班級：資工三
學號：110590034
姓名：楊榮鈞

```python
'''
Modules import
'''
import numpy as np
import cv2

def rgb_to_gray(image):
    '''
    Convert color image to grayscale image
    '''
    gray_image = np.dot(image[..., :3], [0.3, 0.59, 0.11])
    return gray_image.astype(np.uint8)

def gray_to_binary(gray_image, threshold):
    '''
    Convert grayscale image to binary image
    '''
    binary_image = np.where(gray_image > threshold, 0, 255)
    return binary_image.astype(np.uint8)
```

Function rgb_to_gray:
使用 numpy 的 dot 去實現。
rgb 轉灰階是利用 hw1 的公式$(0.3\times R)+(0.59\times G)+(0.11\times B)$去轉成灰階。
最後用 astype(np.uint8)是因為現在的的範圍是 0~255。
回傳 gray_image。
Function gray_to_binary:
使用 numpy 的 where 去實現。
灰階轉 binary 是當大於 threshold 時，會設成 0，要是小於等於 threshold，則是設成 255，也就是背景會設為黑色。
最後用 astype(np.uint8)是因為現在的的範圍是 0~255。
回傳 binary_image。

```python
def label_components(binary_image, connectivity=4):
    '''
    Label connected components in a binary image using the classical algorithm.
    Connectivity type for labeling. Can be 4 or 8. Defaults to 4.
    Raises ValueError: If connectivity is not 4 or 8.
    '''
    labeled_image = np.zeros_like(binary_image, dtype=np.uint64)

    # Initialize Union-Find data structure
    labels = {}
    now_label = 1

    # Define neighbors based on connectivity
