**Digital Image Processing**

**Lab-04**

**Name:** Ahmad Amjad Mughal

**Reg No:** 121672

**Class:** BSCS-6C

**Task**

**Code**

#Ahmad Amjad Mughal

#121672

#BSCS-6C

#Lab-Task Using Connected-Component Algorithm

#All the required libraries and class

from PIL import Image

from PIL import ImageDraw

import sys

import math, random

from itertools import product

class UnionStructure:

def \_\_init\_\_(self):

# List of labels

self.P = []

# Name of the next label, when one is created

self.label = 0

#Assign a current label to r which is integer then we set a new name for another label and finally append that label into list P

def makeLabel(self):

r = self.label

self.label += 1

self.P.append(r)

return r

# Actually we are replacing a label of low priority with high priority as we set high pririty label as root

def setRoot(self, i, root):

while self.P[i] < i:

j = self.P[i]

self.P[i] = root

i = j

self.P[i] = root

# We find the label of root which has highest priority

def findRoot(self, i):

while self.P[i] < i:

i = self.P[i]

return i

# Finds the root of the tree containing node i

# Simultaneously compresses the tree

def find(self, i):

root = self.findRoot(i)

self.setRoot(i, root)

return root

# Joins the two trees containing nodes i and j

def union(self, i, j):

if i != j:

label1 = self.findRoot(i)

label2 = self.findRoot(j)

if label1 > label2: label1 = label2

self.setRoot(j, label1)

self.setRoot(i, label1)

def flatten(self):

for i in range(1, len(self.P)):

self.P[i] = self.P[self.P[i]]

def main():

# Create an instance of UFarray

ufArray = UnionStructure()

# Open the image

img = Image.open('example.png')

#Convert the greyScale image into Binary and set the Threshold

img = img.point(lambda p: p > 150 and 255)

img = img.convert('1')

data = img.load()

#Getting the dimensions of image width \* height

width, height = img.size

# Dictionary that stores the record of prefernces that we give to one label relative to other

labels = {}

for y, x in product(range(height), range(width)):

# If the current pixel is white, it's obviously not a component...

if data[x, y] == 255:

pass

#If the upper pixel has a black color then simply use that label for this pixel too.

elif y > 0 and data[x, y-1] == 0:

labels[x, y] = labels[(x, y-1)]

#Checking the label for right neighbour in case when both top neighbour and left top neighbour are white

elif x < width and y > 0 and data[x+1, y-1] == 0:

c = labels[(x+1, y-1)]

labels[x, y] = c

#If left-neighbour is black but top neghbour is white then simply take a union of c and d

if x > 0 and data[x-1, y] == 0:

d = labels[(x-1, y)]

ufArray.union(c, d)

elif x > 0 and data[x-1, y] == 0:

labels[x, y] = labels[(x-1, y)]

# All the neighboring pixels are white,

# Therefore the current pixel is a new component

else:

labels[x, y] = ufArray.makeLabel()

#flatten fincalizes all the labels list

ufArray.flatten()

#We make a dictionary of colors that are going to be used

colors = {}

# Image to display the components in a nice, colorful way

output\_img = Image.new("RGB", (width, height))

outdata = output\_img.load()

#loop that traverses through each label and find that label placed at location (x , y)

for (x, y) in labels:

# Assign the current Component that label

component = ufArray.find(labels[(x, y)])

# If component has not given any colored label

if component not in colors:

colors[component] = (random.randint(0,255), random.randint(0,255),random.randint(0,255))

# Colorize the image

outdata[x, y] = colors[component]

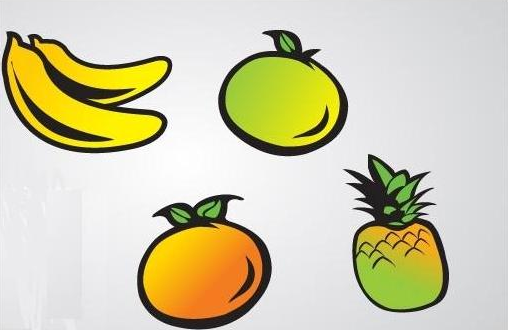
output\_img.show()

output\_img.save(‘newImage’)

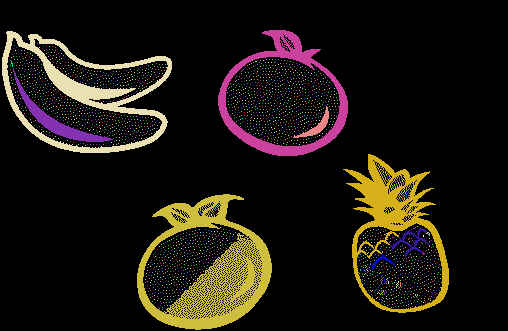
#Program execution starts from here

if \_\_name\_\_ == "\_\_main\_\_": main()

**original Image**

****

**Image after Connected-Component Algorithm**

****

**Description**

We run Connected-Component Algorithm to label with respect to our neighbour and closely related blocks. We do inter labelling and giving them colours according to their labels. We use two dictionaries one for Label and one for colour. Label dictionary contains information of pixels with respect to their (x, y) coordinates and we use random sequence of assigning colours ranging from 0-255. We use left and top neighbours our reference labels. We check whether these labels exist. If not, then simply assign a new label to that pixel or block. We use union data structure and concept of trees to check the priority of labels.