Ain Shams University Faculty of Engineering

Computer Engineering and Software Systems

CSE 483 Computer Vision Project

Submitted To: Prof. Mahmoud Khalil

Submitted On: 15/05/2024

QR Code Reader

Final Submission

The source code for this project is available on google colab at the following link:

https://github.com/Ahmed-T-Taha/QR-Code-Reader

Table of Contents

List of Figures

QR Code Processing and Decoding

Introduction

This project is concerned with applying image processing and noise attenuation techniques in order to find a QR code in an image, straighten it, remove the noise and ultimately decode it to extract the data.

The project is divided into three core sections: pre-processing the image, converting it into numeric cells, and decoding the cells.

In the pre-processing phase, we perform several core functions:

- Color correction and conversion to grayscale
- Removal of periodic noise
- Straightening of warped image
- Thresholding
- Morphological opening and closing

Meanwhile, the conversion phase can be broken down into:

- Removal of the quiet zone around the QR code
- Turning groups of pixels into a single binary cell to get an image of the exact size of the QR code (e.g. 21x21 for version 1 QR)
- Rotating or mirroring if necessary

Lastly, the decoding phase consists of:

- Extracting format information and performing BCH error correction on it
- Applying the mask pattern to the full QR code
- Extracting the data bits from the image
- Performing Reed-Solomon error correction on the data
- Decoding the string representing the final output of QR code

Main reference used for information and diagrams regarding decoding QR Codes:

https://www.thonky.com/qr-code-tutorial/

1- Packages Used

- cv2 (OpenCV) for image processing
- numpy for fast array processing
- matplotlib for displaying the images
- reedsolo for performing error correction on data modules of qr code
- galois for performing (bch) error correction on format information

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

!pip install --upgrade reedsolo
import reedsolo as rs
!pip install --upgrade galois
import galois
```

1 Imports

```
!git clone 'https://github.com/Ahmed-T-Taha/QR-Code-Reader'
test_cases = []
import os
test_cases_path = '/content/QR-Code-Reader/Test-Cases/'
for img_name in os.listdir(test_cases_path) :
  test_cases.append(cv2.imread(test_cases_path + img_name))
```

2 Importing and reading the test cases as cv2 images

2- Image Pre-Processing Functions

pre_process Function

```
def pre_process(img):
  # Remove colored pixels and convert to grayscale
  img_gray = color_correct(img)

  # Remove periodic noise
  img_noise_removed = remove_periodic_noise(img_gray)

  # Straighten image if warped, and return image inside bounding box
  img_straightened = straighten(img_noise_removed)

  # Resize image to standardize masks to be applied
  img_resized = cv2.resize(img_straightened, (1024, 1024))

  # Apply median blur, but only use it moving forward if it results in a major change
  img_median = cv2.medianBlur(img_resized, 21)
  if np.mean(abs(img_resized - img_median)) > 85:
    img_resized = img_median

  # Apply Gaussian Blur then Otsu's Thresholding to avoid hardcoding threshold value
  img_blur = cv2.GaussianBlur(img_resized, (5,5), 0)
  _, img_thresh = cv2.threshold(img_blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

  # Apply opening then closing
  square_se = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 21))
  img_dilated = ~cv2.dilate(~img_thresh, square_se)
  img_closed = ~cv2.erode(~img_dilated, square_se)
  img_eroded = ~cv2.erode(~img_closed, square_se)
  img_opened = ~cv2.dilate(~img_eroded, square_se)

  return img_opened
```

3 pre_process function

The pre-process function starts by calling several other functions which we will explore more in depth later:

- color_correct: removes colored pixels, converts format to grayscale, and if the image is color-inverted inverts it again to match the standard qr code format (black cells on white background)
- remove_periodic_noise: removes periodic noise generated from abnormally high values in band of frequencies in Fourier transform of image
- straighten: If the image is warped uses the warpAffine function to straighten it again and returns only section containing data with a small quiet zone around it.

After each of these functions is called, the image is then resized to be 1024x1024, which allows setting some values to be constant, such as kernel size of convolution functions.

Then, median blur is applied. We check to see that the average difference between what the value of a cell was and its new value. If this value is not very large, then we have most likely applied median blur to an image that does not have salt and pepper noise and can discard this result. However, if the change is large then there are extreme changes in values at nearly every cell which implies the need for the median blur. Therefore, in that case, we would keep the result.

Afterwards, we apply gaussian blur and Otsu's thresholding to avoid hard-coding of threshold values and get more accurate result. Lastly, we apply closing to fix small gaps in cells, followed by opening to remove noisy black pixels that would disrupt removal of quiet zone in decoding.

color_correct Function

```python
def color_correct(img):
  # Make all non-grayscale pixels white as they will not be part of the qr code
  for i, row in enumerate(img):
    for j, pixel in enumerate(row):
      pixel = [int(p) for p in pixel]
      if any((abs(pixel[0] - pixel[1]) > 10,
              abs(pixel[0] - pixel[2]) > 10,
              abs(pixel[1] - pixel[2]) > 10)):
        img[i][j] = (255, 255, 255)

  img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  _, img_thresh = cv2.threshold(img_gray, np.mean(img_gray), 255, cv2.THRESH_BINARY)

  # If more than 99% of border pixels are black, the image is most likely color-inverted
  black = 0
  borders = np.concatenate((img_thresh[0], img_thresh[-1],
                            cv2.transpose(img_thresh)[0], cv2.transpose(img_thresh)[-1]))
  for pixel in borders:
    if pixel == 0:
      black += 1

  if black > 0.99 * len(borders):
    img_gray = cv2.cvtColor(255 - img, cv2.COLOR_BGR2GRAY)

  #img_equalized = cv2.equalizeHist(img_gray)
  return img_gray
```

4 color_correct function

In the color_correct function, we start by iterating over each pixel in the image and checking its BGR tuple. If the difference between any 2 of the 3 values is sufficiently small, the pixel is assumed to be in grayscale (the difference is allowed to be larger than zero to allow for some possible color distortion in the image). If a cell is found not to be in grayscale, it is assumed not to be part of the QR code and replaced with a white pixel (255, 255, 255).

Afterwards, the image is turned to grayscale and temporary thresholding is applied, and the image is analyzed. If over 99% of the cells along the border of the image are found to be black, we can safely assume that the image is color-inverted; at which point we would invert each pixel of the image and return the final result of color_correct.

remove_periodic_noise Function

```python
def remove_periodic_noise(img):
  # get the frequency domain
  f = np.fft.fft2(img)
  fshift = np.fft.fftshift(f)

  # Remove the outlier frequencies
  fshift_abs = abs(fshift)
  fshift_med20 = 20 * cv2.medianBlur(fshift_abs.astype('float32'), 5)
  for i, row in enumerate(fshift_abs):
    for j, point in enumerate(row):
      if point > fshift_med20[i, j]:
        fshift[i, j] = fshift_med20[i, j] / 20

  # get the spatial domain back
  f_ishift = np.fft.ifftshift(fshift)
  img_back = np.fft.ifft2(f_ishift)
  img_back = np.real(img_back)
  img_back_normal = cv2.normalize(img_back, None, 255, 0, cv2.NORM_MINMAX, cv2.CV_8UC1)
  return img_back_normal
```

5 remove_periodic_noise function

To remove periodic noise, first we get the image in the frequency domain. Then we find the median magnitude of the frequencies in the 5x5 square around each frequency.

We multiply this median by 20 to get the threshold representing an outlier frequency at each point in the frequency domain.

We iterate through the frequency, replacing the outlier values by the median in their band.

Finally, we go back to the spatial domain, normalize the image and return.

straighten Function

```
def straighten(img):
  # Find the convex hull of the image
  # The operation is done on a color-inverted version
  _, img_thresh = cv2.threshold(img, np.mean(img), 255, cv2.THRESH_BINARY)
  img_reverse = 255 - img_thresh
  contours, _ = cv2.findContours(img_reverse, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
  cnts = np.concatenate(contours)
  hull = cv2.convexHull(cnts)

  # Approximate the convex hull to a polygon
  def closest_point (points_list, p):
    return min(points_list, key=lambda x: ((x[0]-p[0])**2 + (x[1]-p[1])**2))
  corners = cv2.approxPolyDP(hull, 0.02* cv2.arcLength(hull, True), True)
  corners = [p[0] for p in corners]


  # Find the points in the convex hull closest to each of the 4 corners
  rows, cols = img_thresh.shape
  top_left = closest_point(corners, [0, 0])
  top_right = closest_point(corners, [cols - 1, 0])
  bottom_left = closest_point(corners, [0, rows - 1])
  bottom_right = closest_point(corners, [cols - 1, rows - 1])

  # Find vectors of each of the 4 lines
  top_line = top_left - top_right
  left_line = top_left - bottom_left
  bottom_line = bottom_right - bottom_left
  right_line = bottom_right - top_right
```

6 straighten function 1

In the straighten function, we start by inverting the image color to find the contours and thus the convex hull around all the black pixels representing QR code foreground.

We use cv2.approxPolyDP to try to get an approximate square around the QR from the convex hull. And use the closest_point function to order each of the 4 corners of the shape according to which is closest to each corner of the image. Then we generate vectors representing each of the 4 lines connecting these corners.

```python
# If the angle is sufficiently close to 90, warpAffine is not necessary
if any((abs(find_angle(top_line, left_line) - 90) < 5,
       abs(find_angle(top_line, right_line) - 90) < 5,
       abs(find_angle(bottom_line, left_line) - 90) < 5,
       abs(find_angle(bottom_line, right_line) - 90) < 5)):
  # Here we will get the bounding box for the image
  # We try to pad with 10 pixels if possible to allow quiet zone
  x,y,w,h = cv2.boundingRect(hull)
  left_bound = max(0, x-10)
  right_bound = min(cols - 1, x+h+10)
  top_bound = max(0, y-10)
  bottom_bound = min(rows - 1, y+h+10)
  return img[top_bound : bottom_bound , left_bound : right_bound]


# If we reached this point, then image needs to be warped
# Perform affine warp using the points we got to straighten the image
srcTri = np.array([top_left, top_right, bottom_right]).astype(np.float32)
dstTri = np.array([[10, 10], [cols - 11, 10], [cols - 11, rows - 11]]).astype(np.float32)
warp_mat = cv2.getAffineTransform(srcTri, dstTri)
img_warp = cv2.warpAffine(img, warp_mat, (cols, rows), borderMode=cv2.BORDER_REPLICATE)
return img_warp
```

7 straighten function 2

After we have found the 4 corners, and the 4 lines connecting them, we find the angle made at each of the 4 corners. If any of the angles is sufficiently close to 90, then the image might need rotation, but it is not warped and does not need the warpAffine function to be applied. Here we use the cv2.boundingRect function to find the bounds of the image and add padding of 10 pixels. We then return this image which has a 10-pixel quiet zone to compensate for inaccuracies in the convex hull.

However, if none of the corners form a 90-degree angle, we use the cv2.getAffineTransform on the three points representing the top right corner of the QR code (as even if the image is mirrored or rotated 90 degrees, there will still be a locator box there) and apply the cv2.warpAffine function to get the straightened QR code.

3- Image To QR Cells Conversion Functions

remove_quiet_zone Function

```
def remove_quiet_zone (img) :
  # Find limits of QR code
  start_row = -1
  start_col = -1
  end_row = -1
  end_col = -1

  for row_index, row in enumerate(img):
    for pixel in row:
      if pixel != 255:
        start_row = row_index
        break
    if start_row != -1: break

  for row_index, row in enumerate(img[::-1]):
    for pixel in row:
      if pixel != 255:
        end_row = img.shape[0] - row_index
        break
    if end_row != -1: break
```

8 remove_quiet_zone function 1

```
for col_index, col in enumerate(cv2.transpose(img)):
  for pixel in col:
    if pixel != 255:
      start_col = col_index
      break
  if start_col != -1: break

for col_index, col in enumerate(cv2.transpose(img)[::-1]):
  for pixel in col:
    if pixel != 255:
      end_col = img.shape[1] - col_index
      break
  if end_col != -1: break


return img[start_row:end_row, start_col:end_col]
```

9 remove_quiet_zone function 2

To remove quiet zone, we iterate through the rows and the first black pixel we meet (which would not be noise due to the opening operation) we set that as the first row. We perform a similar operation while traversing from the last row, and repeat the process for columns.

Finally, we return only the rows and columns considered as not part of the quiet zone.

find_locator_boxes Function

```
locator_box = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 0],
    [0, 1, 0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0, 1, 0],
    [0, 1, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0]]

def find_locator_boxes (qr_cells):
  # As we are dealing with binary data:
  # Taking the absolute of the element-wise difference
  # Then summing up all cells in the result tells us how many cells are different
  # We allow for 1 bit to be off from a perfect locator box
  locator_found = {
      'top_left': np.sum(np.abs(qr_cells[:7, :7] - locator_box)) < 2,
      'top_right': np.sum(np.abs(qr_cells[-7:, :7] - locator_box)) < 2,
      'bottom_left': np.sum(np.abs(qr_cells[:7, -7:] - locator_box)) < 2,
      'bottom_right': np.sum(np.abs(qr_cells[-7:, -7:] - locator_box)) < 2
  }

  return locator_found
```

10 find_locator_boxes helper function

get_qr_numeric Function

