Problem 1:

Histogram Computation: Compute the histogram of the image coins.png, by finding the frequency of pixels for each intensity level {0,1,...,255}. Show the histogram by plotting frequencies w.r.t. intensity levels. Comment on what you observe. Also, find the average intensity of the image using this histogram. Verify the result with the actual average intensity.

Function: Histogram

Input: Grayscale image

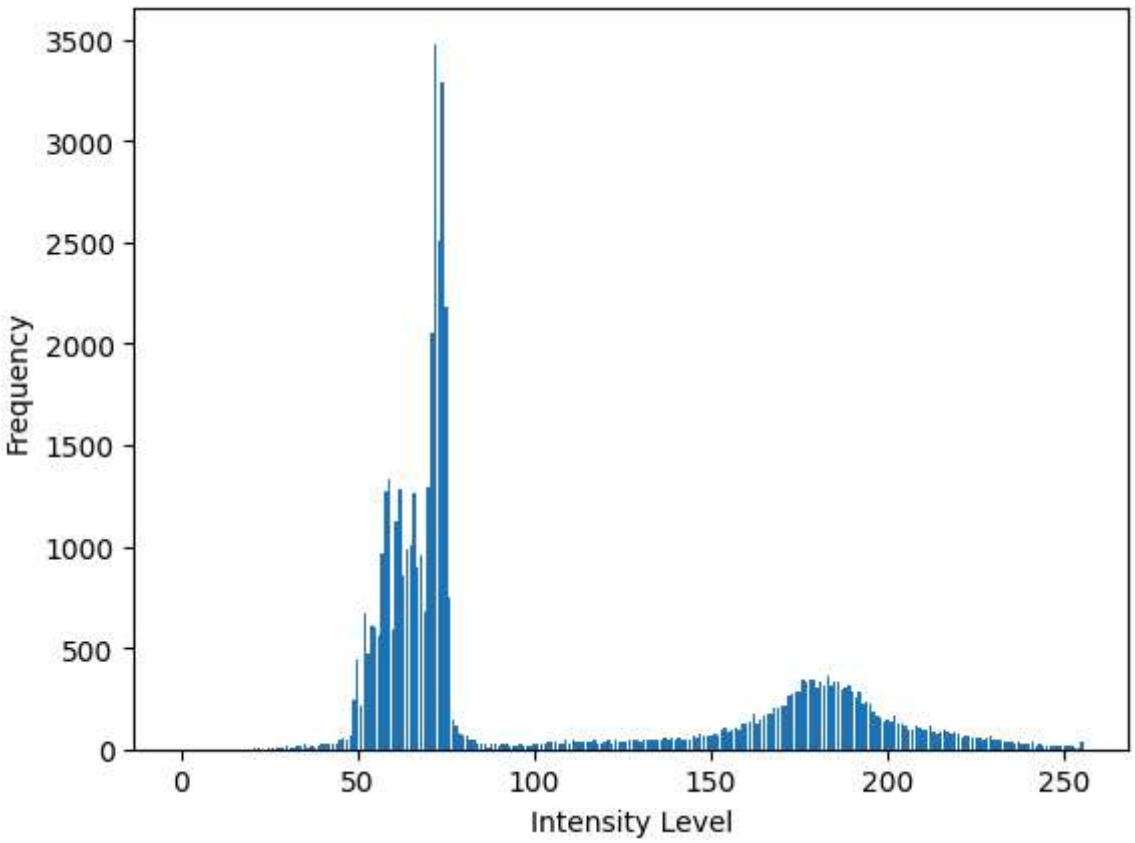
Output Frequencies at each intensity level (a list/vector of size 256)

In [4]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg

def Histogram(image_name):
    # Read the grayscale image (coins.png)
    img = mpimg.imread(image_name)
    shape_0 = img.shape