    # (y, x)
    if connectivity == 4:
        neighbors = [(-1, 0), (0, -1)]
    elif connectivity == 8:
        neighbors = [(-1, 0), (-1, -1), (-1, 1), (0, -1)]
    else:
        raise ValueError("Connectivity should be either 4 or 8.")

    print(connectivity)
    print(neighbors)
    print('bin=', np.unique(binary_image))
    rows, cols = binary_image.shape
```

Function label_components (分成三個截圖，這是第一個截圖):
首先利用 numpy 的 zeros_like 建立 labeled_image 的 array，其中 dtype=np.uint64 是因為 label 的數量可能會很多，但是不確定會有多少，所以先設定 np.uint64。
接著初始化 labels 的 dictionary 和 now_label，以及根據 connectivity 的值判斷我們需要判斷的 neighbors 有哪些，當 connectivity 不是 4 或是 8 時，會 raise ValueError。
其中 connectivity 為 4 的話，要判斷的方向是上方(-1, 0)和左方(0, -1)；connectivity 為 8 的話，要判斷的方向是上方(-1, 0)、左上方(-1, -1)、右上方(-1, 1)和左方(0, -1)。
根據 binary_image 的 shape 得到 labeled 的 rows(高)和 cols(寬)。

```python
# First pass
for now_y in range(rows):
    for now_x in range(cols):
        if binary_image[now_y, now_x] == 255:
            neighbor_labels = []

            for direction_y, direction_x in neighbors:
                neighbor_y, neighbor_x = now_y + direction_y, now_x + direction_x

                if 0 <= neighbor_y < rows and 0 <= neighbor_x < cols:
                    neighbor_label = labeled_image[neighbor_y, neighbor_x]

