Q1 - Lists and Dictionaries (15 Marks)

To keep track of scores during IPL matches, a dictionary is used as follows:

```
{
  "ipl1": { "rohit": 57, "virat": 38 },
  "ipl2": { "smith": 9, "warner": 42 },
  "ipl3": { "rahane": 41, "tare": 63, "russel": 91 }
}
```

Each player and match is represented by a string and the scores by integers.

Input:

Stats of each match will be presented on lines over stdin:
<match_name>:<player_name>-<runs>,<player_name>-<run>,...

Output:

Construct and print the dictionary with the structure mentioned above, followed by a list containing tuples (name, total-score across all matches)for each player sorted by the total-score in descending order, (and decreasing lexicographic order of player names).

Assume that,

1. match_name will be unique for all matches
2. player_name will be unique in a given match
3. runs will always be greater than equal to 0

For instance:

Input:

```
3
match1:p1-9,p2-38
match2:p3-19,P1-49
m3:p3-1,p4-6,p1-91
```

Output:

```
{'match1':{'p1':9, 'p2':38}, 'match2':{'p3':19, 'P1':49},
'm3':{'p3':1, 'p4':6, 'p1':91}}
[('p1', 100), ('P1', 49), ('p2', 38), ('p3', 20), ('p4', 6)]
```

just print the dictionary and the list (each on a separate line), don't worry about space.

NOTE: Here P1 and p1 are different!

Q2 – Denoising the images (35 Marks)

Images are basically matrices. Grayscale (black and white) images can be represented using 2d numpy arrays. The value at each position of the matrix denotes the light intensity at that pixel in the image. We shall work with grayscale images, but, you can easily extend it to colored images. Usually, each element is an 8-bit integer and its value ranges from 0 (black) to 255 (white). But, you can use default 64-bit integers for this assignment.

Just FYI, color images are 3 dimensional numpy arrays, in which, third dimension is of size 3. This third dimension stores colour. You can imagine that it is a 2d array of triplets. Each element of a triplet gives the intensity of Red, Green and Blue colour respectively.

Here is some background. Images often contain noise. One such noise is **salt and pepper** noise. Here, some pixels get corrupted with probability p , into either black (0) or white (255). Here is an example of an image which has been corrupted with salt and pepper noise:

Here is another:

So, here is the deal: we will give you multiple (possibly overlapping) patches (along with their locations) of an image and you have to reconstruct the original image, while minimising the noise. We will also give you the overall size of image.Understand that each patch carries pixel values of only few locations. But, for each pixel in the (to be) reconstructed image, you will get many values (likely to be distinct), from different patches. Now, the challenge is to decide which value do we put inside that pixel location.

Here is an algo that you shall follow **at each pixel**of (to be) reconstructed image:
Let i_1, i_2… i_m be 'm' values that you get for a pixel. We will scan them sequentially (this is equivalent to saying that we will see patches sequentially) and maintain **4**values. Let's call them **`black_count, mid_count, white_count`**, **`mid_total`**. Suppose that we have scanned till i_t. At this point:

- **`black_count`** is the number of times, we encountered a 0.
- **`white_count`** is number of times we encountered 255.
- **`mid_count`** is the number of times we encountered something between 0 and 255 (both exclusive).
- **`mid_total`** is the sum of such values.

Once you have scanned all the patches, if **`mid_count`** is non-zero, the final value shall be the (**`mid_total`**/ **`mid_count`**) (rounded to nearest non-negative int < 256). Otherwise, it will be 0 or 255, depending on whether **`black_count`** is > **`white_count`** (which implies, if **`black_count`** <= **`white_count`**, put 255, else, put 0).

Note: `black_count <= white_count` will also happen when no patch has fallen on a pixel. But, in this case, put a 0 (not 255).

You have to do this at every pixel location in an image. There might be some pixels where no patches fall! Put 0 (black) there.

You should write a function named **`reconstruct_from_noisy_patches`**and save it into a file named **task 2.py**.
The signature of `reconstruct_from_noisy_patches`should be:

```
def reconstruct_from_noisy_patches(input_dict, shape):
      """
      input_dict:
      key: 4-tuple: (topleft_row, topleft_col, bottomright_row,
      bottomright_col): location of the patch in the original image.
      topleft_row, topleft_col are inclusive but bottomright_row,
      bottomright_col are exclusive. i.e. if M is the reconstructed matrix.
      M[topleft_row:bottomright_row, topleft_col:bottomright_col] will give the
      patch.

      value: 2d numpy array: the image patch.

      shape: shape of the original matrix.
      """
# return the reconstructed image
```

Note: you can use loops only to iterate over different patches.

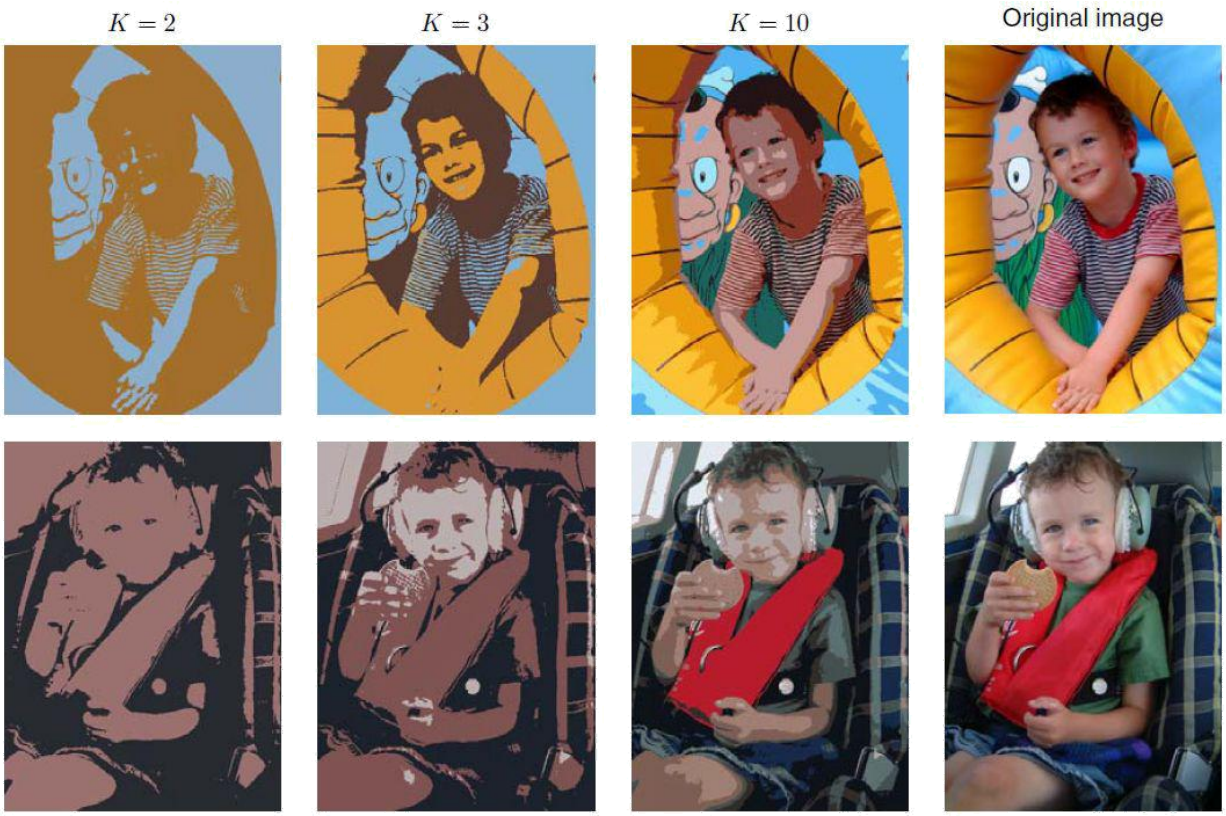
Q3 – K-clustering of images (25 Marks)

Background:

We can represent colours fairly accurately as a combination of Red, Green and Blue. In a grayscale image, we use a 2d matrix to represent the brightness of the pixel at different locations. Similarly, we can use 3 different 2d matrices to denote intensities of R, G and B colours respectively, to represent a colourful image. These 3 matrices are called colour channels, or simply, channels. These channels are stacked over each other to make a 3d array. So, basically, a colourful image can be represented as a 3d matrix, whose 3rd dimension denotes colours. In the case of a grayscale image, each pixel is a scalar intensity. But, **for coloured images, each pixel is a 3d vector**, whose components denote the intensities of RGB colours.

In this task, we would group colours to get 'flatter' but cool images. We would take advantage of the fact that colours are nothing but vectors.

There are various algorithms, which can be used to group vectors. We shall use KMeans++. You need to give it the whole set of vectors of an image along with a number **k**, and it will return you the **k**groups of these vectors. You don't need to know how it works. Just use it! Look up: scipy.cluster.vq.kmeans2.



source: Pattern Recognition and Machine Learning, Christopher M. Bishop

Problem Statement

Here is the concrete problem statement:
Write a script called task3.py, which takes following commands:

--input in, where in is the path of input image.
--k k, k in kmeans++. It's an int, ranging from 1 to 50.
--output out, where out is the path of output image

It will take the image, apply k-means++ algorithm on its pixels and use replace each pixel by the centroid of the group of its pixel and save the constructed image back.
It's all that matters. **You can use whatever you want to complete this task (loops are also allowed)**. We will just see the output image. Don't bother about checking for invalid inputs. (For example, we won't give negative numbers or floats as k).

Here is a sequence of **recommended steps to do this task**. Using some other way is also fine.

1. The array elements will be integers in range [0,255]. You should convert it into float array.
2. Convert the array into a 2d-array of shape (n, 3). n $(= nr \times nc)$ is the total number of pixels in the image. nr is number of rows in the image and nc is the number of columns in the image. Let's call this matrix M. Each row of M is a vector representing a pixel in an image. Again, the vector contains the RGB values at that pixel.
3. Pass this matrix to kmeans2with the argument **minit = '++'**.

4. Kmeans++ will give you two return values:
   a. Centroid: an array of shape (k,3). $i^{th}$is the centroid i^{th}cluster.
   b. Label: label[i] is the index of the centroid which is closest to i^{th}row of M.
5. Use this to replace each pixel by its centroid.
6. Save the image back.

Q4 - (15 marks = 7 + 3 + 2 + 3)

1. Write a function `mean_filter` to implement a mean filter of 1-d array of shape (n,) and kernel size 2k + 1 . Output should be a float array. Output size should be same as input size. Assume zero padding to achieve this. Its signature should be as follows:

```
def mean_filter(arr, k):
# applies mean filter to 1-d array with the kernel size 2k+1
# write your code here
```

2. Write another function `generate_sin_wave` which takes argument (period , range_ , num) and returns an array of shape (num ,). The contents of array are samples of the function sin (2πx/period) with given period in the given range. range_ is a 2-tuple (xmin , xmax).

3. Write a function `noisify(array, var)` which adds gaussian noise with mean 0 and variance `var` to array and returns it. It shouldn't modify the input array.

4. Create a function driver which does the following using above defined functions:

   a. Create an array (let's call it `clean_sin`) of shape (1000,) containing a sin wave of period 2 in the range $[-2, 8]$ using `generate_sin_wave` and plot it .

   b. Make another array (let's call it `dirty_sin`) by adding a gaussian noise of standard deviation 0.05. Plot it! Of course, it's shape will be same as that of `clean_sin` .

   c. Now, use mean_filter with kernel size 3 (k = 1) to clean `dirty_sin` to create an array called `cleaned_sin` . Plot it!

   d. Save the plots with respective names as: 'clean_sin.png', 'dirty_sin.png' and 'cleaned_sin.png'.

driver should be kept in a file driver.py. Other functions should be kept in a file task4.py

This task should be accomplished without any kind of loops, comprehensions or functions like np.vectorize, etc. Just like inlab.
Do not use inbuilt convolve functions for this task