    # Normalize the image to 0-255 range
    img = (255 * img).astype(int)

    # Count occurrences for each intensity level
    img_freq = np.zeros(256, dtype=int)
    for l in range(256):
       img_freq[l] = np.count_nonzero(img == l)

    # Average calculated using Histogram
    w_t = np.sum(img_freq)
    mean_hist = 0
    for j in range(256):
        mean_hist += j*img_freq[j]/w_t

    mean_act = np.sum(img)/(shape_0[0]*shape_0[1])
    print("Actual average: {} \nHistogram average: {} ".format(mean_act, mean_hist)


    return img_freq



img_freq = Histogram("coins.png")


fig, ax1 = plt.subplots(1, 1)
# Plot the histogram
```

```
ax1.bar(np.arange(256), img_freq)
ax1.set_xlabel("Intensity Level")
ax1.set_ylabel("Frequency")
```

```
Actual average: 103.30500158906722
Histogram average: 103.30500158906722
```

Out[4]: Text(0, 0.5, 'Frequency')



Comment on Problem 1

We can see in the histogram that there are majorly two groups where the intensity of pixels falls. The first group has a high frequency around lower intensity levels (around 40-80), indicating a concentration of darker pixels, while the second group appears around higher intensity levels (around 140-250), suggesting a cluster of brighter pixels. During binarization, the threshold value should be set between 90 and 130 to effectively separate these two groups. also average intensity calculate using histogram is equal to actual average.

Problem 2

Otsu's Binarization: In the class, we showed that $\sigma_w^2(t) + \sigma_b^2(t) = \sigma_t^2(t)$, where t is the threshold for binarization. Binarize the image coins.png by finding the optimal threshold in t by: (a) Minimizing the within class variance $\sigma_w^2(t)$ over t. (b) Maximizing the between class variance $\sigma_b^2(t)$ over t. Verify that both methods are equivalent. Plot $\sigma_w^2(t)$, $\sigma_b^2(t)$ and $\sigma_w^2(t)$ +

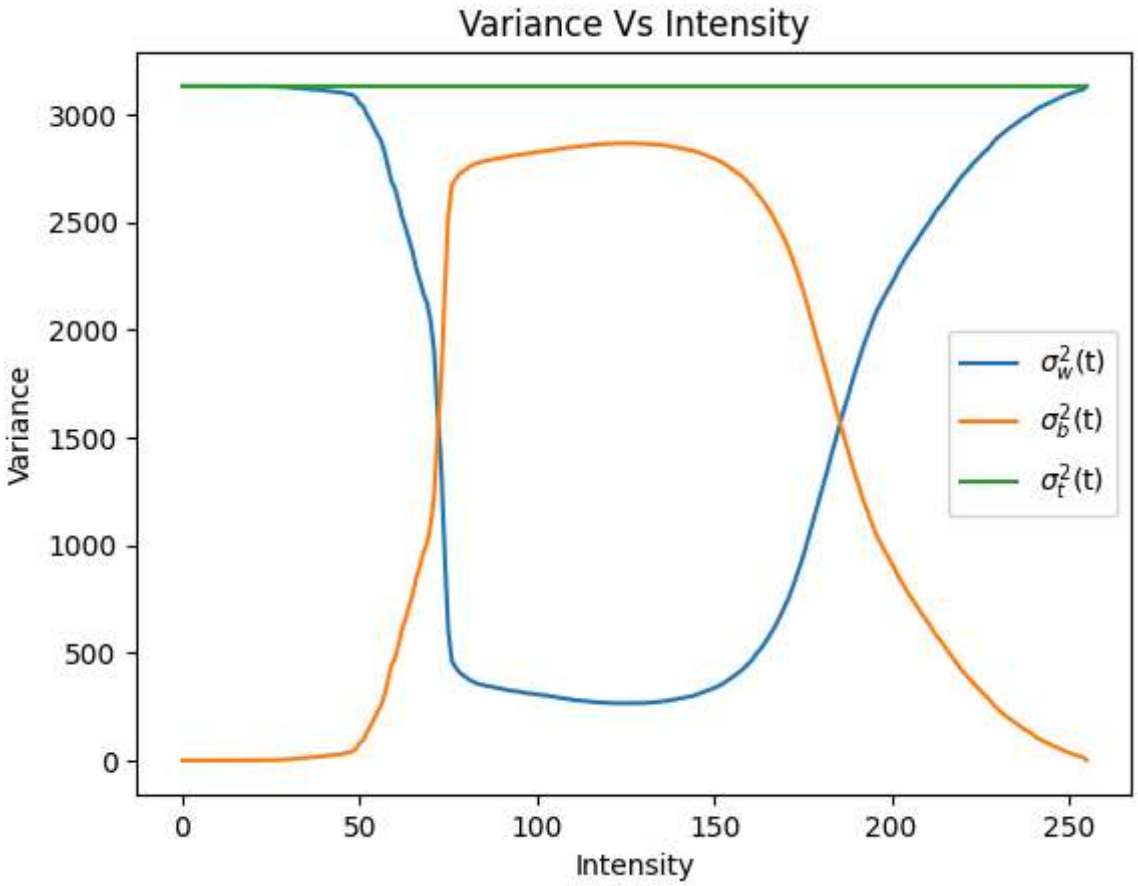
$\sigma_b^2(t)$ w.r.t. t. Function Within class variance Function Between class variance Input Grayscale image, threshold Input Grayscale image, threshold Output Within class variance Output Between class variance

In [12]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg

def Between_class_variance(image_name, threshold):


    var_wb_lst = np.zeros((256,1), dtype = int)
    img_freq = Histogram(image_name)
    #print(coins_array)
    w_t = np.sum(img_freq)

    mean_t = 0
    var_t = 0
    for j in range(256):
        mean_t += j*img_freq[j]/w_t

    for j in range(256):
        var_t += ((j-mean_t)**2)*img_freq[j]/w_t

    for t in range(256):
        mean_0 = 0
