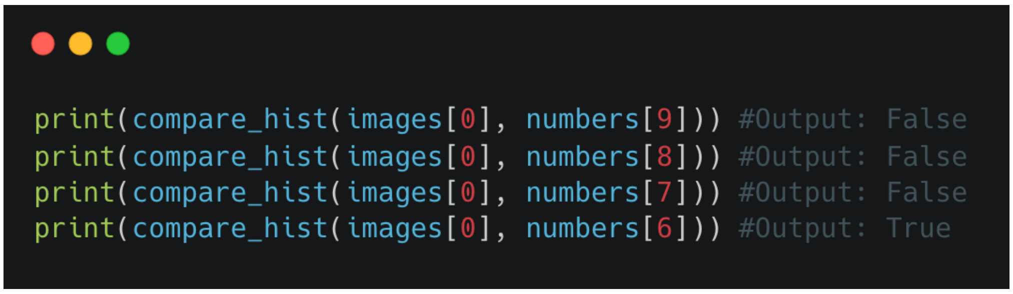
```
#cv2.destroyAllWindo
#exit()

for i, number in enumerate(numbers):
    cv2.imshow(str(i), number)
cv2.waitKey(0)
cv2.destroyAllWindows()
exit()
```

c) The ' compare_hist' method implemented. It aims to check if the target image appears in the source image. It works as the following: The method slices the region of the image where the topmost number along the vertical axis appears. It then divides the source image to small windows of the same size as the target image. Then, it computes the histograms of the target image and each of the windows. The method then computes the EMD between the target and each of the windows, if any of the EMDs is less than 260 it returns 'True', otherwise it returns 'False'.



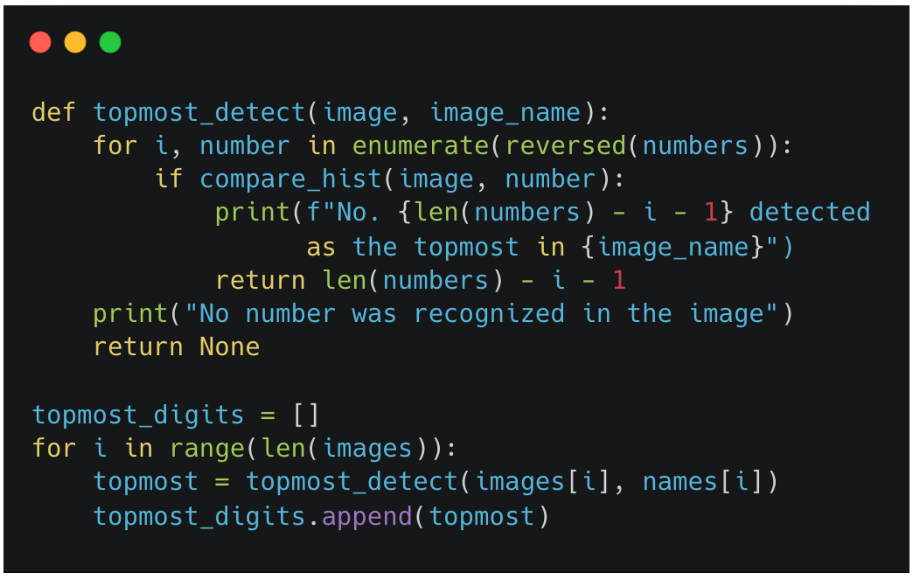
```
def compare_hist(src_image, target):
    window_height = target.shape[0]
    window_width = target.shape[1]
    relevant_image = src_image[100:135, 20:55]
    windows = np.lib.stride_tricks.sliding_window_view(relevant_image,
        (window_height, window_width))
    target_hist = cv2.calcHist([target], [0], None, [256],
        [0, 256]).flatten()
    for i in range(1, len(target_hist)):
        target_hist[i] += target_hist[i-1]
    for y in range(len(windows)):
        for x in range(len(windows[0])):
            windows_hist = cv2.calcHist([windows[y,x]], [0], None,
                [256], [0, 256]).flatten()
            for i in range(1, len(windows_hist)):
                windows_hist[i] += windows_hist[i-1]
            emd = np.sum(np.abs(windows_hist - target_hist))
            if emd < 260:
                return True
    return False
```

The method was tested using the 1st image, which topmost number is 6, along with the images of the digits 9,8,7, and 6. The results showed success.