```
def get_qr_numeric(qr_no_quiet_zone):
  # Attempts to break down the qr code into versions 1, 2, 3, 4 (21, 25, 29, 33 cells)
  # If none of them generate containing a locator box, returns -1
  for grid_cells_num in range(21, 34, 4):
    grid_cell_size = int( max(
          np.ceil(qr_no_quiet_zone.shape[0]/grid_cells_num),
          np.ceil(qr_no_quiet_zone.shape[1]/grid_cells_num)) )

    qr_dim = grid_cells_num * grid_cell_size
    qr_no_quiet_zone = cv2.resize(qr_no_quiet_zone, (qr_dim, qr_dim))

    # Transform to grid of cells
    qr_cells = qr_no_quiet_zone.reshape((
        grid_cells_num,
        grid_cell_size,
        grid_cells_num,
        grid_cell_size,
    )).swapaxes(1, 2)

    # Get numeric values by median in area
    qr_cells_numeric = np.ndarray((grid_cells_num, grid_cells_num), dtype=np.uint8)
    for i, row in enumerate(qr_cells):
        for j, cell in enumerate(row):
            qr_cells_numeric[i, j] = (np.median(cell) // 255)

    # Checks if a locator box was found in any corner
    # If none are found, check the next version
    if any(find_locator_boxes(qr_cells_numeric).values()):
      return qr_cells_numeric

  # If this point is reached, none of the versions contained a locator box
  # The qr code could not be processed correctly
  return -1
```

11 get_qr_numeric function

This function tries to break down the qr code into 21x21 cells then 25, 29 and 33 (corresponding to versions 1-4 QR codes respectively)

First, the cell size is taken to be the larger value between number of rows or columns divided by number of cells. Then, the image is resized to be exactly equal to (cell size * cell num) x (cell size * cell num) pixels. This ensures that the QR code is exactly square and that the array can be reshaped successfully.

Now we have an array of size cell num x cell num. Where each element in this 2d array is another 2d array containing all the pixels representing a specific cell. By taking the median of each of these arrays, we arrive at a binary interpretation of the QR code.

Afterwards, we check to see if any locator boxes can be found. If no locator boxes are found, we move on the next version and try again. If no locator boxes are ever found, then either no QR code is present, or pre-processing was insufficient for decoding.

rotate_mirror Function

```
def rotate_mirror (qr_cells):
  # Find which corners contain locator boxes
  locator_found = find_locator_boxes(qr_cells)

  # If a locator box is in the bottom right, an action must be taken
  if(locator_found['bottom_right']):
    if(locator_found['bottom_left'] and locator_found['top_right']):
      # Rotate the image 180 degrees
      qr_cells = np.rot90(qr_cells, 2)
    elif(locator_found['bottom_left'] and locator_found['top_left']):
      # Mirror the image horizontally
      qr_cells = np.fliplr(qr_cells)
    elif(locator_found['top_right'] and locator_found['top_left']):
      # Mirror the image vertically
      qr_cells = np.flipud(qr_cells)

  return qr_cells
```

12 rotate_mirror function

This function aims to fix a QR code that has been mirrored or rotated 180 degrees.

It works by checking where locator boxes are found and depending on where they were found applying the appropriate function:

- np.rot90 twice to rotate 180 degrees if locator boxes were found in the 2 bottom corners and top right.
- np.fliplr if locator boxes were found in the 2 bottom corners and top left.
- np.flipud if locator boxes were found in the 2 top corners and bottom right.

4- Decoding Functions

decode Function

```
def decode (img_init):
  # Plot each stage of qr code processing
  fig, (plt1, plt2, plt3) = plt.subplots(1, 3)
  fig.set_figwidth(15)
  for ax in fig.axes:
    ax.set_xticks([])
    ax.set_yticks([])

  # Read the raw image
  plt1.imshow(img_init)
  plt1.set_title('Initial Image')

  # Pre-process the image
  img_processed = pre_process(img_init)
  plt2.imshow(img_processed, cmap='gray')
  plt2.set_title('Image after pre-processing')

  # Remove quiet zone and rotate/flip if necessary
  qr_no_quiet_zone = remove_quiet_zone(img_processed)
  qr_cells_numeric = get_qr_numeric(qr_no_quiet_zone)
  if type(qr_cells_numeric) == int:
    return 'Could not process QR Code'
  qr_cells = rotate_mirror(qr_cells_numeric)

  # Show final output
  plt3.imshow(qr_cells, cmap='gray')
```

13 decode function 1

decode is the main function of the program. After reading the image using cv2.imread it is immediately passed to the decode function which calls all other functions including the pre-processing ones. Firstly, we generate a matplotlib figure containing 3 subplots. The first for the initial image, the second for the image after pre-processing, and the last for the final QR code which will be decoded.

Initially, we call the pre_process function, then remove_quiet_zone, and finally rotate_mirror. At this point, we have the QR code in the format of cells ready to be decoded.

```
# Get the format info
ecl_char, mask_str = get_format_info(qr_cells)
if mask_str == -1:
  return 'Could not correct format info'

# Find the version number (version 1 has 21 cells per row/column,
# and each successive version increases the cells by 4)
cells_num = qr_cells.shape[0]
ver = int((cells_num - 17) / 4)
ver_info = str(ver) + '-' + ecl_char

# Apply the mask to the whole qr code
qr_cells = apply_mask(qr_cells, mask_str)

# Get the bytes representing all data and ecc in the qr code
message_bytes = extract_data(qr_cells, ver)

# Correct all errors and show output
ans = correct_errors(message_bytes, ver_info)

plt3.set_title(f'Output: {ans}')
return ans
```

14 decode function 2

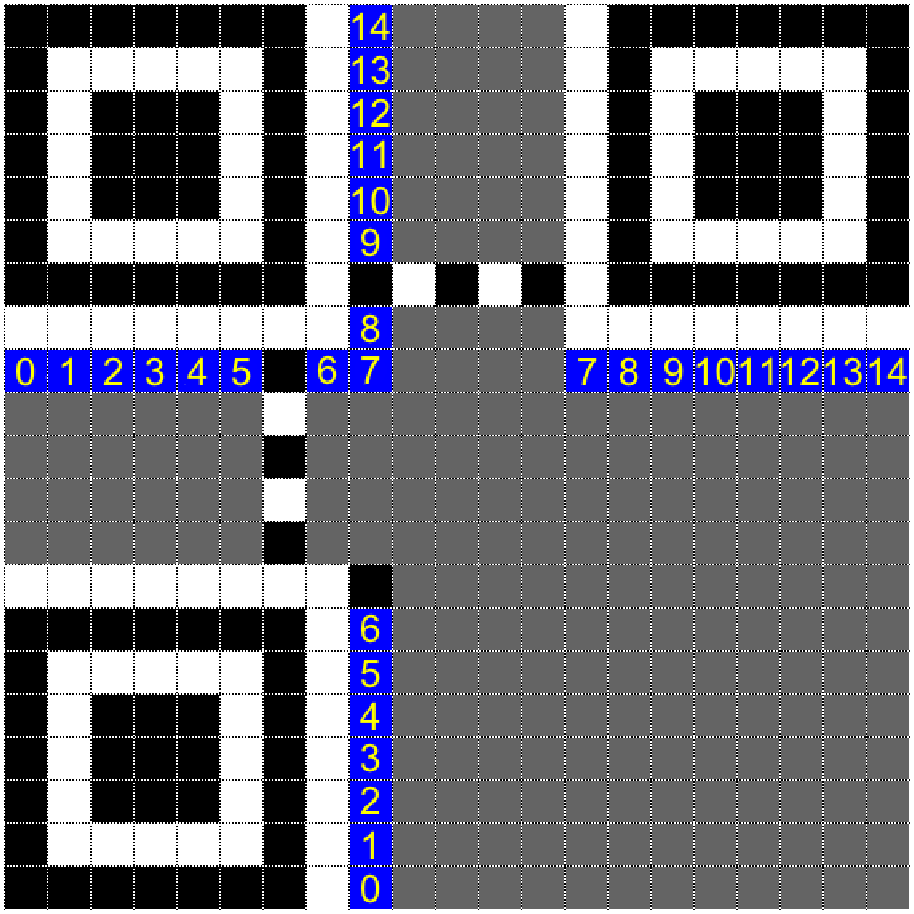
Now with the QR cells, we will call several other functions which we will explore more in depth later.

Firstly, we use get_format_info to find the error correction level (ecl_char which is L, M, Q, or H) and the 3-bit mask (represented as a string: mask_str)

Then, we find the version number. A version 1 QR code has 21 cells per row/column and each successive version has 4 more cells per row/column.

Afterwards, we apply the appropriate mask to all cells in the QR code using apply_mask, then get all the data/error correction bits in order using extract_data. Finally, we use the version information to apply reed-solomon error correction and get final output.

get_format_info Function



15 format information layout

```python
def get_format_info(qr_cells):
  # We will try to find the error-corrected format info from 2 sources
  # If both fail, we will return -1
  for try_count in range(2):
    # Find format info with error correction bits and apply format mask
    format = []
    if try_count == 0:
      format.extend(qr_cells[8, 0:6])
      format.extend(qr_cells[8, 7:9])
      format.append(qr_cells[7, 8])
      format.extend(reversed(qr_cells[0:6, 8]))
    elif try_count == 1:
      format.extend(reversed(qr_cells[-7: , 8]))
      format.extend(qr_cells[8, -8:])

    # Format Error Correction Mask
    format_mask = [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0]
    format = [a^b for a, b in zip(format, format_mask)]
```

16 get_format_info 1

The function is contained in a for loop. It performs all of the following steps twice; once for each of the 2 sets of format information. However, if error correction on the first set succeeds, it will not attempt the second time

It starts by getting the 5 format information bits (2 error correction level and 3 mask pattern) and the 10 error correction bits (as shown in figure 15). Then the 15 bits are XOR'ed with the standard mask to get the bits we can perform error correction on.

```
  # Use the galois library to perform error correction on both sets of format info
  # When the number of errors is returned as -1, the codeword could not be corrected, try the other one
  bch = galois.BCH(15, 5)
  corrected_format, err_no = bch.decode(format, errors=True)

  # If the number of corrected errors is -1, correcting the data has failed
  # If this happens, attempt the other set of format info
  if err_no != -1:
    mask_bits = corrected_format[2:5]
    mask_bits = [a^b for a, b in zip(mask_bits, [1, 0, 1])] # XOR with format mask
    mask_bits = [int(not(c)) for c in mask_bits] # Invert bits as black represents 0
    mask_str = ''.join([str(c) for c in mask_bits]) # Convert to string

    ecl_bits = corrected_format[:2]
    ecl_bits = [a^b for a, b in zip(ecl_bits, [1, 0])] # XOR with format mask
    ecl_bits = [int(not(c)) for c in ecl_bits] # Invert bits as black represents 0

    # L = 11, M = 10, Q = 01, H = 00
    ECL_TABLE = [['H', 'Q'], ['M', 'L']]
    ecl_char = ECL_TABLE[ecl_bits[0]][ecl_bits[1]]

    return  ecl_char, mask_str

  # If this point is reached, error correcting the format info has failed
  try_count +=1

# If we reach this point, we have failed to correct the format info in both cases
# We return 2 values as that is what is unpacked at return
return -1, -1
```

17 get_format_info 2

Next, it performs error correction on the bits using the BCH class of the galois library. If the number of corrected errors = -1, the library has failed to correct the bits and we will continue the loop. If this occurs with both sets of format information, the format info is too damaged and the QR code could not be decoded. Otherwise, we proceed with extracting the format information.

For both the ecl bits and the mask bits, we xor with the format mask again and invert the bits to have the data that matches the appropriate lookup tables for mask patterns and error correction level.

Finally, we return the character representing ecl and the string representing mask pattern.

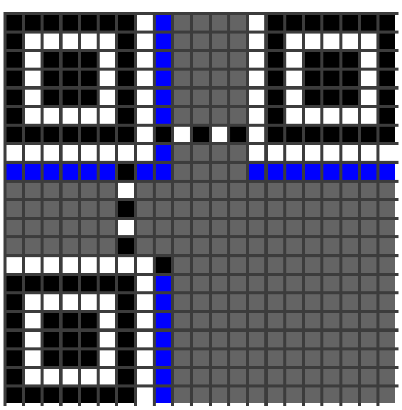
apply_mask Function

```python
# Masks applied to whole qr code
MASKS = {
    "000": lambda i, j: (i * j) % 2 + (i * j) % 3 == 0,
    "001": lambda i, j: (i / 2 + j / 3) % 2 == 0,
    "010": lambda i, j: ((i * j) % 3 + i + j) % 2 == 0,
    "011": lambda i, j: ((i * j) % 3 + i * j) % 2 == 0,
    "100": lambda i, j: i % 2 == 0,
    "101": lambda i, j: (i + j) % 2 == 0,
    "110": lambda i, j: (i + j) % 3 == 0,
    "111": lambda i, j: j % 3 == 0,
    }

def apply_mask(qr_cells_numeric, mask_str):
  # Applies the mask to all cells in the qr code
  for i, row in enumerate(qr_cells_numeric):
    for j, cell in enumerate(row):
      qr_cells_numeric[i, j] = int(cell if MASKS[mask_str](i, j) else not cell)
  return qr_cells_numeric
```

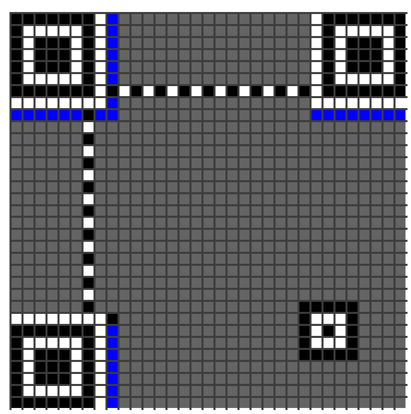
18 apply_mask function

This function applies the appropriate mask pattern to all cells in the QR code. Technically, we should not apply the mask to certain reserved areas like format information, timing patterns, alignment patterns, locator boxes and the dark module. However, all of these will not be accessed again for the remaining duration of the program, so modifying them will have no negative effects.

extract_data Function



19 reserved areas for version 1



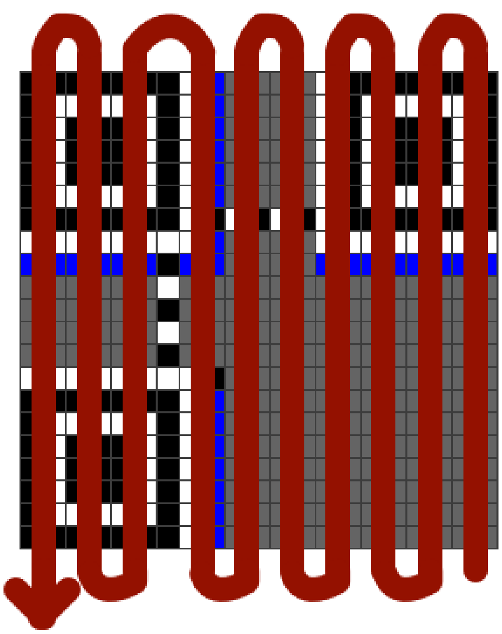
20 reserved areas for version 4

All the grey bits in figures (19 and 20) are the bits containing data. Everything else are reserved bits that contain format information or alignment/timing patterns which are irrelevant to the data we are attempting to decode. Note that the alignment pattern in the bottom right corner of figure 20 is present in version 2-6 QR codes.

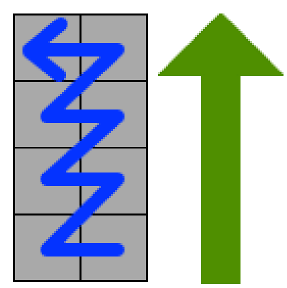
```python
def extract_data(qr_cells, version):

  # Creating a 2d array where 1 represents a reserved area where no data will be found
  qr_len = qr_cells.shape[0]
  qr_reserved_areas = np.zeros((qr_len, qr_len))

  # Area reserved for top left locator box and format info
  qr_reserved_areas[:9, :9] = np.ones((9, 9))

  # Area reserved for top right locator box and format info
  qr_reserved_areas[:9, -8:] = np.ones((9, 8))

  # Area reserved for bottom left locator box and format info
  qr_reserved_areas[-8:, :9] = np.ones((8, 9))

  # Area reserved for top timing diagram
  qr_reserved_areas[6, :] = np.ones(qr_len)

  # Area reserved for left timing diagram
  qr_reserved_areas[:, 6] = np.ones(qr_len)

  # If version > 1 there are more reserved areas
  if version > 1:
    # Add area reserved for alignment pattern
    qr_reserved_areas[-9:-4, -9:-4] = np.ones((5, 5))

    #Add area reserved for 7 remainder bits
    qr_reserved_areas[-11:-8, :2] = np.ones((3, 2))
    qr_reserved_areas[-12 , 0] = 1
```

21 extract_data function 1

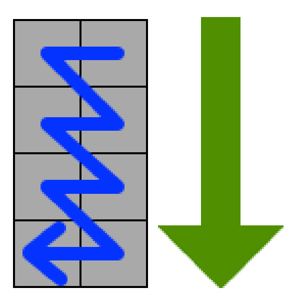
Firstly, we create a 2d array of zeros where 0 means a bit contains data, while 1 means it is a reserved bit. We set reserved areas to 1 in the pattern that was discussed in the previous section. In addition to the 7 remainder bits that are reserved in some QR codes of version > 1.



22 data bit progression



23 upward bit progression



24 downward bit progression

```python
# Iterate through all the bits in the pattern explained in:
# https://www.thonky.com/qr-code-tutorial/module-placement-matrix#pattern-of-placement
message_bits = []
row, col = qr_len - 1, qr_len - 1

while(col > 0):
  # When we reach the timing pattern, start after it
  if col == 6:
    row, col = 9, 5
    continue
  # Start the new column where the previous one ended
  if row > qr_len - 1:
    row = qr_len - 1
  elif row < 0:
    row = 0
  # Iterate through 2 consecutive columns, appending bits in order
  # Making sure not to append bits included in reserved areas
  while(0 <= row <= qr_len - 1):
    if (not qr_reserved_areas[row, col]):
      message_bits.append(qr_cells[row, col])
    if (not qr_reserved_areas[row, col - 1]):
      message_bits.append(qr_cells[row, col - 1])

    # In these columns we iterate upwards
    if any((col == 3 , col % 4 == 0)):
      row -= 1
    # All other columns iterate downwards
    else:
      row += 1
  # Iterate over 2 columns at a time
  col -= 2

# Turn the array of bits to an array of bytes
message_bytes = [int("".join(map(str, message_bits[i:i+8])), 2) for i in range(0, len(message_bits), 8)]
return message_bytes
```

25 extract_data function 2

In this section, we iterate over all of the cells in the QR code following the patterns illustrated in figures 22-24.

The data bits are placed starting at the bottom-right of the matrix and proceeding upward in a column that is 2 modules wide. When the column reaches the top, the next 2-module column starts immediately to the left of the previous column and continues downward. Whenever the current column reaches the edge of the matrix, move on to the next 2-module column and change direction. If a function pattern or reserved area is encountered, the data bit is placed in the next unused module.

Figure 22 shows the pattern of placing the data bits in the QR code. Notice that when the vertical timing pattern is reached, the next column starts to the left of it.

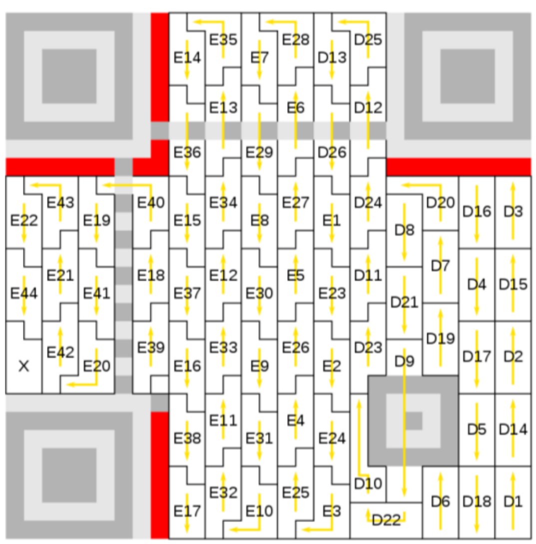
Finally, we take the data and convert each 8 bits to a byte for easier error correction using the reedsolo library and return.

correct_errors Function

```python
def correct_errors (message_bytes, version):
  # This table shows:
  # First: Number of blocks that codewords are divided into
  # Second: Number of data codewords per block
  # Third: Number of error correction codewords per block
  VERSIONS = {
      '1-L': (1, 19, 7),
      '3-Q': (2, 17, 18),
      '4-L': (1, 80, 20),
  }

  if version not in VERSIONS:
    return 'No support for this version\nwith this error correction level yet'

  blocks_num, data_num, ecc_num = VERSIONS[version]
  try:
    message_decoded = []
    rsc = rs.RSCodec(nsym=ecc_num)
    # The data and ecc for each block are interleaved
    # so each block's data shows up at location if bit_loc mod blocks_num = block
    for block in range(blocks_num):
      block_bytes = [b for i, b in enumerate(message_bytes) if i % blocks_num == block]
      block_decoded = rsc.decode(block_bytes)[0]
      message_decoded.extend(block_decoded)

    # We use zfill to restore leading zeros and remove the last 4 bits (null)
    # Expected number of bits is extracted from number of data bytes * 8 - 4
    # Where number of data bytes is number of blocks * number of data bytes per block
    data_bits = bin(int.from_bytes(message_decoded, byteorder='big'))[2:-4].zfill(blocks_num * data_num * 8 - 4)
```

26 correct_errors function 1

The VERSIONS dictionary contains the information relevant to the 3 QR code formats that show up in the test cases, but can easily be extended to include more based on the table available at this link: https://www.thonky.com/qr-code-tutorial/error-correction-table



27 Interleaved data bytes

In this section of the function, we take each section of data and error correction bytes and use the reedsolo library to perform error correction on them. Taking note of using the correct number of error correction bytes as given by the format information and the interleaving of data bytes as illustrated in figure 27.

If when using reedsolo decode function a ReedSolomonError was raised, then the data had too many errors and could not be corrected.

We pad the resulting data codewords with leading zeros to restore leading zeros in encoding mode and convert to bits to allow decoding of data depending on encoding mode and length bits.

```
  # Check the encoding mode
  if (int(data_bits[:4], 2) == 4): # Byte encoding = 0100 = 4
    # Use the error-corrected length, not the original
    len_int = int(data_bits[4:12], 2)
    data_bits = data_bits[12:]
    data_bytes = int(data_bits, 2).to_bytes((len(data_bits)+7)//8, 'big')
    data_decoded = data_bytes[:len_int].decode(encoding="utf-8")
    return data_decoded


  elif (int(data_bits[:4], 2) == 2): # Alphanumeric Encoding = 0010 = 2
    len_int = int(data_bits[4:13], 2) # Alphanumeric Encoding has 9 bit length
    data_bits = data_bits[13:]

    # Each 11 bits encode 2 characters
    data_decoded = ''
    charPairs = [int(data_bits[i : i+11], 2) for i in range(0, len(data_bits), 11)]
    for pair in charPairs:
      # According to this guide: https://www.thonky.com/qr-code-tutorial/alphanumeric-mode-encoding
      # Alphanumeric encoding gives each character its value based on its index in this lookup string
      alphanumeric_lookup = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ $%*+-./:'
      # The first character is represented by the quotient of the 11-bit value divided 45
      # The second character is the remainder
      data_decoded += alphanumeric_lookup[pair // 45]
      data_decoded += alphanumeric_lookup[pair % 45]
    return data_decoded[:len_int]

  else: return 'No matching encoding mode found'

# This exception occurs when there are too many errors to correct
except rs.ReedSolomonError:
  return 'Could not correct data'
```

28 correct_errors function 2

In this section of the function, we take the first 4 bits to check the encoding mode. Then the next 8 bits show the length of the message (9 in the case of alphanumeric encoding).

Then we apply the appropriate function to decode the final message bits to a string representing the final result of the QR code which we return.

5- Passed Test Cases

```
decode(test_cases[0])
'01-Good job!'
```

Initial Image | Image after pre-processing | Output: 01-Good job!

```
decode(test_cases[8])
'LeffyBinaYaDonya'
```

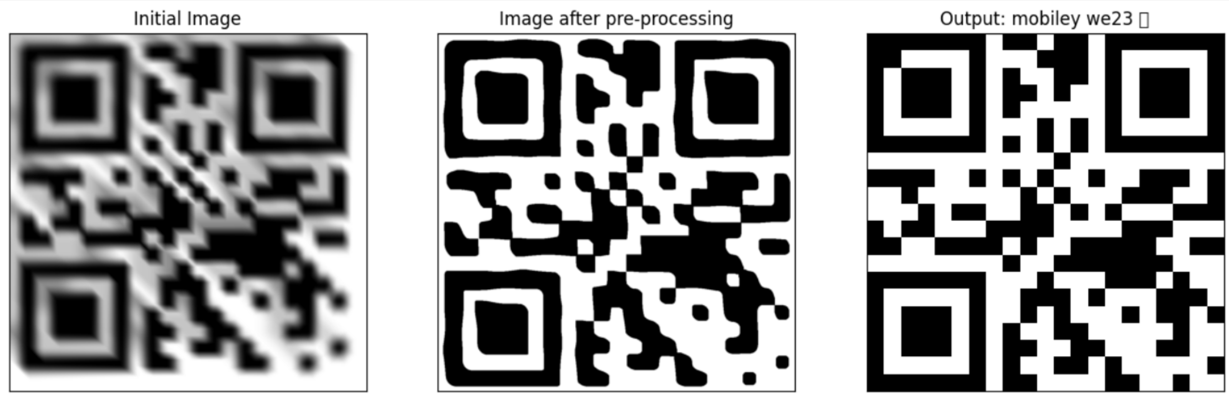
Initial Image | Image after pre-processing | Output: LeffyBinaYaDonya

```
decode(test_cases[4])
'Black mirror hehe'
```

Initial Image | Image after pre-processing | Output: Black mirror hehe

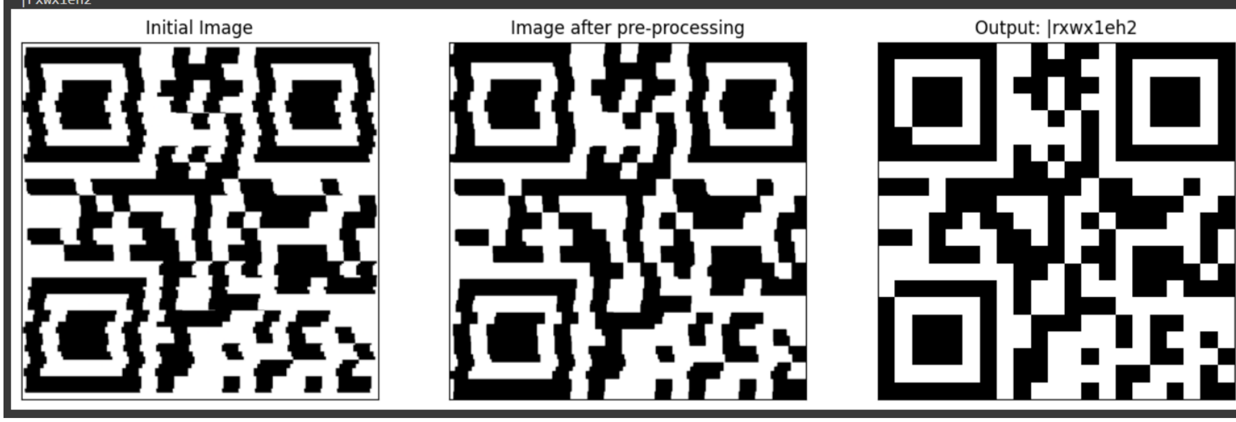
```
decode(test_cases[5])
```

```
'mobiley we23 💀'
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyph 128128 (\N{SKULL}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128128 (\N{SKULL}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```



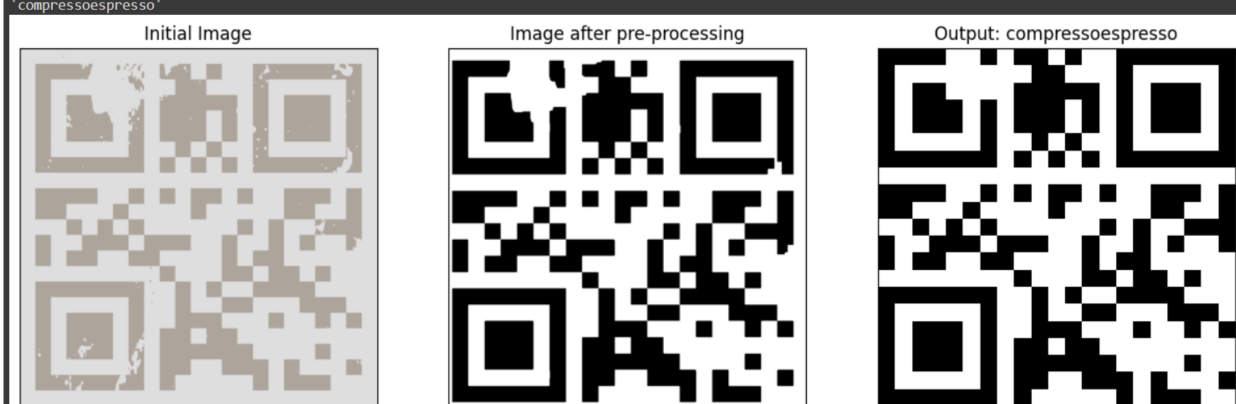

```
decode(test_cases[7])
```

```
'|rxwx1eh2'
```



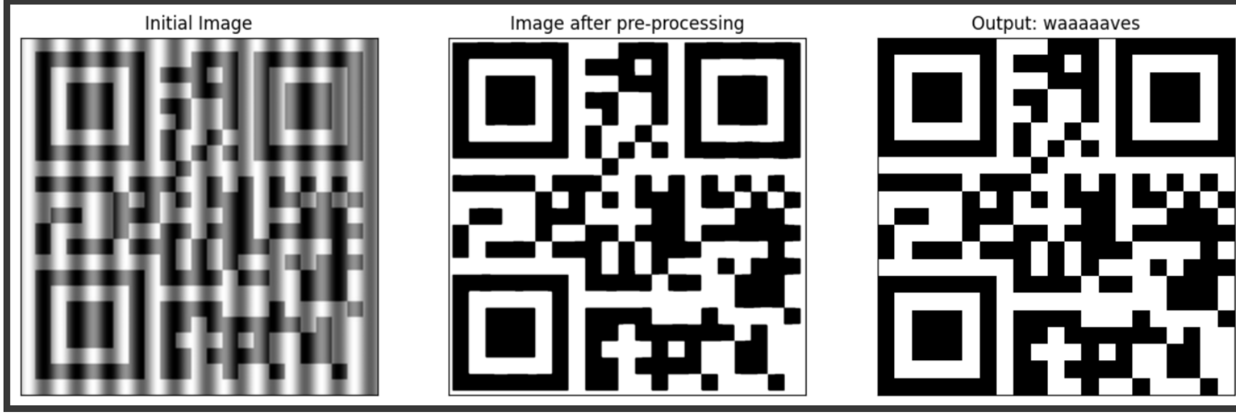

```
decode(test_cases[9])
```

```
'compressoespresso'
```



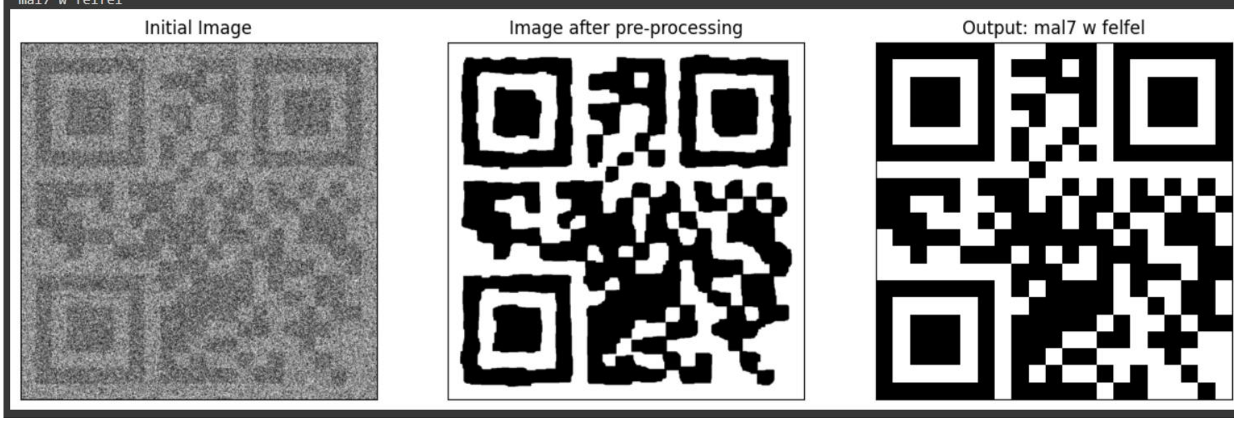
```
decode(test_cases[10])
```

```
'waaaaaves'
```



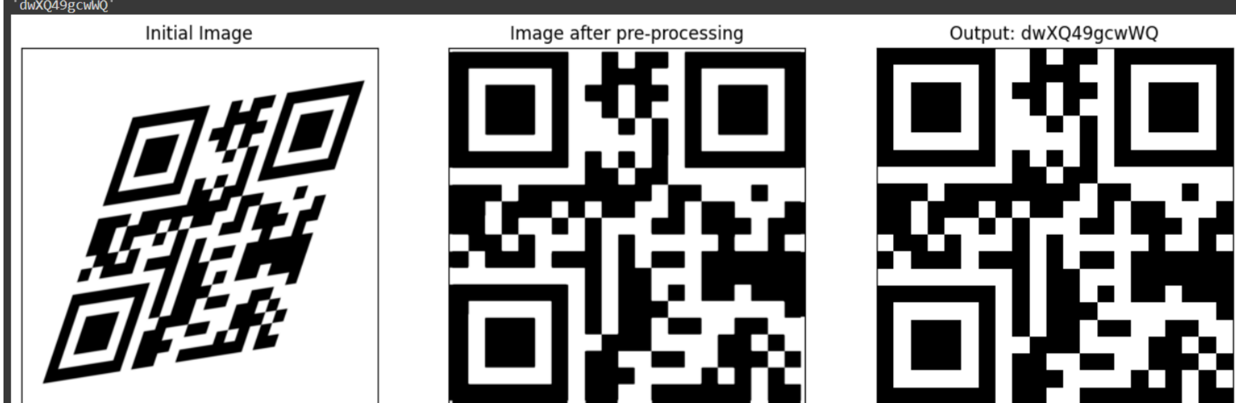

```
decode(test_cases[6])
```

```
'mal7 w felfel'
```

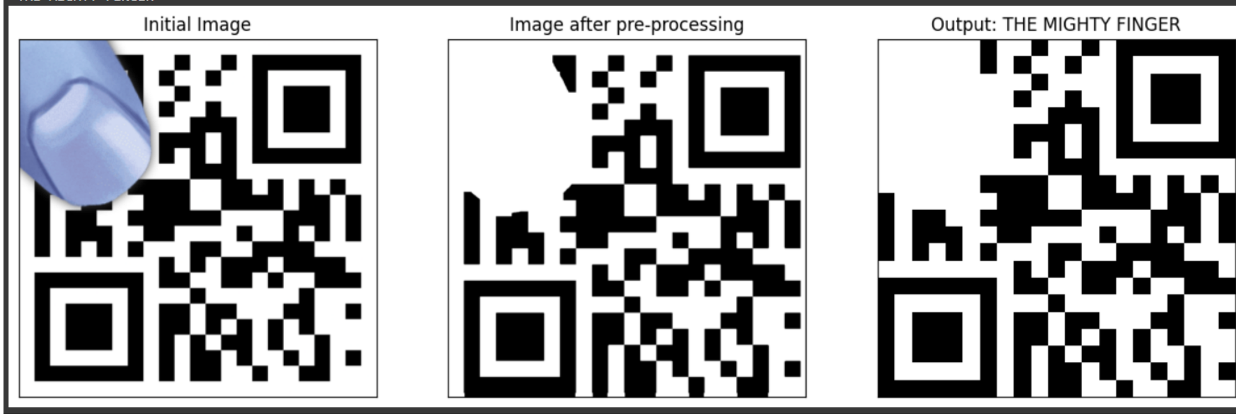



```
decode(test_cases[3])
```

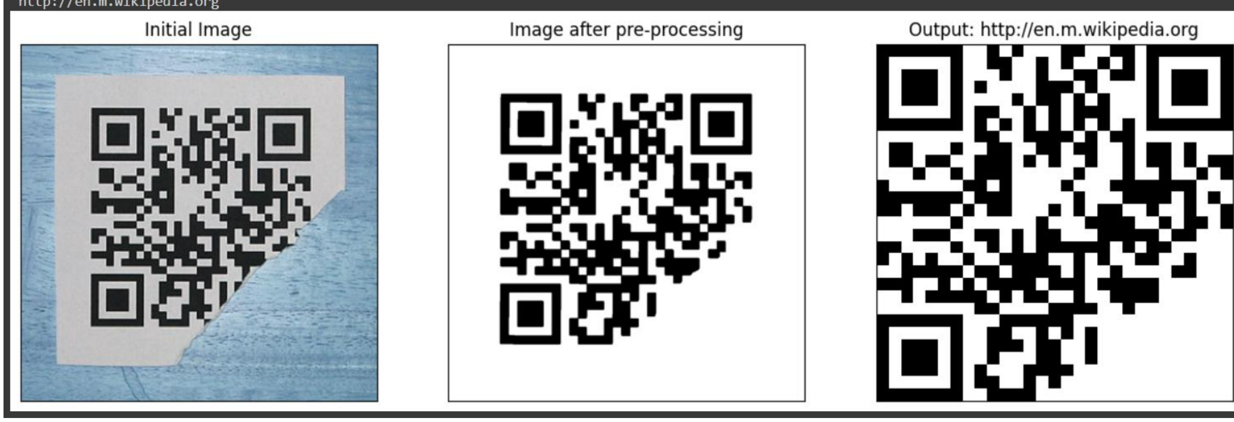
```
'dwXQ49gcwWQ'
```



```
decode(test_cases[1])
'THE MIGHTY FINGER'
```




```
decode(test_cases[11])
'http://en.m.wikipedia.org'
```




```
decode(test_cases[2])
'Wow, you did it! Here's your reward: https://youtu.be/dQw4w9WgXcQ'
```