        mean_1 = 0
        var_0 = 0
        var_1 = 0
        w_0 = 0
        w_1 = 0
        for i in range(t + 1):
            w_0 += img_freq[i]

        for j in range(t + 1, 256):
            w_1 += img_freq[j]

        if  w_0 > 0:
            for i in range(t + 1):
                mean_0 += i * img_freq[i] / w_0
        if t < 255 and w_1 > 0:
            for j in range(t + 1, 256):
                mean_1 += j * img_freq[j] / w_1

        var_b = int((w_0*w_1*(mean_0-mean_1)**2)/(w_t**2))
        var_wb_lst[t] = var_b
    return var_wb_lst


def Within_class_variance(image_name, threshold):


    var_wb_lst = np.zeros((256,1), dtype = int)
    img_freq = Histogram(image_name)
    #print(coins_array)
```

```
    w_t = np.sum(img_freq)

    mean_t = 0
    var_t = 0
    for j in range(256):
        mean_t += j*img_freq[j]/w_t

    for j in range(256):
        var_t += ((j-mean_t)**2)*img_freq[j]/w_t

    for t in range(256):
        mean_0 = 0
        mean_1 = 0
        var_0 = 0
        var_1 = 0
        w_0 = 0
        w_1 = 0
        for i in range(t + 1):
            w_0 += img_freq[i]

        for j in range(t + 1, 256):
            w_1 += img_freq[j]

        if  w_0 > 0:
            for i in range(t + 1):
                mean_0 += i * img_freq[i] / w_0
        if t < 255 and w_1 > 0:
            for j in range(t + 1, 256):
                mean_1 += j * img_freq[j] / w_1


        if  w_0 > 0:
            for i in range(t + 1):
                var_0 += ((i-mean_0)**2) * img_freq[i] / w_0
        if t < 255 and w_1 > 0:
            for j in range(t + 1, 256):
                var_1 += ((j-mean_1)**2) * img_freq[j] / w_1
        var_w = int((w_0*var_0 + w_1*var_1)/w_t)
        var_wb_lst[t] = var_w

    return var_wb_lst



var_w_arr = Within_class_variance("coins.png",0)
var_b_arr = Between_class_variance("coins.png",0)
print("Intensity Which minimizes inter class variance: {}".format(np.argmin(var_w_a
print("Intensity Which maximizes intra class variance: {}".format(np.argmax(var_b_a

plt.plot(np.arange(256),var_w_arr[:], label = "$σ^2_w$(t)")
plt.plot(np.arange(256),var_b_arr[:], label = "$σ^2_b$(t)")
plt.plot(np.arange(256),var_b_arr[:] + var_w_arr[:], label = "$σ^2_t$(t)")


plt.xlabel("Intensity")
```

```
plt.ylabel("Variance")
plt.title("Variance Vs Intensity")

plt.legend()
plt.show()
```

```
Actual average: 103.30500158906722
Histogram average: 103.30500158906722
Actual average: 103.30500158906722
Histogram average: 103.30500158906722
Intensity Which minimizes inter class variance: 123
Intensity Which maximizes intra class variance: 123
```