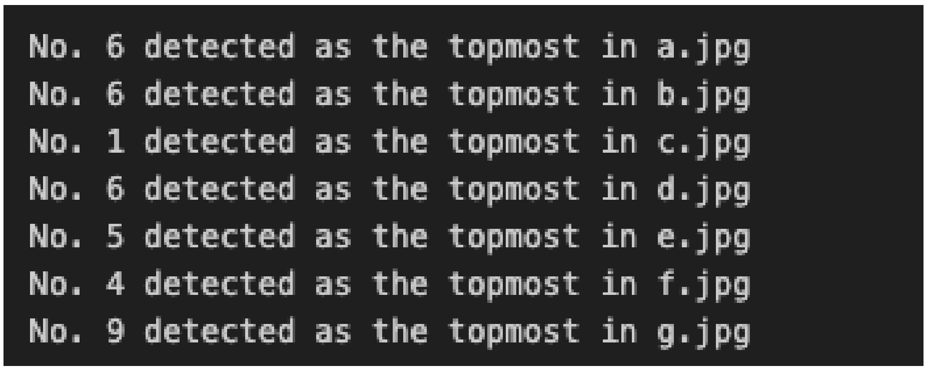
```
print(compare_hist(images[0], numbers[9])) #Output: False
print(compare_hist(images[0], numbers[8])) #Output: False
print(compare_hist(images[0], numbers[7])) #Output: False
print(compare_hist(images[0], numbers[6])) #Output: True
```

d) The task implemented by adding the below code. For each image, it iterates over the digit images in reverse order, trying (using compare_hist) to detect them in the image, once digit is found, it prints a message indicates that and returns the digit. If no digit found it prints a message accordingly and returns 'None'.



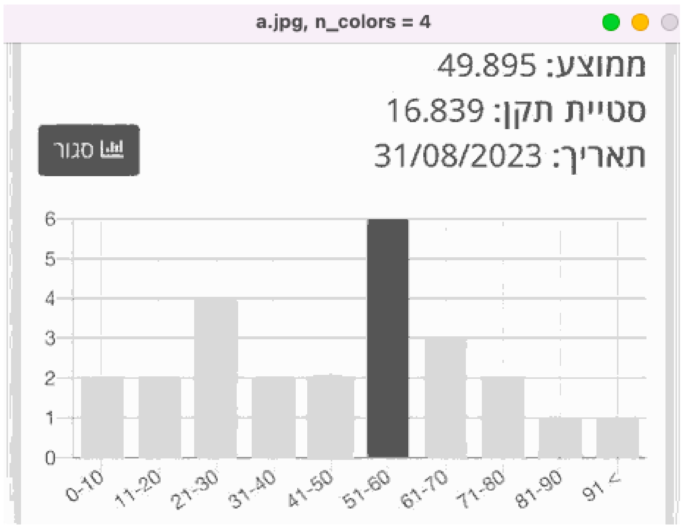
```
def topmost_detect(image, image_name):
    for i, number in enumerate(reversed(numbers)):
        if compare_hist(image, number):
            print(f"No. {len(numbers) - i - 1} detected
                  as the topmost in {image_name}")
            return len(numbers) - i - 1
    print("No number was recognized in the image")
    return None

topmost_digits = []
for i in range(len(images)):
    topmost = topmost_detect(images[i], names[i])
    topmost_digits.append(topmost)
```

The following are the results of the execution, they have been verified to be correct:



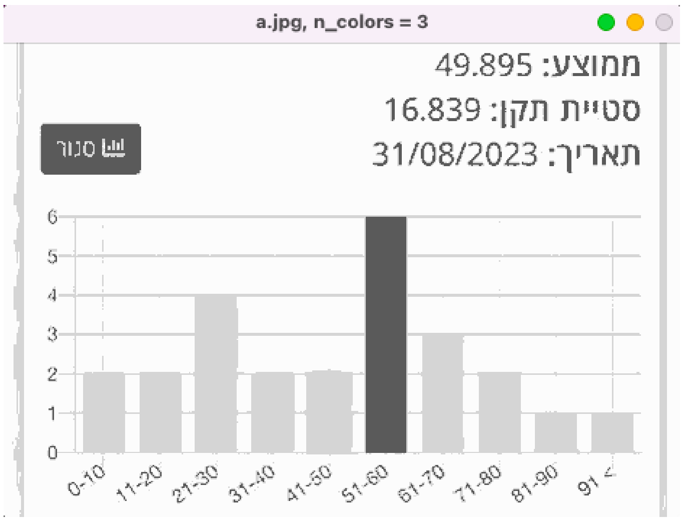
```
No. 6 detected as the topmost in a.jpg
No. 6 detected as the topmost in b.jpg
No. 1 detected as the topmost in c.jpg
No. 6 detected as the topmost in d.jpg
No. 5 detected as the topmost in e.jpg
No. 4 detected as the topmost in f.jpg
No. 9 detected as the topmost in g.jpg
```

e) The first image quantized (using the default number of gray levels, which is 4) and here is the result:

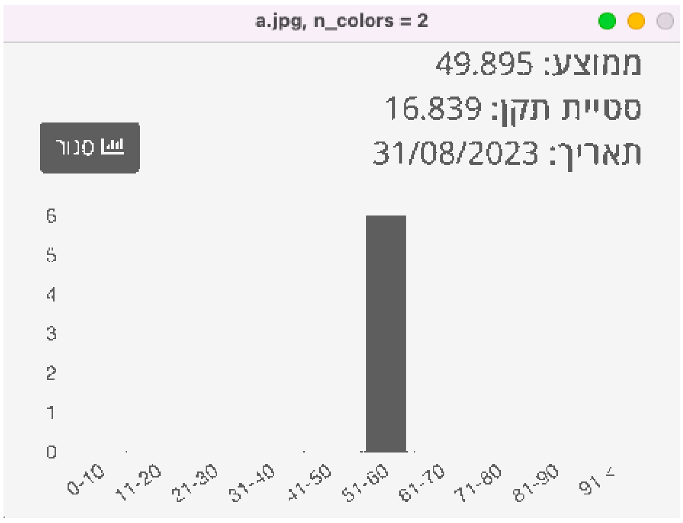


ממוצע: 49.895
סטיית תקן: 16.839
תאריך: 31/08/2023

The image then quantized using 3 and 2 gray levels, The results are below respectively:



ממוצע: 49.895
סטיית תקן: 16.839
תאריך: 31/08/2023



We can notice that using 3 gray levels preserves the appearance of the bars and separate them from the background. Using 2 levels of gray will cause the bars, except the highest one, disappear.

We decided to choose the minimum number of gray levels that preserve the relevant details (the bars) and distinct them from the background, so we chose 3 gray levels. The choice of the minimum is done for simplicity reasons and in order to avoid unnecessary complexity.

After choosing the desired number of degree levels, the quantization then done to all the images using the following line:

```
quantized_images = quantization(images, 3)
```

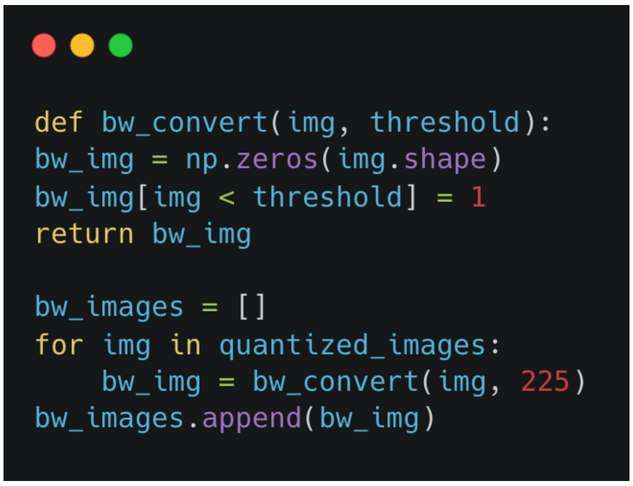
It followed by the below line, which extracts the different gray levels in the first image, this done to decide about the threshold value:

```
gray_degrees = np.unique(quantized_images[0])
```

The output was of the above line was:

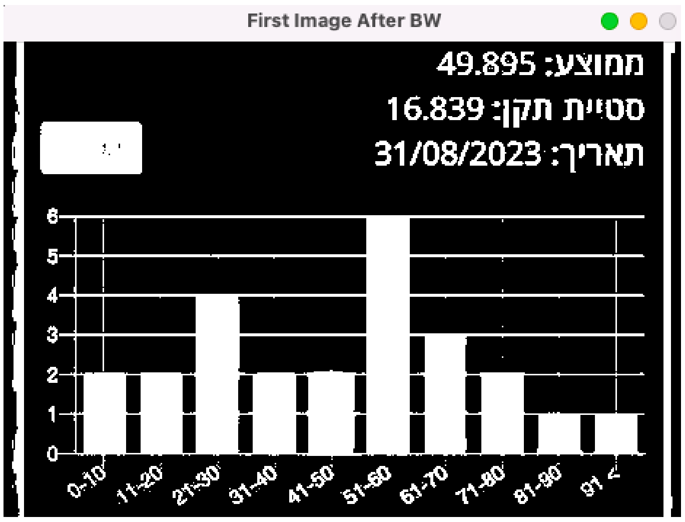
```
[ 90 213 252]
```

We assumed that 252 is the color of the background since it's the brightest in the image, while 213 and 90 are the colors of the bars. Accordingly, and due to how the 'get_bar_height' function works, we decided to use a threshold of 225, converting the colors above this threshold to 0 and those below it to 1:

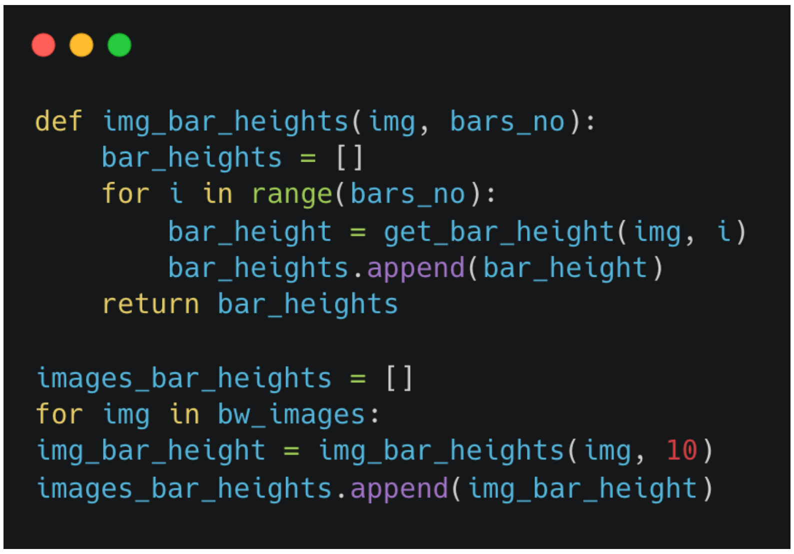


```
def bw_convert(img, threshold):
bw_img = np.zeros(img.shape)
bw_img[img < threshold] = 1
return bw_img

bw_images = []
for img in quantized_images:
    bw_img = bw_convert(img, 225)
bw_images.append(bw_img)
```

Here's the result of applying the above code on the first image:

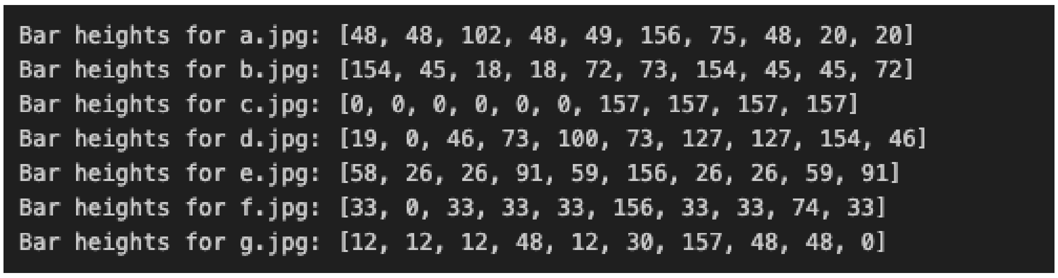


f) The task done by adding the following code. It creates a list of lists, each inner list contains the heights of the bars in a specific image.



