FACULTY OF INFORMATION TECHNOLOGY

REPORT PROJECT 01
COLOR COMPRESSION

Lecturer: Nguyễn Văn Quang Huy
Phan Thị Phương Uyên
Vũ Quốc Hoàng
Lê Thanh Tùng

Student's Information:

Student's name: Nguyễn Thoại Đăng Khoa

Student's ID: 20127043

JUNE 13TH, 2022
VNUHCM – UNIVERSITY OF SCIENCE

TABLE OF CONTENTS

I. INTRODUCTION

In recent years, image compression theory is becoming increasingly important for minimizing data redundancy and saving more hardware space and transmission bandwidth. The K-means algorithm can be used to compress the image. The principle of K-means clustering algorithm for compressing images is as follow:

- Preferred number of selected clusters K is very important, K must be less than the number of image pixels N.

- Using each pixel of the image as a data point, clustering it with the K-means algorithm to obtain the centroid $\boldsymbol{\mu}$.

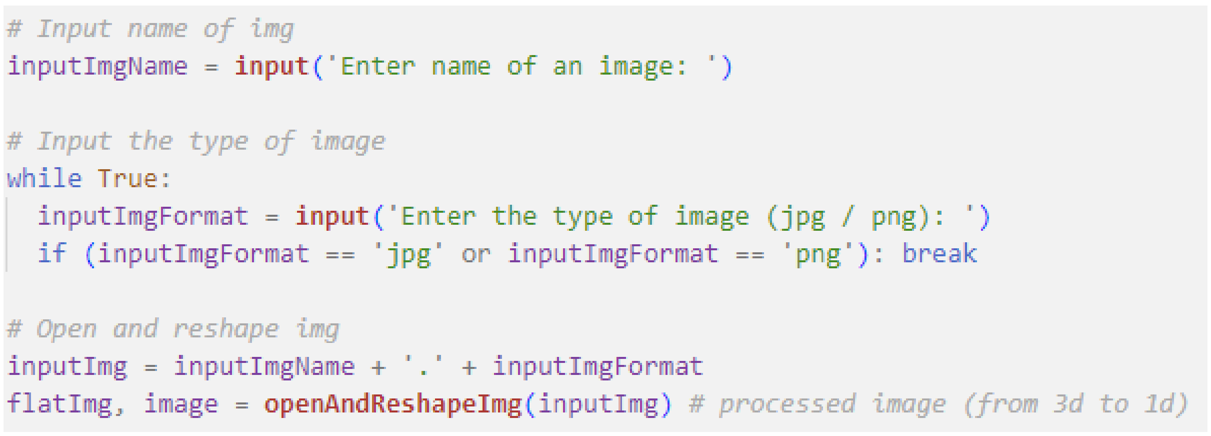
- Storing the centroid and the index of the centroid of each pixel, so it not need to keep all the original data.

To easily understand the flow of the program, I'll start with the sequence of the lines of the main function.

II. PROCESSING DETAILS

2.1. Pre-processing

Firstly, input description and image preprocessing:

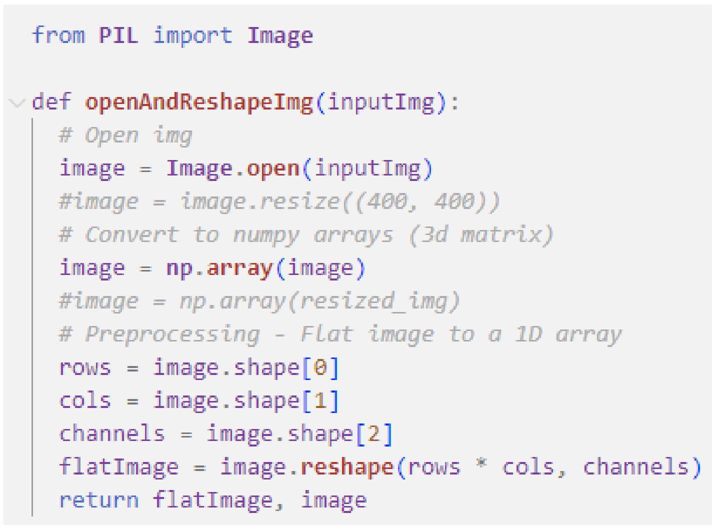


```
# Input name of img
inputImgName = input('Enter name of an image: ')

# Input the type of image
while True:
    inputImgFormat = input('Enter the type of image (jpg / png): ')
    if (inputImgFormat == 'jpg' or inputImgFormat == 'png'): break

# Open and reshape img
inputImg = inputImgName + '.' + inputImgFormat
flatImg, image = openAndReshapeImg(inputImg) # processed image (from 3d to 1d)
```