Comment on Problem 2

In the "Variance vs Intensity" plot, we observe that the within-class variance ($\sigma_w^2(t)$) decreases as the intensity increases, while the between-class variance ($\sigma_b^2(t)$) initially increases, peaks around a certain intensity level, and then decreases. This peak represents the optimal threshold for separating the two classes. Meanwhile, the total variance ($\sigma_t^2(t)$) remains relatively constant. The optimal threshold for binarization should be set around the intensity level where $\sigma_b^2(t)$ is maximized and $\sigma_w^2(t)$ is minimized, ensuring maximum separability between the classes. It can also be noted that the maximum value of $\sigma_b^2(t)$ occurs at the same intensity where $\sigma_w^2(t)$ is minimized.

To calculate the threshold, it is preferable to use the function "Between_class_variance" instead of the "Within_class_variance" function. The "Between_class_variance" function is computationally less expensive and more efficient.

Problem 3

Adaptive Binarization: Divide the image sudoku.png into N ×N non-overlapping blocks (for e.g., given an 80×120 image with 8×8 blocks, there are 64 blocks of size 10×15). Apply Otsu's binarization on all blocks independently, and stitch them back together into the original image shape. Compare the results for the following: (a) Binarization on the full image (b) Adaptive binarization applied on i. 2 ×2 blocks ii. 4 ×4 blocks iii. 8 × 8 blocks iv. 16 ×16 blocks

In [20]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
```

In [38]:
```
def Histogram1(image_array):
    shape_0 = image_array.shape


    img_freq = np.zeros(256, dtype=int)
    for l in range(256):
        img_freq[l] = np.count_nonzero(image_array == l)


    return img_freq
```

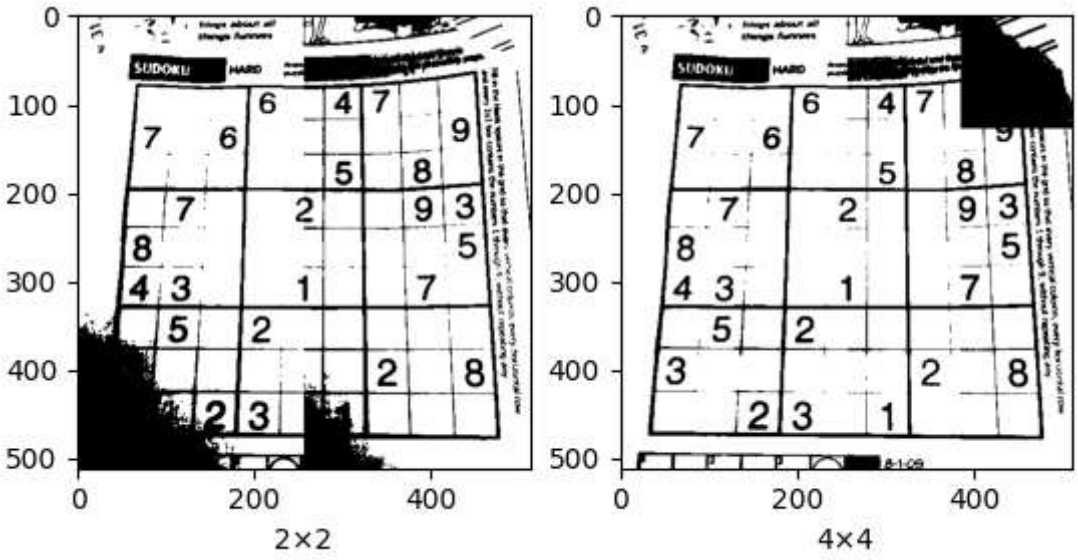
In [39]:
```
def Between_class_variance1(image_array):