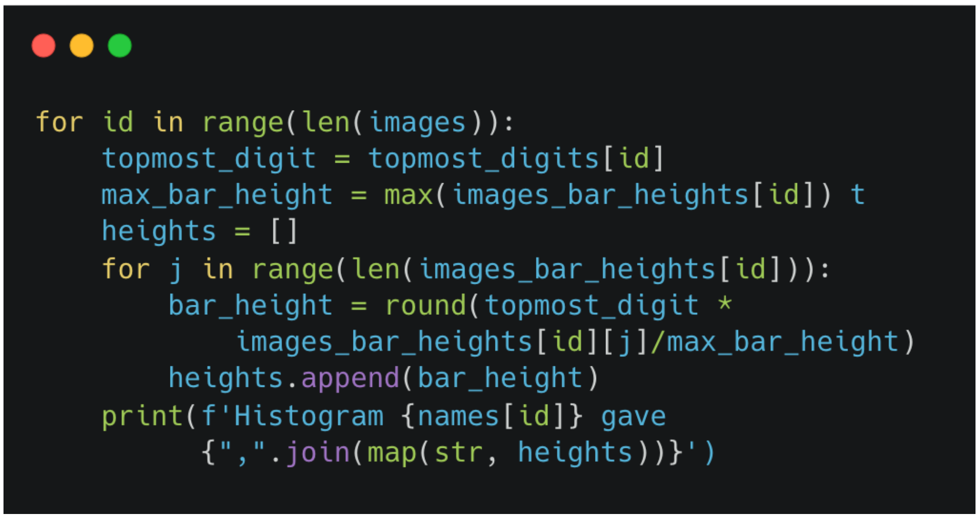
```python
def img_bar_heights(img, bars_no):
    bar_heights = []
    for i in range(bars_no):
        bar_height = get_bar_height(img, i)
        bar_heights.append(bar_height)
    return bar_heights

images_bar_heights = []
for img in bw_images:
img_bar_height = img_bar_heights(img, 10)
images_bar_heights.append(img_bar_height)
```

Below are the results received:



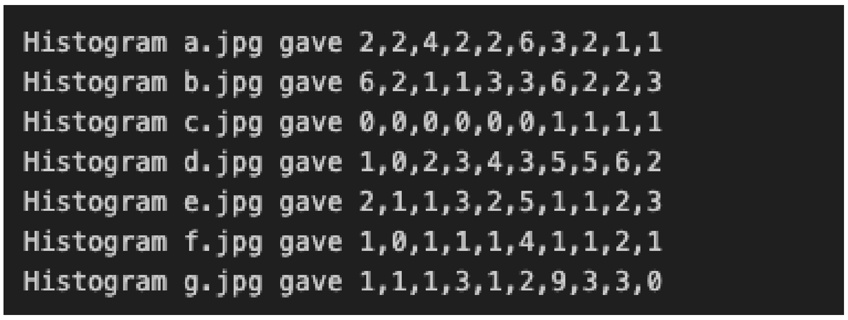
```
Bar heights for a.jpg: [48, 48, 102, 48, 49, 156, 75, 48, 20, 20]
Bar heights for b.jpg: [154, 45, 18, 18, 72, 73, 154, 45, 45, 72]
Bar heights for c.jpg: [0, 0, 0, 0, 0, 0, 157, 157, 157, 157]
Bar heights for d.jpg: [19, 0, 46, 73, 100, 73, 127, 127, 154, 46]
Bar heights for e.jpg: [58, 26, 26, 91, 59, 156, 26, 26, 59, 91]
Bar heights for f.jpg: [33, 0, 33, 33, 33, 156, 33, 33, 74, 33]
Bar heights for g.jpg: [12, 12, 12, 48, 12, 30, 157, 48, 48, 0]
```

g) The task done by adding the below code. The variable 'max-student-num' is the topmost digit that appears along the horizontal axis, The 'max-bin-height' variable is the maximum among the bar heights. The 'bin-height's were computed previously.



```python
for id in range(len(images)):
    topmost_digit = topmost_digits[id]
    max_bar_height = max(images_bar_heights[id]) t
    heights = []
    for j in range(len(images_bar_heights[id])):
        bar_height = round(topmost_digit *
            images_bar_heights[id][j]/max_bar_height)
        heights.append(bar_height)
    print(f'Histogram {names[id]} gave
          {",".join(map(str, heights))}')
```

The following is the received output:



```
Histogram a.jpg gave 2,2,4,2,2,6,3,2,1,1
Histogram b.jpg gave 6,2,1,1,3,3,6,2,2,3
Histogram c.jpg gave 0,0,0,0,0,0,1,1,1,1
Histogram d.jpg gave 1,0,2,3,4,3,5,5,6,2
Histogram e.jpg gave 2,1,1,3,2,5,1,1,2,3
Histogram f.jpg gave 1,0,1,1,1,4,1,1,2,1
Histogram g.jpg gave 1,1,1,3,1,2,9,3,3,0
```