Let the user enter the image file name to be processed with the proposed format jpg/png. Then the name of the image is stored under the inputImgName variable, which is passed as an argument to the openAndReshapeImg function. In this function, I will reshape the input image.



```
from PIL import Image

def openAndReshapeImg(inputImg):
    # Open img
    image = Image.open(inputImg)
    #image = image.resize((400, 400))
    # Convert to numpy arrays (3d matrix)
    image = np.array(image)
    #image = np.array(resized_img)
    # Preprocessing - Flat image to a 1D array
    rows = image.shape[0]
    cols = image.shape[1]
    channels = image.shape[2]
    flatImage = image.reshape(rows * cols, channels)
    return flatImage, image
```

The size of the input image is (rows, cols, 3), flatten all the pixel values to a single dimension of size (rows*cols) and the dimension of each pixel is 3 representing RGB values. The size of the flatten image will be (rows*cols, channels).

image.shape[0] stands for rows, image.shape[1] stands for columns, and image.shape[2] stands for channels of an RGB image (3 channels are red, green, blue).

The return values of this function are flatImage which was flattened from the input, image (convert input image to numpy arrays)

2.2. K-means algorithm

Next, let the user enter some basic parameters used for the k-means algorithm:

```
# Enter the number of clusters
k_clusters = int(input('Enter the number of clusters (recommend <= 10): '))

# Enter the maximum number of iterations
while True:
  max_iter = int(input('Enter the max number of iterations (max 1000) (recommend <= 10): '))
  if (max_iter > 0 or max_iter <= 1000): break

# Choose type of initial centroids
while True:
  init_centroids = input('Enter the type of initial centroid (type random or in_pixels): ')
  if (init_centroids == 'random' or init_centroids == 'in_pixels'): break
```

The first is the number of clusters (the last number of colors left in the image) to be stored in the k_clusters variable, the maximum number of iterations for the algorithm is stored in the max_iter variable, and the type of initial centroids (random or in_pixels) is stored in the init_centroids variable.

After that, we will apply all of these to kmeans function:

```
# Apply K-means clustering algorithm
centroids, labels = kmeans(flatImg, k_clusters, max_iter, init_centroids)
```

```
def kmeans(img_1d, k_clusters, max_iter, init_centroids='random'):
    ### YOUR CODE HERE
    #initializing centroids based on init_centroids type
    centroids = centroids_initializing(img_1d,k_clusters,init_centroids)
    #initalize 0 for all elements in labels array
    labels = np.full(img_1d.shape[0], 0)
    while max_iter != 0:
        # Assign label to every pixel
        labels = labels_assign(img_1d, centroids)
        # Update centroid
        centroids = centroids_updating(img_1d, labels, centroids.shape)
        max_iter -= 1
    return centroids, labels
```

At this function, the implementation idea consists of 4 main steps:

- Step 1: Initialize centroids
- Step 2: Assign the label for each pixel
- Step 3: Update centroids
- Step 4: Repeat step 2 and 3 until a stopping condition is reached

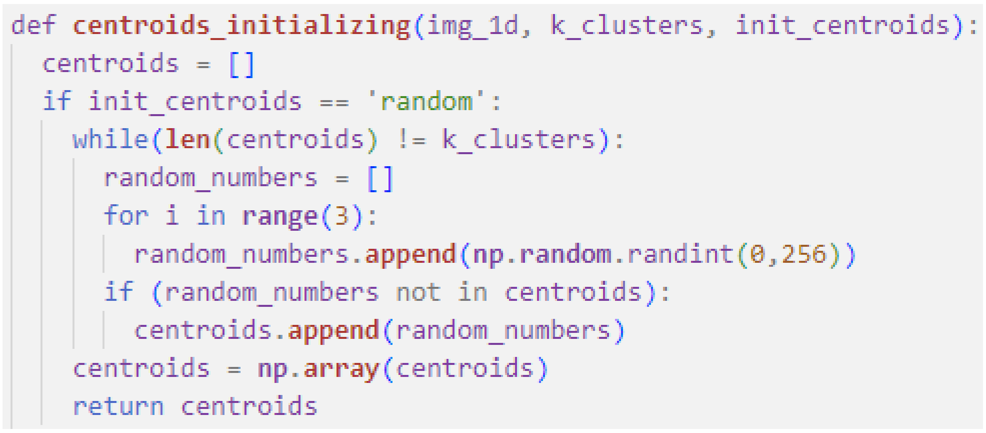
The return values are centroids, and labels. Centroids are the set of values of centroids (which color represents). Labels are a set of indexes of centroids in which every color pixel belongs to.