    var_wb_lst = np.zeros((256,1), dtype = int)
    img_freq = Histogram1(image_array)
    #print(coins_array)
    w_t = np.sum(img_freq)

    mean_t = 0
    var_t = 0
    for j in range(256):
        mean_t += j*img_freq[j]/w_t

    for j in range(256):
        var_t += ((j-mean_t)**2)*img_freq[j]/w_t

    for t in range(256):
        mean_0 = 0
        mean_1 = 0
        var_0 = 0
        var_1 = 0
        w_0 = 0
```

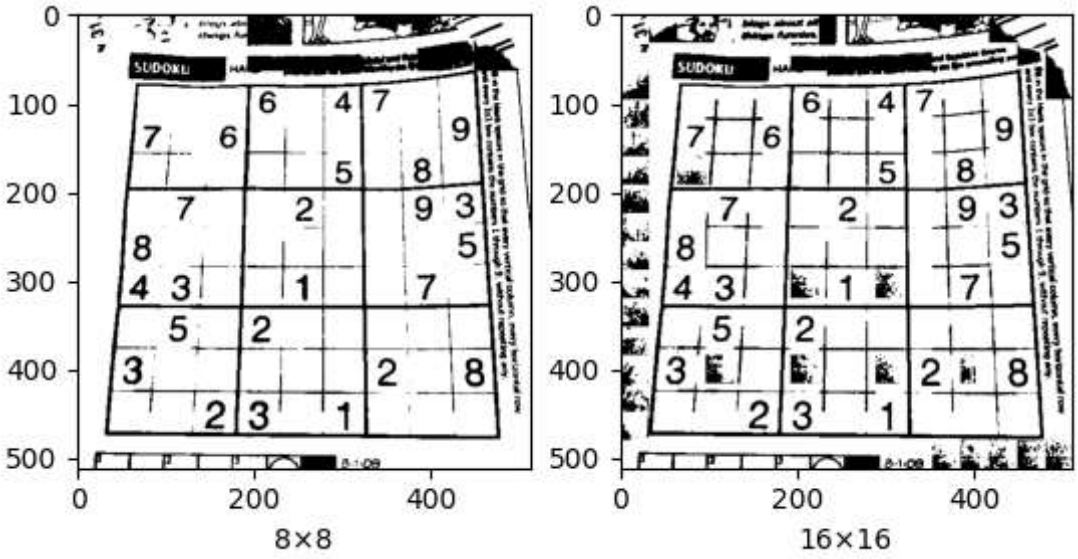
```
        w_1 = 0
        for i in range(t + 1):
            w_0 += img_freq[i]

        for j in range(t + 1, 256):
            w_1 += img_freq[j]

        if  w_0 > 0:
            for i in range(t + 1):
                mean_0 += i * img_freq[i] / w_0
        if t < 255 and w_1 > 0:
            for j in range(t + 1, 256):
                mean_1 += j * img_freq[j] / w_1