                    if neighbor_label != 0:
                        neighbor_labels.append(neighbor_label)

            if len(neighbor_labels) == 0:
                labeled_image[now_y, now_x] = now_label
                labels[now_label] = now_label
                now_label += 1
            else:
                min_label = min(neighbor_labels)
                labeled_image[now_y, now_x] = min_label
                for label in neighbor_labels:
                    if label != min_label:
                        root_label = labels[label]
                        while root_label != labels[root_label]:
                            root_label = labels[root_label]

                        if labels[min_label] == min_label:
                            labels[max(root_label, min_label)] = min(root_label, min_label)
                        else:
                            root_min_label = labels[min_label]
                            while root_min_label != labels[root_min_label]:
                                root_min_label = labels[root_min_label]
                            labels[max(root_label, root_min_label)] = min(root_label, root_min_label)
```

Function label_components (分成三個截圖，這是第二個截圖):
首先會先 pass 第一次，去標記所有 pixel 的 label，並儲存 label 到 labels 裡面，其中 labels 的 value 會儲存每個 key 的 parent(相鄰 neighbor 的最小 label)，當 labels 的 key 等於 label 時，代表這個 key 是 root。

當 binary_image 的 pixel 為 255 的時候，會初始化 neighbor_labels 的 list。
接著會找出 neighbor 的方位以及 pixel 的位置，如果位置超過圖片範圍會跳過，如果沒超出範圍，則會把這個 neighbor 的位置上的 label 存到 neighbor_labels 中。
當儲存完所有 neighbor 的 label 之後，會開始判斷現在這個 pixel 的 label 要是什麼。

如果 neighbor_labels 的長度是 0，就會把現在這個位置填入 now_label 的值，然後把這個 label 放到 labels 中，並將 key 和 value 設成 now_label，接著 now_label 加上一。

如果 neighbor_labels 有存在 label 時，會先找出 neighbor_labels 的 min_label(裡面最小的 label)，然後將現在的位置設定成 min_label。

在設定完現在位置的 label 後，我會讓 neighbor_labels 的 label 在 labels 中設定他們的 parent，所以會先找到 label 的 root_label 是誰(label 的 root)。接下來根據 min_label 是否為 root 決定他們的 value 要怎麼給。
如果 min_label 是 root 代表我只需要 min_label 跟 root_label 比較誰是 parent，然後把 parent 放到比較大的 label 中。
如果 min_label 不是 root，那我需要先找到 root_min_label(min_label 的 root)，然後比較 root_min_label 和 root_label 誰是 parent，然後把 parent 放到比較大的 label 中。

```
    # Second pass
    for y in range(rows):
        for x in range(cols):
            if labeled_image[y, x] != 0:
                root_label = labels[labeled_image[y, x]]
                while root_label != labels[root_label]:
                    root_label = labels[root_label]
                labeled_image[y, x] = root_label