In first step, we initialize centroids, and labels. centroids is created from centroids_initializing function; labels is initialized with 0 for all elements in array (the number of elements is equal to the rows of flatImg we passed)

2.3. Initialize centroids

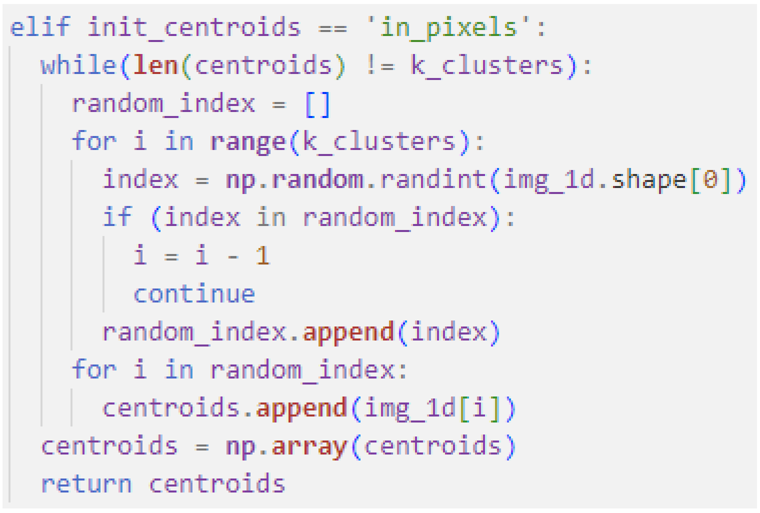
In the centroids_initializing function, we pass parameters such as img_1d (flattened image), k_cluster, max_iter and init_centroids (initialized type). There are 2 types of generation: random or in_pixels. Now we will analyze each way to create one:

First, we need to create an array of centroids so that we can return the corresponding output (centroids).



```
def centroids_initializing(img_1d, k_clusters, init_centroids):
  centroids = []
  if init_centroids == 'random':
    while(len(centroids) != k_clusters):
      random_numbers = []
      for i in range(3):
        random_numbers.append(np.random.randint(0,256))
      if (random_numbers not in centroids):
        centroids.append(random_numbers)
    centroids = np.array(centroids)
    return centroids
```

With **random mode**, because RGB image has 3 channels (red, green, blue), we need to create a random_numbers array containing a set of 3 colors with each color from 0 to 255 (using the numpy.random.randint library to generate). This set of 3 colors must be separate, if there is no duplicate value we will add to centroids array, otherwise don't add. The centroids will now contain randomly generated color vectors, the number of vectors based on the value of k_cluster passed in. Then, convert centroids to numpy array for the next processing step.



```
elif init_centroids == 'in_pixels':
  while(len(centroids) != k_clusters):
    random_index = []
    for i in range(k_clusters):
      index = np.random.randint(img_1d.shape[0])
      if (index in random_index):
        i = i - 1
        continue
      random_index.append(index)
    for i in random_index:
      centroids.append(img_1d[i])
  centroids = np.array(centroids)
  return centroids
```

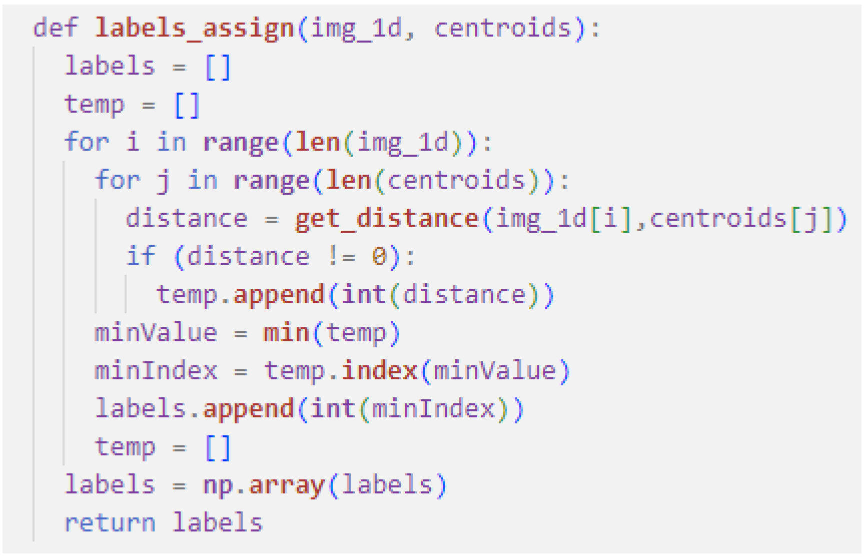
With **in_pixels mode**, we create a random_index array, with each element being the index of the color element in flatImg. We run random values between line 0 and the last line of flatImg. If there are no duplicate values, we add to the random_index array. After having enough values = k_cluster then stop. Then we add to the centroids array each element of flatImg with the corresponding index.

If not 2 modes are mentioned above, the function will return None.

After initializing centroids, and labels. We assign the label for each pixel and update centroids. Loop until the max_iter == 0, then stop.

Next, we find the way how we label for each pixel, and update centroids.

2.4. Assign label for each pixel



```
def labels_assign(img_1d, centroids):
  labels = []
  temp = []
  for i in range(len(img_1d)):
    for j in range(len(centroids)):
      distance = get_distance(img_1d[i],centroids[j])
      if (distance != 0):
        temp.append(int(distance))
    minValue = min(temp)
    minIndex = temp.index(minValue)
    labels.append(int(minIndex))
    temp = []
  labels = np.array(labels)
  return labels
```

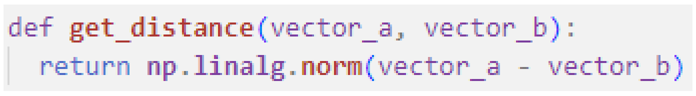
The parameter passed is the flattened image array (img_1d) and centroids.

We initialize the labels array, and temp array.

Traverse each color point contained in img_1d and calculate the distance from it to each centroid element present in the centroids array (using get_distance function). Then, add these distance values to the temp array. Next, we get the index of the smallest element in the array temp and assign it to minIndex. After that, we add minIndex to the labels.

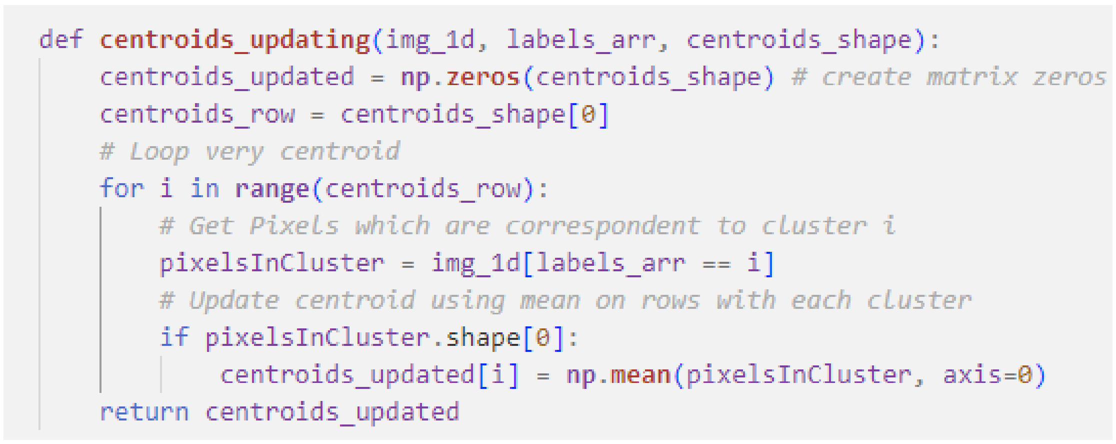
So, we know which cluster the i-th color point belongs to (each element in the labels will correspond to the index of the cluster to which each color point in the img_1d array belongs)

Note: we calculate distance between 2 vector by using numpy.linalg.norm



```python
def get_distance(vector_a, vector_b):
  return np.linalg.norm(vector_a - vector_b)
```

2.5. Update centroids



```python
def centroids_updating(img_1d, labels_arr, centroids_shape):
    centroids_updated = np.zeros(centroids_shape) # create matrix zeros
    centroids_row = centroids_shape[0]
    # Loop very centroid
    for i in range(centroids_row):
        # Get Pixels which are correspondent to cluster i
        pixelsInCluster = img_1d[labels_arr == i]
        # Update centroid using mean on rows with each cluster
        if pixelsInCluster.shape[0]:
            centroids_updated[i] = np.mean(pixelsInCluster, axis=0)
    return centroids_updated
```

The parameter passed is the flattened image array (img_1d) and labels array (labels_arr) and the shape of centroids (centroids_shape)

We create a new centroids (centroid_updated) with the shape corresponding to the current centroids. We initialize all values in centroids_updated = 0.

Then, then we loop each line in centroids (using shape[0]), get pixels which are correspondent to cluster i-th (by compare labels array with row i-th of centroids) and fetched from img_id

We update centroids by using mean on row with each cluster (use numpy.mean)

Finally, return new centroids (centroids_updated) which was updated

III. POST-PROCESSING

3.1. Reshape to the original image

```
def reshapeToOriginalImg(centroids, labels, image):
    # Reconstructing the img by replacing every single pixel with its centroid
    outputImg = centroids[labels].astype(np.uint8)
    # Reshape to the original shape
    return outputImg.reshape(image.shape)
```

After processing, we reshape the compressed image of (rows*cols, 3) dimensions to the original (rows, cols, 3) dimensions. The input parameters are centroids, labels, and the original image array (a numpy array). Then, we reconstruct the image by replacing every single pixel with its centroid. Return the output image which was processed.

3.2. Display output image and export to file

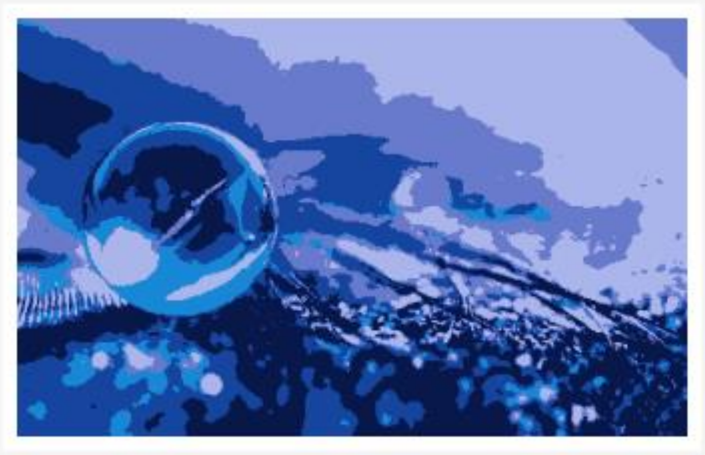
```
# Display output image
plt.axis('off') # hide axis
plt.imshow(outputImg) # display the processed image
```

We use matplotlib.pyplot library as plt to display image. Use plt.axis('off') to hide axis. Use plt.imshow to display the processed image

Original image

Output image (k_cluster = 5, max_iter = 7)

```
Image was successfully processed.
```



After display image, let user enter the image format for save:

```python
# Save image into specific format
print("\nImage was successfully processed. \n")
while True:
    outputImgFormat = input("Enter the type of image for save: (jpg/png/pdf): ")
    if (outputImgFormat == 'jpg' or outputImgFormat == 'png' or outputImgFormat == 'pdf'): break

# Export image to file
outputImgName = inputImgName + '_cluster' + str(k_clusters) + '.' + outputImgFormat
Image.fromarray(outputImg.astype(np.uint8)).save(outputImgName)
```

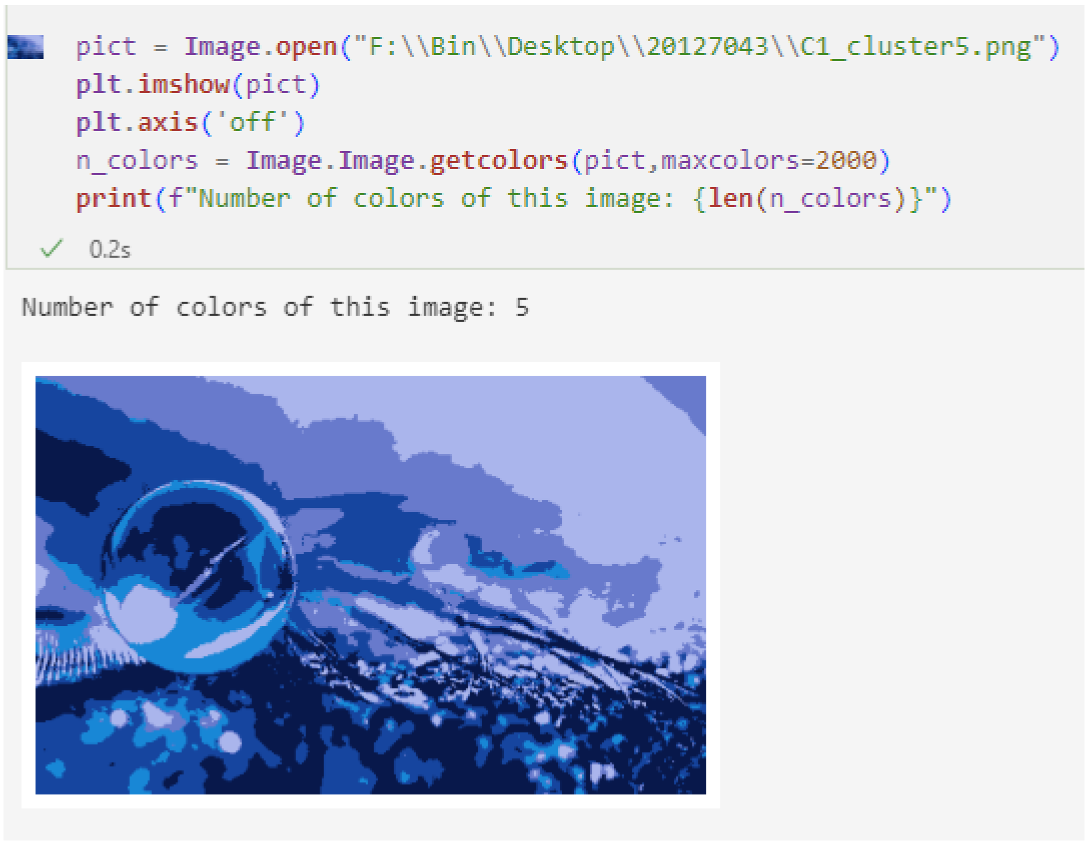
There are 3 available types: JPG / PNG / PDF
Then we export image to file with the format: inputName + k_cluster + format type

For example:

Input name is ABC.jpg ➔ User chooses k_cluster = 3 ➔ After processing ➔ User selects output format: png ➔ Output name: ABC_cluster3.png

3.3. Checking output

In this above-processed image (k_cluster = 5), I will use the python imaging library to check the number of colors in the processed image



```
pict = Image.open("F:\\Bin\\Desktop\\20127043\\C1_cluster5.png")
plt.imshow(pict)
plt.axis('off')
n_colors = Image.Image.getcolors(pict,maxcolors=2000)
print(f"Number of colors of this image: {len(n_colors)}")
```

✓ 0.2s

```
Number of colors of this image: 5
```

IV. RESULT ANALYSIS

Original image

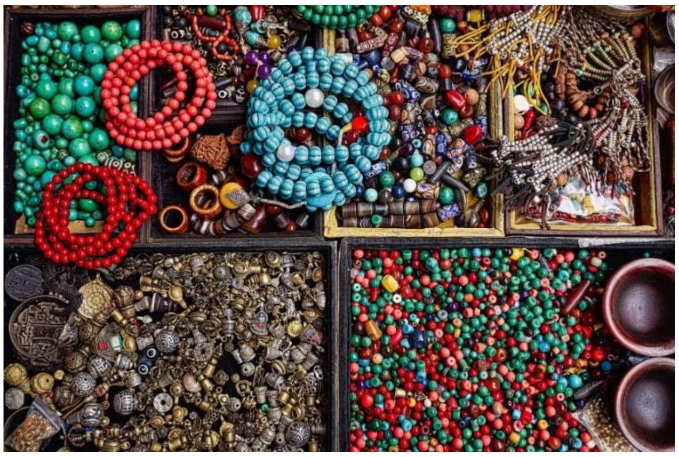
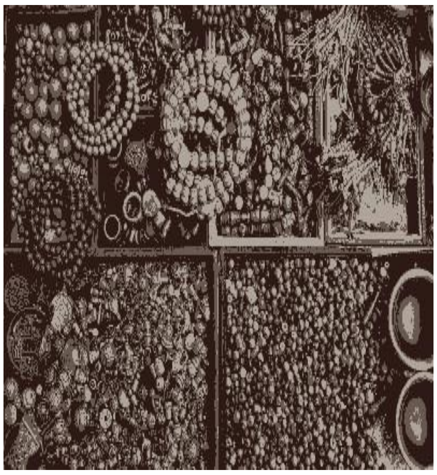
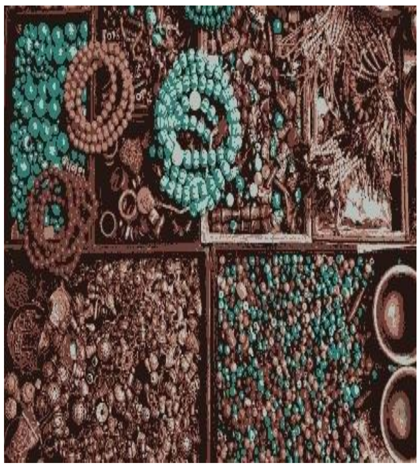
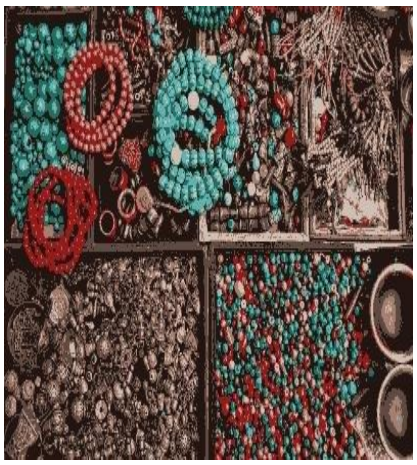
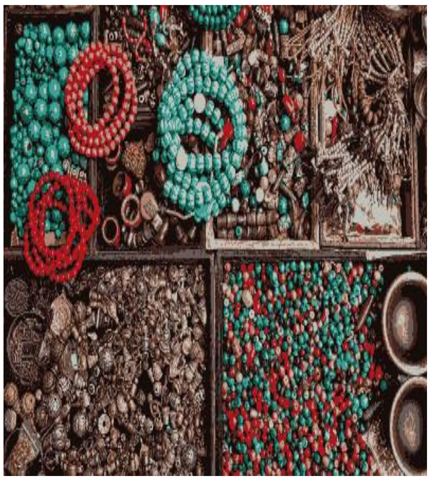
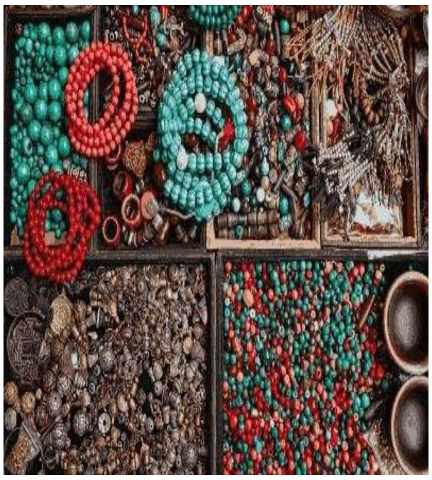
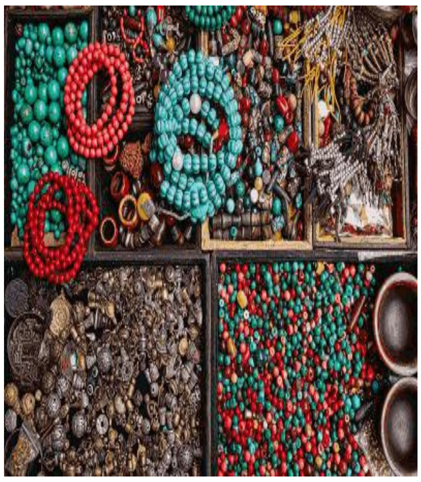


Image compression (random mode)		
		
k_cluster = 3, max_iter = 10	k_cluster = 5, max_iter = 10	k_cluster = 7, max_iter = 10
		
k_cluster = 9, max_iter = 10	k_cluster = 20, max_iter = 10	k_cluster = 50, max_iter = 10

Original image

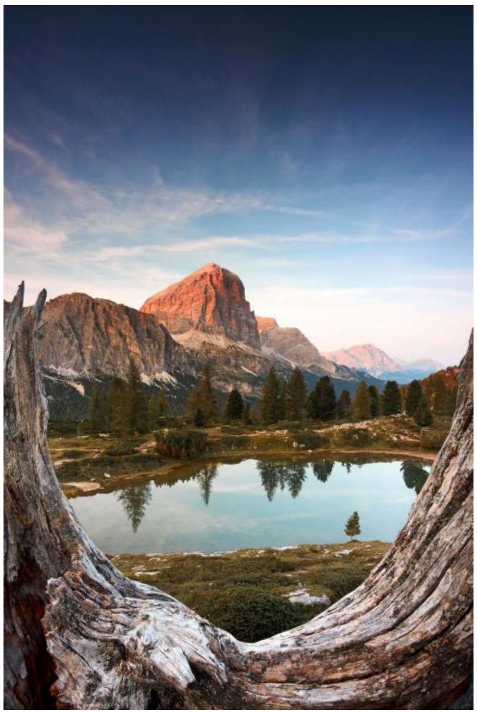
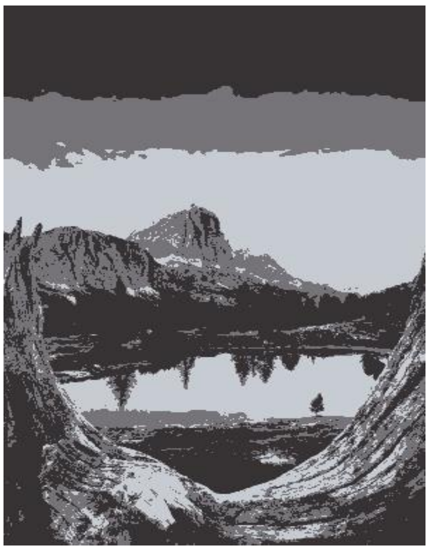
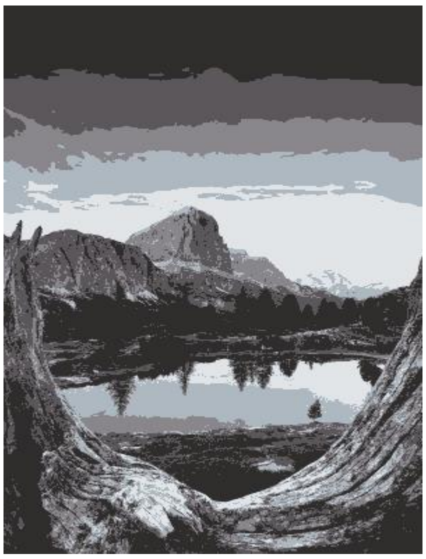
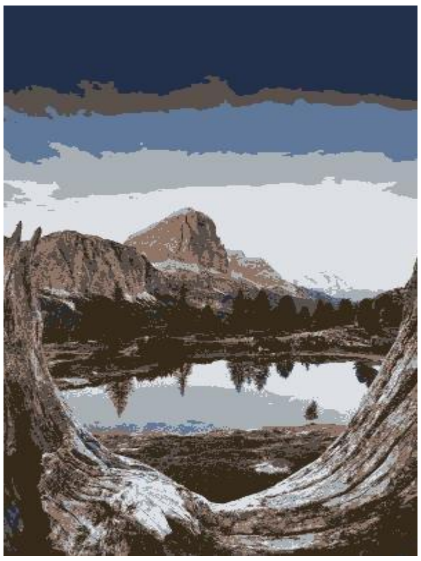
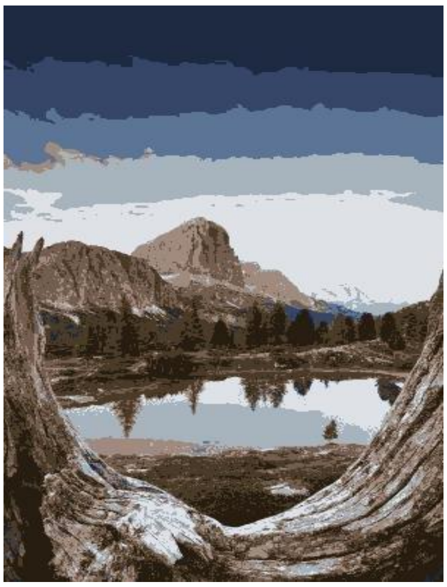
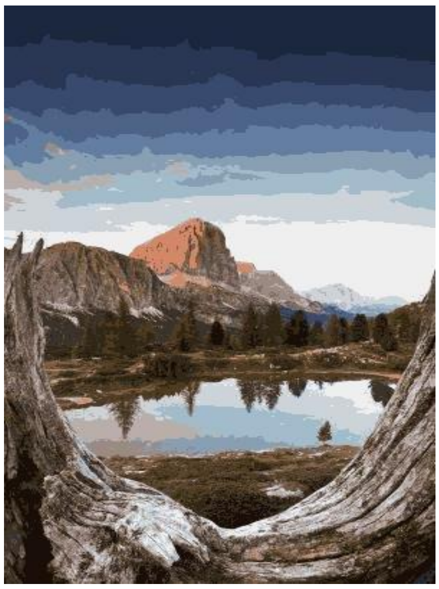
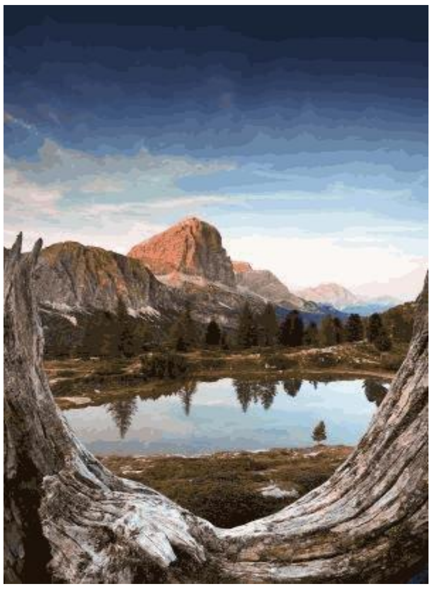


Image compression (in_pixels mode)		
		
k_cluster = 3, max_iter = 10	k_cluster = 5, max_iter = 10	k_cluster = 7, max_iter = 10

		
k_cluster = 9, max_iter = 10	k_cluster = 20, max_iter = 10	k_cluster = 50, max_iter = 10

Comment (from personal perspective):

- Based on the results, we can see that the number of k_cluster represents the number of colors returned.
- The larger k_cluster is, the more colorful and vivid will be, and the smaller k_cluster is, the fewer colors will be returned.
- But in general, the content of the image is still preserved, and the viewer can still recognize the object in the image.
- K-means uses lossy compression, so it is not possible to recover the original image from the compressed image. The larger the compression ratio, the larger the difference between the compressed image and the original image.

V. PROBLEMS

- The algorithm I made is based on the brute-force approach, so it takes a long time for medium images and a very long time for high-sharpness images.
- To be able to check the results, I had to resize the image to a smaller size (pixels of width <= 300, pixels of height <= 450).
- **Improvement**: Refer to other algorithms, refer to friends, and lecturers.

VI. REFERENCES

1. Image compression using K means clustering - OpenGenus IQ
2. Image Compression using K-Means Clustering - Towards ...
3. https://www.youtube.com/watch?v=4b5d3muPQmA&t=375s
4. https://github.com/phungvhbui/AppMath-Project-KmeansClustering