        var_b = int((w_0*w_1*(mean_0-mean_1)**2)/(w_t**2))
        var_wb_lst[t] = var_b
    return var_wb_lst
```

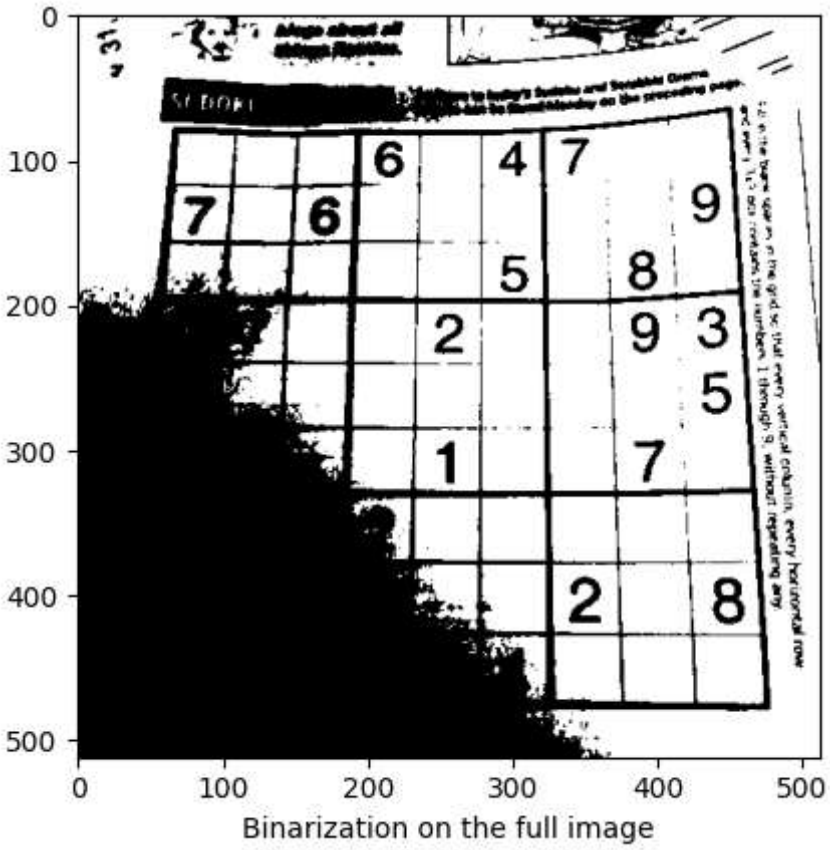
In [40]:
```
def Adaptive_binarization1(image_name, N):
    sudoku_img = mpimg.imread(image_name)
    shape_0 = sudoku_img.shape

    sudoku_img_o = np.zeros(shape_0, dtype=int)
    block_xdim = int(shape_0[0]/N)
    block_ydim = int(shape_0[1]/N)


    for jj in range(0, shape_0[0], block_xdim):
        for kk in range(0, shape_0[1],block_ydim):
            # Ensure block slicing does not go out of bounds
            subarray = sudoku_img[jj:min(jj + block_xdim, shape_0[0]), kk:min(kk +
            block = (255 * subarray).astype(int)

            # Assuming Within_Class_Variance returns a 3-element array
            var_block_wbt = Between_class_variance1(block)
            threshold_block = np.argmax(var_block_wbt[:])  # Correct use of var_blo