    # print('labels =', labels)
    return labeled_image
```

Function label_components (分成三個截圖，這是第三個截圖):
在 pass 完第一次之後，會在 pass 第二次，這次主要是檢查每個 pixel 的 label 是否為 root，並把不是 root 的 label 全部換成 root(把不同 label 但是連在一起的 pixel 的 label 全部設成 root 的 label)。

首先判斷每個有 label 的 pixel 的 label 是否為 root。當 label 是 root 的話，就會檢查下一個 pixel；當 label 不是 root 的話，根據第一次儲存的 labels 的所有 label 找到 root，然後把現在 pixel 的 label 換成 root_label，當換完 pixel 後，會再檢查下一個 pixel。
最後回傳 labeled_image。

```
def label_to_color(labeled_image):
    '''
    Assign color to label
    '''
    unique_labels = np.unique(labeled_image)
    print(unique_labels)
    colors = np.random.randint(100, 256, (len(unique_labels), 3), dtype=np.uint8)
    colored_image = np.zeros((labeled_image.shape[0], labeled_image.shape[1], 3), dtype=np.uint8)
    for label, color in zip(unique_labels, colors):
        if label != 0:
            colored_image[labeled_image == label] = color
    return colored_image
```

Function label_to_color:
首先會用 numpy 的 unique 找出所有 label，並存在 unique_labels 中。

根據 unique_labels 的長度給出相同長度的隨機顏色存入 colors 中，並將 colors 設定 dtype=np.uint8，因為顏色的 value 範圍是 100~255，然後將顏色的 value 的最小值設為 100 是防止太小的 value 會讓顏色太暗，跟背景的黑色不容易辨識。
將 colored_image 初始化，利用 numpy 的 zeros 讓他的長和寬跟 labeled_image 一樣，然後分成 RGB 三個 value，dtype=np.uint8 是因為 RGB value 的範圍為 0~255。
接著把所有 labeled_image 不為 0 的 label 的 pixel 根據 unique_labels 和 colors 把 colored_image 的 RGB 填上隨機顏色。
最後回傳 colored_image。

```
def image(number, threshold):
    '''
    For img{number}.png
    '''
    image = cv2.imread(f'images/img{number}.png')
    gray_image = rgb_to_gray(image)
    binary_image = gray_to_binary(gray_image, threshold)

    labeled_image_4 = label_components(binary_image, connectivity=4)
    labeled_image_8 = label_components(binary_image, connectivity=8)

    colored_image_4 = label_to_color(labeled_image_4)
    colored_image_8 = label_to_color(labeled_image_8)
    cv2.imshow('Original Image', image)
    cv2.imshow('Grayscale Image', gray_image)
    cv2.imshow('Binary Image', binary_image)
    cv2.imshow('4 Connected Components', colored_image_4)
    cv2.imshow('8 Connected Components', colored_image_8)
    cv2.imwrite(f'results/img{number}_4.png', colored_image_4)
    cv2.imwrite(f'results/img{number}_8.png', colored_image_8)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Function image:
先用 cv2 的 imread 讀取圖片到 image，然後利用 rgb_to_gray 把 image 轉成 gray_image，再利用 gray_to_binary 把 gray_image 轉成 binary_image。
接下來用 label_components 把 binary_image 轉成 labeled_image_4 和 labeled_image_8 (在參數設定 connectivity=4，會使用 4-connected，在參數設定 connectivity=8，會使用 8-connected)。
接著利用 label_to_color 將 labeled_image 的 label 填上隨機顏色，把 labeled_image_4 轉成 colored_image_4 還有 labeled_image_8 轉成 colored_image_8。

利用 cv2 的 imshow 查看 original image、gray image、binary image、4-connected 的 labeled_image 加上隨機顏色的結果和 8-connected 的 labeled_image 加上隨機顏色的結果。
利用 cv2 的 imwrite 將 4-connected 的 labeled_image 加上隨機顏色的結果和 8-connected 的 labeled_image 加上隨機顏色的結果存到 results 裡面。
利用 cv2 的 waitKey(0)和 destroyAllWindows()將 imshow 用顯示出來的圖片關閉。

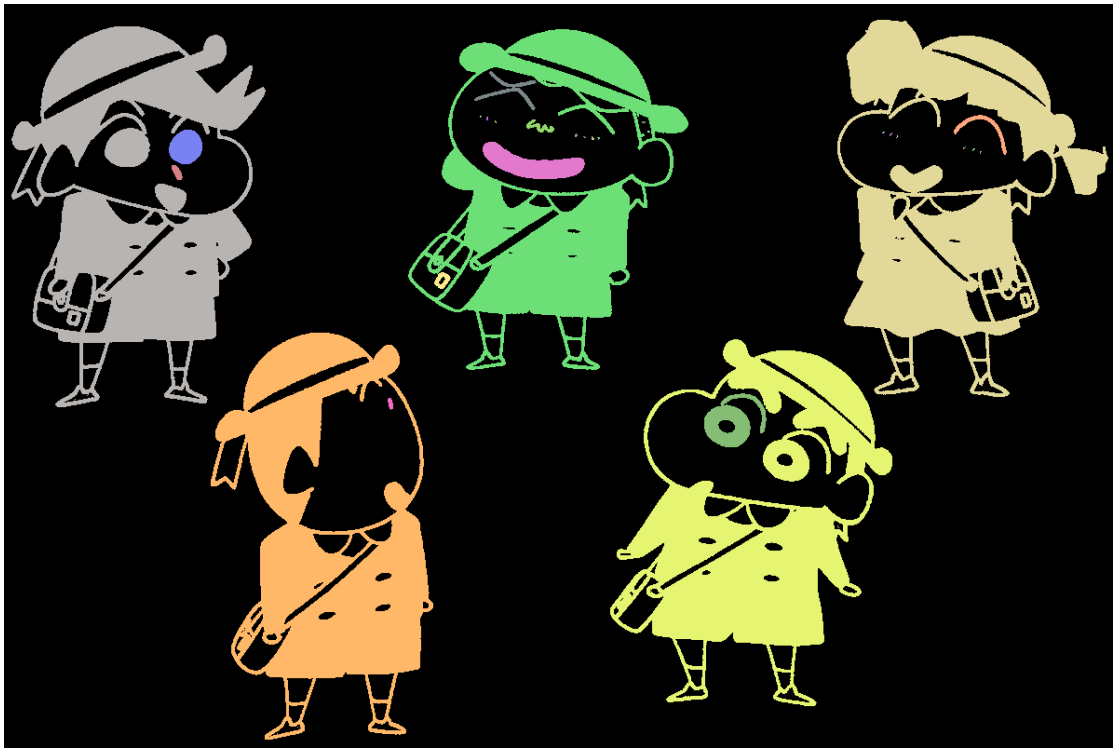
```
if __name__ == '__main__':
    image(number=1, threshold=170)
    image(number=2, threshold=173)
    image(number=3, threshold=206)
    image(number=4, threshold=240)
```

執行 image，產生所有助教給的圖片的 4-connected 的 labeled_image 加上隨機顏色的結果和 8-connected 的 labeled_image 加上隨機顏色的結果。

Result images

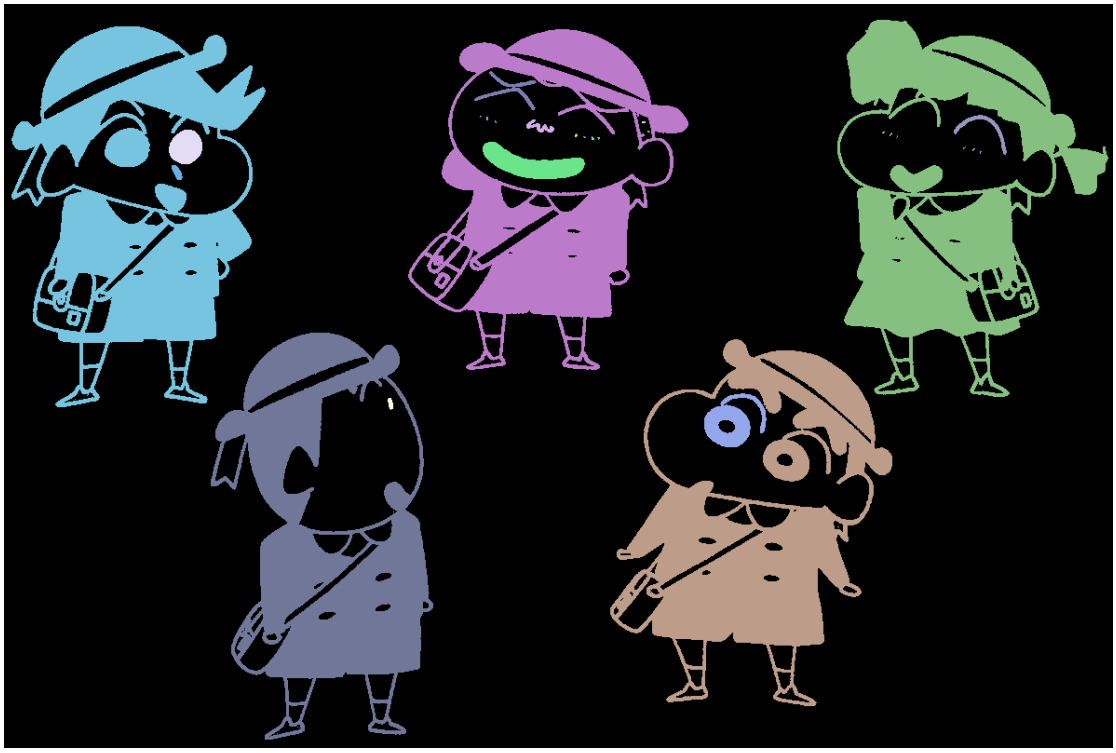
這次的 Result images 有很多縮小的時候看起來像是分開的圖案但是顏色相同，在放大檢視後，可以發現 pixel 是連在一起的，所以圖案的顏色才會是相同的。

img1_4.png



我預期的結果跟最後的輸出相同，跟 8-connected 相差比較明顯的是書包的扣子和旁邊的方塊，4-connected 的結果是分開的。

img1_8.png



我預期的結果跟最後的輸出相同，跟 4-connected 相差比較明顯的地方是書包的扣子和旁邊的方塊，8-connected 的結果是連在一起。

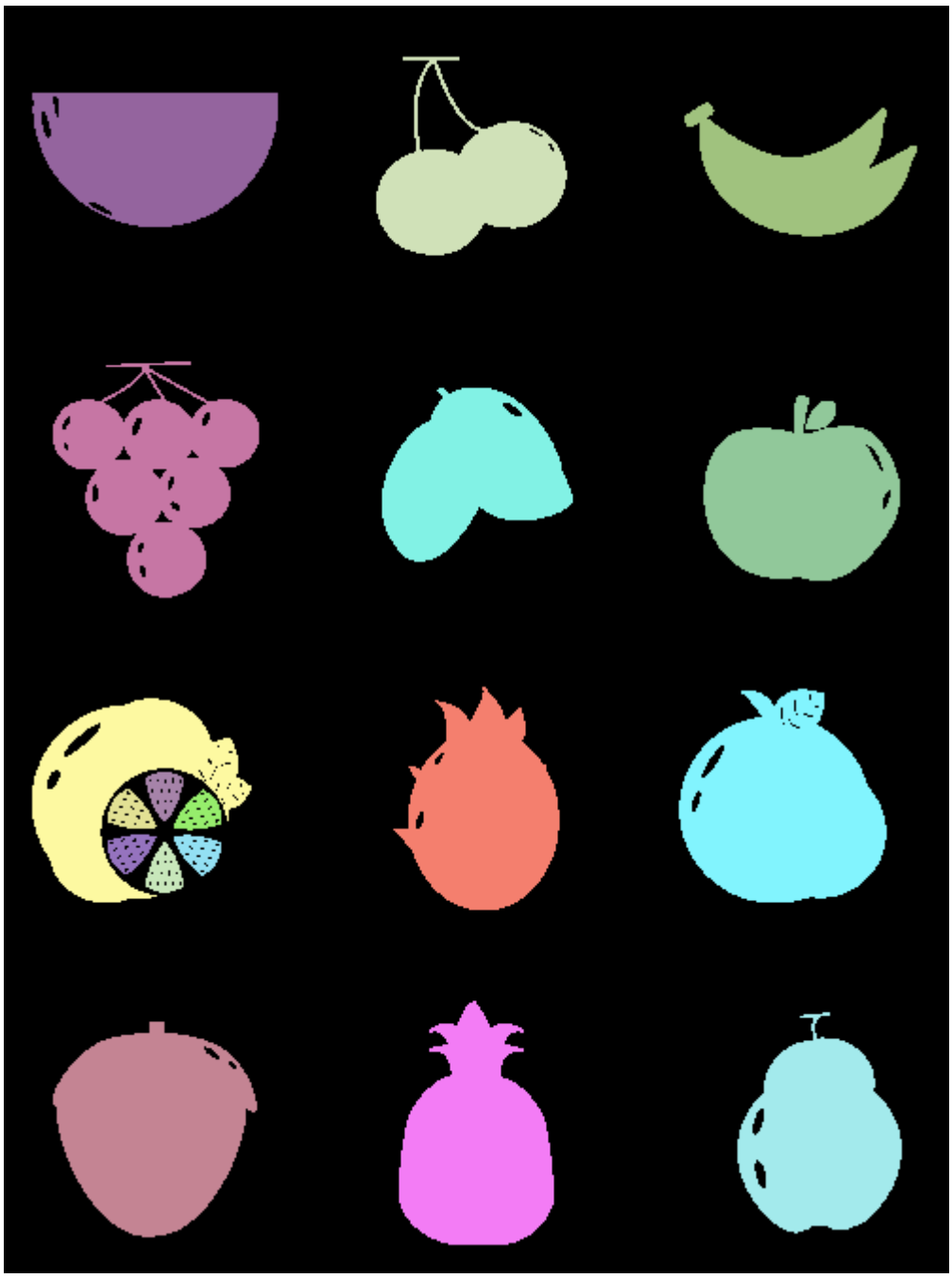
img2_4.png

我預期的結果跟最後的輸出相同，放大看 mm 巧克力的袋子的左下方的數字(217)可以很明顯的看出 4-connected 和 8-connected 的差別，左上的 mm 巧克力球的邊邊可以看到有點藍灰的顏色，4-connected 是分開的，而 8-connected 的部分是連起來的。

img2_8.png

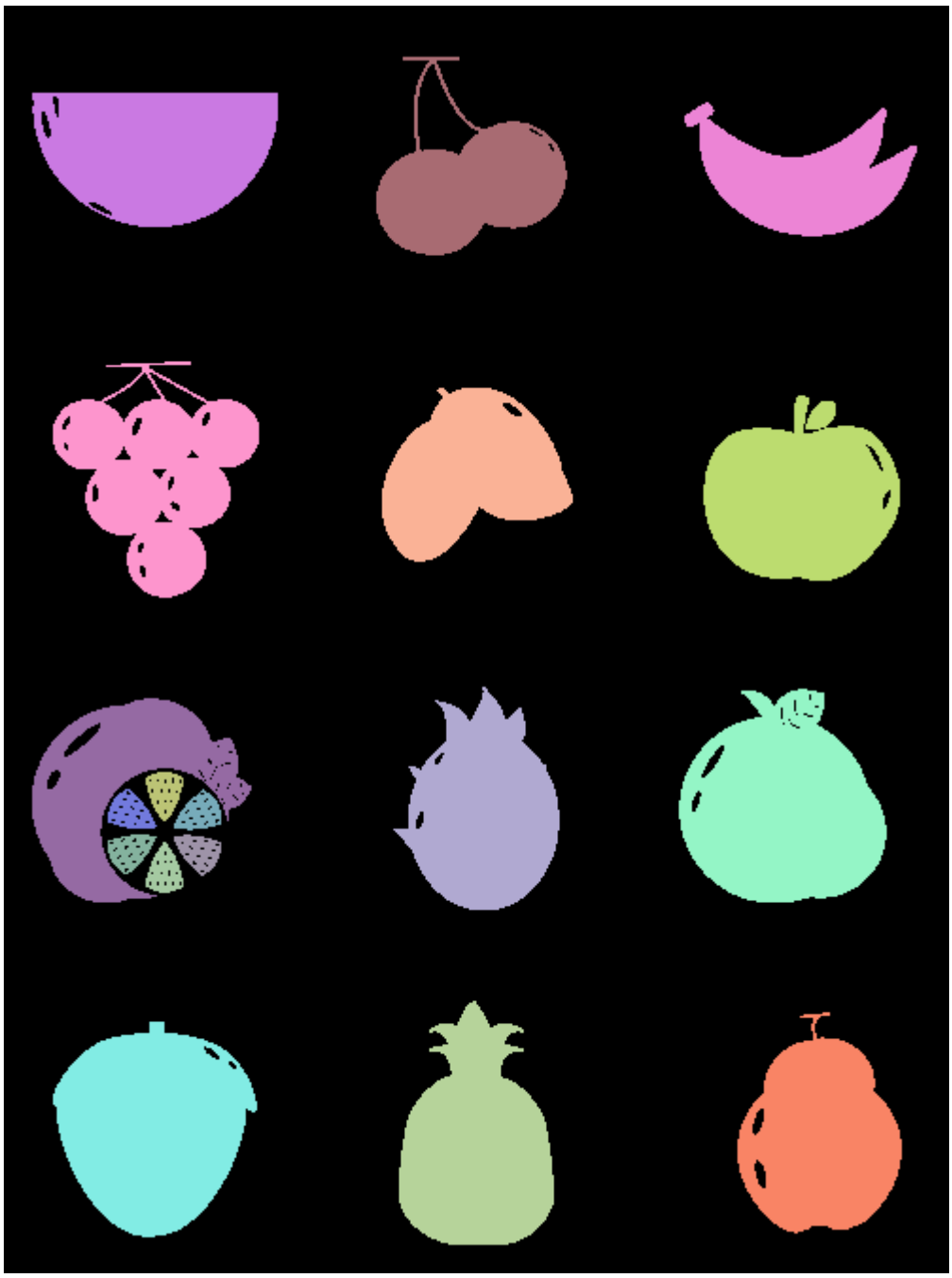
我預期的結果跟最後的輸出相同，放大看 mm 巧克力的袋子的左下方的數字(217)可以很明顯的看出 4-connected 和 8-connected 的差別，左上的 mm 巧克力球的邊邊可以看到跟 4-connected 的差別在 8-connected 的顏色跟包裝的顏色一樣。

img3_4.png



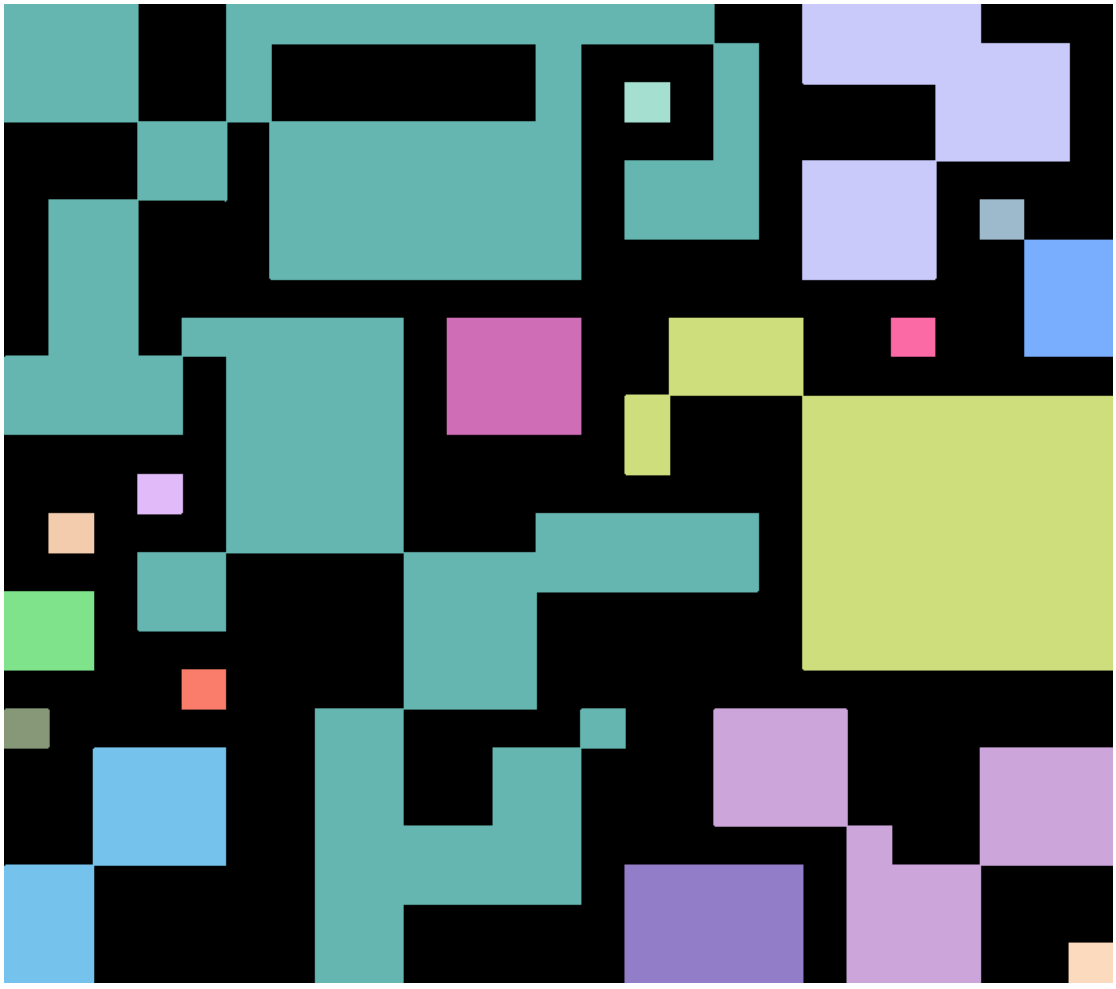
我預期的結果跟最後的輸出相同，跟 8-connected 的 label 數量一樣，所以只有顏色不同。

img3_8.png



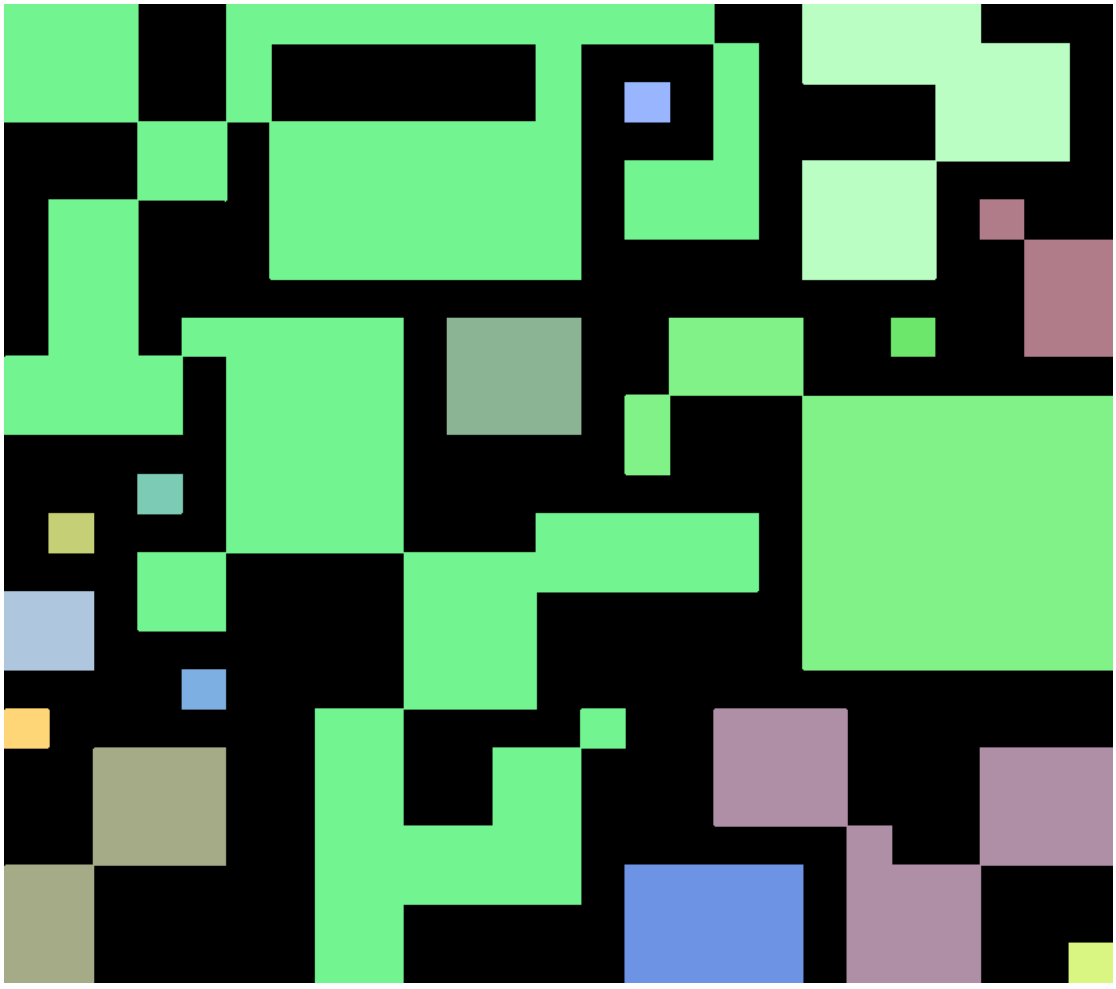
我預期的結果跟最後的輸出相同，跟 4-connected 的 label 數量一樣，所以只有顏色不同。

img4_4.png



我預期的結果跟最後的輸出相同，雖然有很多方塊是連在一起的但是從右上方塊中，可以明顯看出 8-connected 跟 4-connected 的差別，4-connected 的方塊是分開的。

img4_8.png



我預期的結果跟最後的輸出相同，雖然有很多方塊是連在一起的但是從右上方塊中，可以明顯看出 8-connected 跟 4-connected 的差別，8-connected 的方塊是連在一起的。

最後的結果總結：
在一開始寫 hw2 的時候有兩個難點，第一個是 threshold 的調整，沒調好 threshold 會讓圖片的結果變得很奇怪，像是有些圖片會有很 pixel 連在一起，會讓 4-connected 和 8-connected 的結果看起來差不多。
第二個是在寫 label component 的 function 的時候，我一開始在 pixel 有 neighbor 的時候，是直接將 pixel 設定為 min_label，然後把所有在 labels 中非 min_label 的 key 的 parent 全部設為 min_label，這個寫法導致我 loss 掉許多原本非 min_label 的 key 的 parent，因此圖片會出現很多 pixel 連在一起但是不同的 label。
而出現很多 pixel 連在一起但是不同的 label 的這個問題的解決方法是，我開始找非 min_label 的 label 的 root(root_label)還有 min_label 的 root(min_root_label)，然後去比較這兩個 root 誰比較小，比較小的 label 設定為

比較大的 label 的 parent。
這次 hw2 中，我有更深刻的解到 label component 的實作方法，也發現 binary image 的 threshold 對 label component 的重要性。