            # Loop through the block
            for j in range(block_xdim):
                for k in range(block_ydim):
                    if block[j, k] > threshold_block:
                        sudoku_img_o[jj + j, kk + k] = 255
                    else:
                        sudoku_img_o[jj + j, kk + k] = 0

    return sudoku_img_o
```

In [41]:
```
sudoku_img = mpimg.imread("sudoku.png")
shape_0 = sudoku_img.shape

sudoku_binary_img_2 = Adaptive_binarization1("sudoku.png", 2)
sudoku_binary_img_4 = Adaptive_binarization1("sudoku.png", 4)
sudoku_binary_img_8 = Adaptive_binarization1("sudoku.png", 8)
sudoku_binary_img_16 = Adaptive_binarization1("sudoku.png", 16)
sudoku_binary_img_full_image = Adaptive_binarization1("sudoku.png", 1)
```

In [42]:
```
# Create subplots for the first set of images
fig, ax = plt.subplots(1, 2)  # 1 row, 2 columns
ax[0].imshow(sudoku_binary_img_2, cmap='gray')
ax[0].set_xlabel("2×2")
ax[1].imshow(sudoku_binary_img_4, cmap='gray')
ax[1].set_xlabel("4×4")

# Create subplots for the second set of images
fig, ax1 = plt.subplots(1, 2)  # 1 row, 2 columns
ax1[0].imshow(sudoku_binary_img_8, cmap='gray')
ax1[0].set_xlabel("8×8")
ax1[1].imshow(sudoku_binary_img_16, cmap='gray')
ax1[1].set_xlabel("16×16")

# Show all subplots
plt.show()

# Display the final image separately
plt.imshow(sudoku_binary_img_full_image, cmap='gray')
plt.xlabel("Binarization on the full image")
plt.show()
```







Comment on Problem 3

The right top corner of the original image("sudoku.png") has high intensity, while the left bottom corner has low intensity, creating a gradient from high intensity in the top right to low intensity in the bottom left. As a result, during full-image binarization, the left bottom corner becomes black, while the right top corner becomes white. In adaptive binarization, the image is binarized in sections, which takes into account intensity variations within each block. As we increase the number of blocks, the quality of the binarized image initially

improves, but beyond a certain point, further increasing the number of blocks will result into prsesence of artifacts and noise(for example 256*256).

compare full image buinariozation and block image

Problem 4

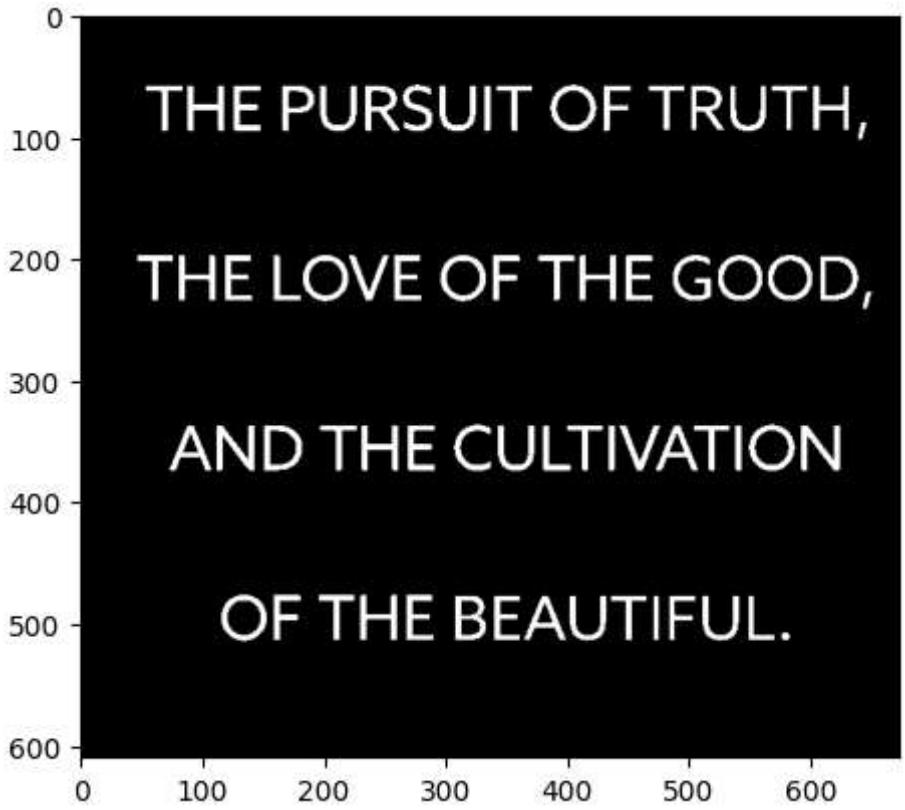
Connected Components: Binarize the image quote.png and count the total number of characters excluding punctuations using connected component analysis. In the class, we have seen connected component analysis using 4-neighbour connectivity. Formulate the logic to implement and find the connected components using 8-neighbour connectivity of pixels. The 8-neighbours of a pixel (x,y) is defined as:

```
        N8(x,y) = {(x-1,y),(x-1,y -1),(x,y -1),(x+1,y -1), (x
+1,y),(x +1,y +1),(x,y +1),(x-1,y +1)}

        Function    Count characters
        Input       Image
        Output      Number of character
```

In [43]:
```python
import numpy as np
import matplotlib.pyplot as plt


def get_neighbors(x, y, shape):
    """Get 8-neighbors for a given pixel (x, y) considering image boundaries."""
    neighbors = []
    for dx, dy in [(-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0), (1, 1), (0, 1), (-1
        nx, ny = x + dx, y + dy
        if 0 <= nx < shape[0] and 0 <= ny < shape[1]:
            neighbors.append((nx, ny))
    return neighbors


def connected_components(image):
    """Find connected components using 8-neighbor connectivity."""
    label = 1  # Start labeling from 1
    labels = np.zeros_like(image, dtype=int)
    component_sizes = []

    for x in range(image.shape[0]):
        for y in range(image.shape[1]):
            if image[x, y] == 1 and labels[x, y] == 0:
                # Start a new component
                stack = [(x, y)]
                component_size = 0
                while stack:
                    cx, cy = stack.pop()
                    if labels[cx, cy] == 0:
                        labels[cx, cy] = label
```

```
                        component_size += 1
                        for nx, ny in get_neighbors(cx, cy, image.shape):
                            if image[nx, ny] == 1 and labels[nx, ny] == 0:
                                stack.append((nx, ny))

                component_sizes.append(component_size)
                label += 1

    # Sorting component_sizes for plotting
    component_sizes.sort()

    # We are assuming that no of pixel representing the character is 2 times of  no
    for i in range(1,len(component_sizes)):
        if component_sizes[i]>2*component_sizes[i-1]:
            threshold = i

    No_of_char = len(component_sizes) - threshold


    return labels, No_of_char

def count_characters(image_name):
    # Load the image in grayscale
    img = plt.imread(image_name)
    img = (255* img).astype(int)



    # Binarize the image: assume characters are darker
    var_wb_lst = Between_class_variance1(img)
    threshold  = np.argmax(var_wb_lst[:])
    binary_img = (img < threshold).astype(int)

    plt.imshow(binary_img, cmap='gray')
    plt.show()

    # Find connected components, filter out small components
    labels, num_components = connected_components(binary_img)  # Adjust min_size as

    return num_components

# Path to the image
image_name = 'quote.png'

# Count the number of characters
number_of_characters = count_characters(image_name)
print(f"Number of characters (excluding punctuations): {number_of_characters}")
```

```
141
```




```
Number of characters (excluding punctuations): 64
```

Comment on Problem 4

The "count_characters" function uses the "Between_class_variance" function to binarize the image, which has intensity values ranging from 0 to 255. The "get_neighbors" function identifies the 8-connected neighbors of a given pixel "(x, y)" while considering the image boundaries to prevent out-of-bounds errors. The "connected_components" function begins by finding unvisited pixels that are part of a component (indicated by a value of 255). It then assigns a unique label to each component and tracks their sizes. After labeling, the function establishes a threshold to differentiate between characters and punctuation based on component sizes, assuming that the number of pixels representing a character is more than twice the size of the pixels representing punctuation.

In [ ]: