IMAGE ENCRYPTION AND DECRYPTION

Every digital image is composed of tiny units called *pixels*. Each pixel represents a color, and the entire image is essentially a grid of these pixels. Each pixel's color is typically represented by a certain number of bits, known as the bit depth. Common bit depths include:

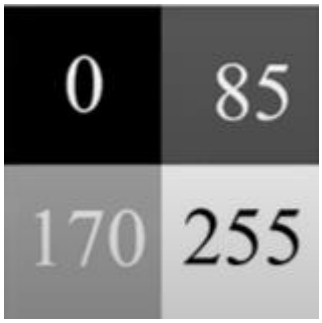
- **8-bit:** 256 possible colors.
- **24-bit** (True Color): 16.7 million colors, where each color is represented by three 8-bit values (one for each of the Red, Green, and Blue channels).

How Images are Stored :

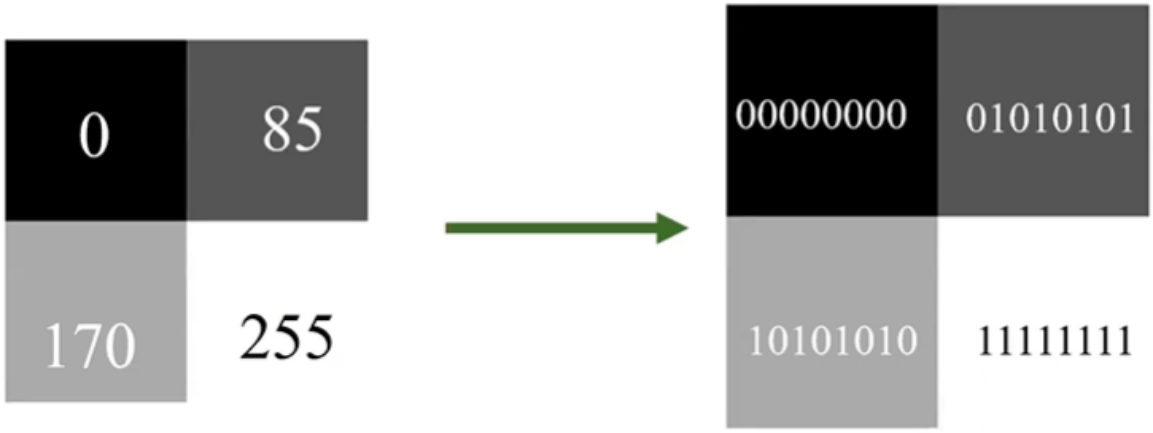
1. **Pixel Matrix:** The image is stored as a 2D matrix where each element of the matrix corresponds to a pixel in the image. The matrix dimensions are defined by the width and height of the image.
2. **Color Channels:** In a color image, each pixel is usually broken down into three color channels: red, green, and blue. Each channel is represented by 8 bits, giving a range of values from 0 to 255 for each color.
3. **Bit Depth:** The term "bit depth" refers to the number of bits used to represent the color of a single pixel. For RGB images, the bit depth is 24 bits (8 bits per channel).

Example :

1. **Black and White (Grayscale) Image:** In a grayscale image, each pixel is represented by a single 8-bit value ranging from 0 (black) to 255 (white).



0 corresponds to white, 255 to white, and in between are the shades



2. **Color Image:**

For an RGB image, a pixel might be represented as:

Red: 128 (binary: 10000000)

Green: 64 (binary: 01000000)

Blue: 255 (binary: 11111111)

The pixel data is stored as a combination of these values in the matrix.

Practical Implications :

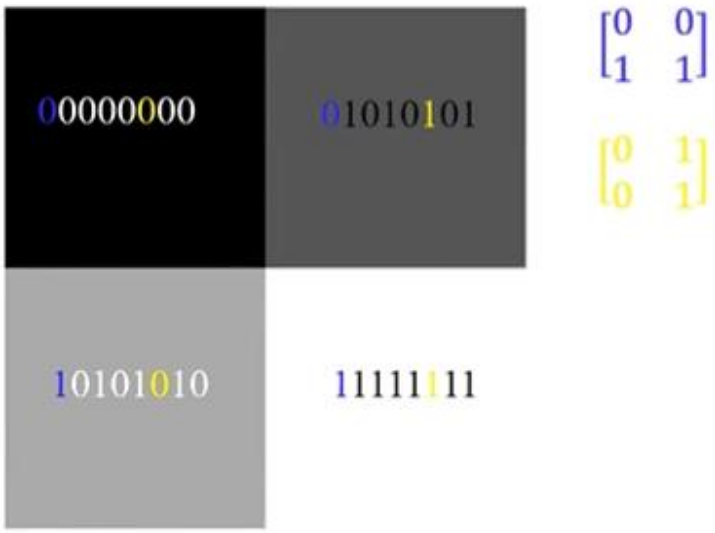
1. **Image Processing**: By manipulating the values in the pixel matrix, you can perform various image processing tasks, such as encryption, filtering, and transformation.

2. **Compression:** Images can be compressed to reduce file size by encoding the pixel data more efficiently, while still preserving visual quality.

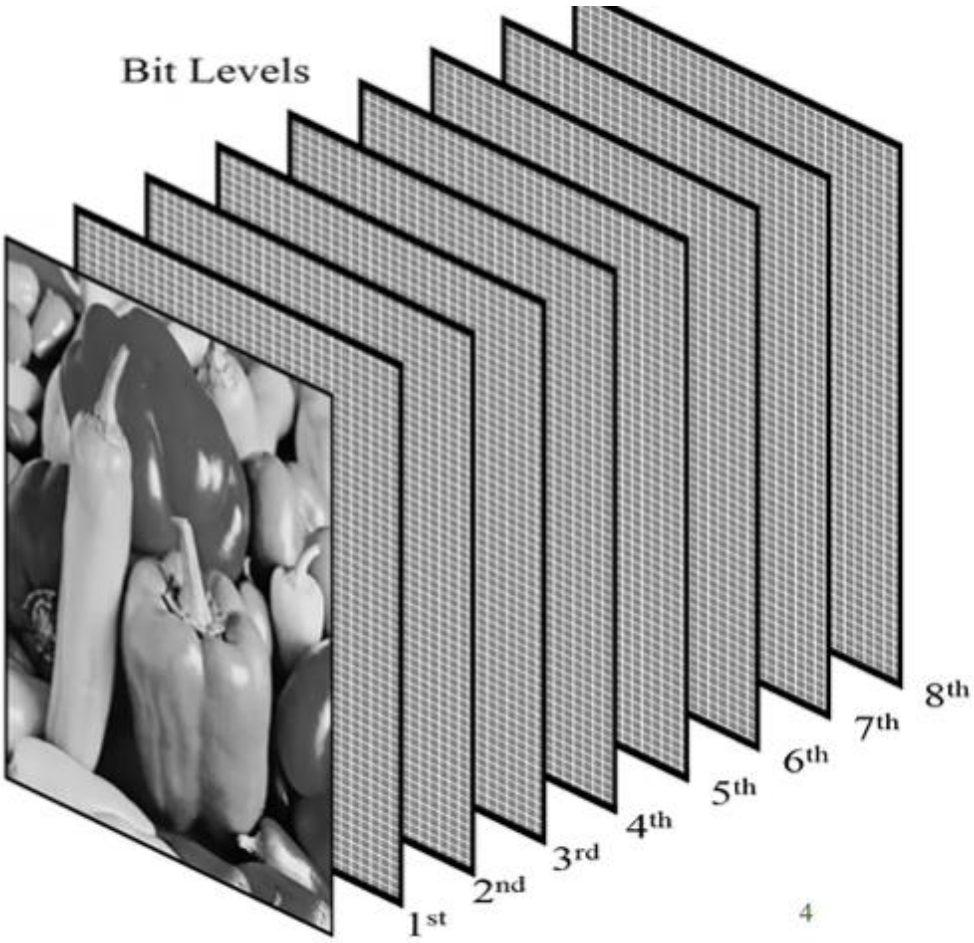
Image representation :

Each of those bits for each pixel, from 1st to 8th can be gathered into a single binary matrix.

These correspond to the bit levels of the image



Bit Planes :



1. Bit Plane 0:

- **Represents:** The least significant bit (LSB) of each pixel's binary value.
- **Value Contribution:** This bit contributes the smallest amount to the pixel value. In binary terms, this bit has a place value of 20=12^0 = 120=1.
- **Effect on Image:** Bit Plane 0 is often very sparse, with many pixels having either 0 or 255 (binary 0 or 1). It has the least impact on the overall image detail but contributes to subtle variations.

2. Bit Plane 1:

- **Represents:** The second least significant bit of each pixel's binary value.
- **Value Contribution:** This bit has a place value of 21=22^1 = 221=2. It contributes slightly more to the pixel value than Bit Plane 0.
- **Effect on Image:** This bit plane captures slightly more detail than Bit Plane 0 and starts to form more recognizable features in the image.

3. Bit Plane 2:

- **Represents**: The third least significant bit of each pixel's binary value.
- **Value Contribution:** This bit has a place value of 22=42^2 = 422=4. It provides more contribution to the pixel intensity compared to Bit Planes 0 and 1.
- **Effect on Image:** Bit Plane 2 starts to show more distinct patterns and features. It adds noticeable detail to the image.

4. Bit Plane 3:

- **Represents**: The fourth least significant bit of each pixel's binary value.
- **Value Contribution:** This bit has a place value of 23=82^3 = 823=8. It significantly impacts the pixel value.
- **Effect on Image:** This bit plane contributes to more substantial image details and helps to define edges and textures.

5. Bit Plane 4:

- **Represents**: The fifth least significant bit of each pixel's binary value.
- **Value Contribution:** This bit has a place value of 24=162^4 = 1624=16. It provides even more contribution to the pixel value.
- **Effect on Image:** Bit Plane 4 adds more detail and contrast. It helps in forming clearer shapes and finer details.

6. Bit Plane 5:

- **Represents**: The sixth least significant bit of each pixel's binary value.
- **Value Contribution:** This bit has a place value of 25=322^5 = 3225=32. It has a significant effect on the pixel intensity.
- **Effect on Image:** This bit plane provides important details and contributes to the overall image contrast and texture.

7. Bit Plane 6:

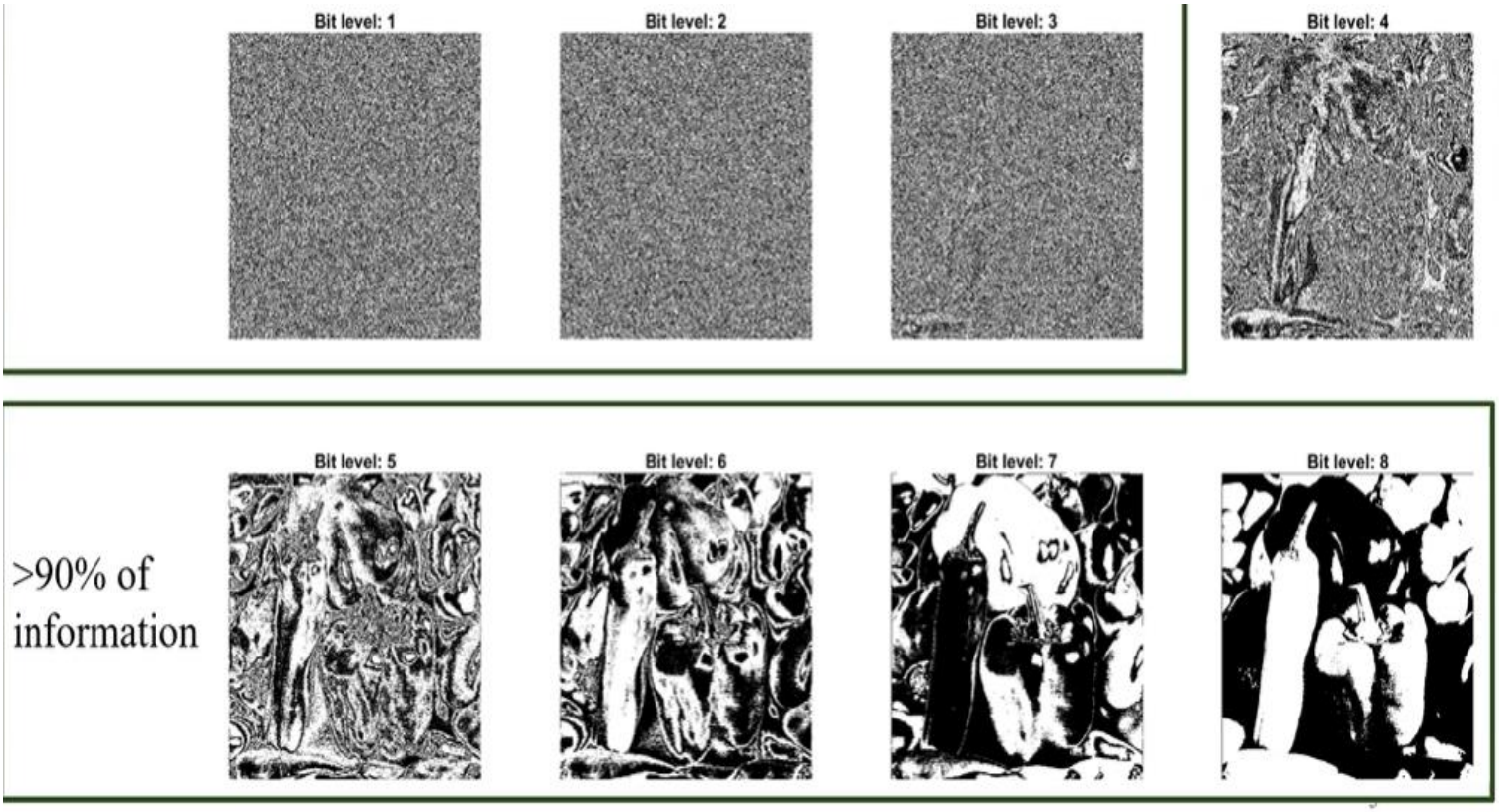
- **Represents:** The seventh least significant bit of each pixel's binary value.
- **Value Contribution:** This bit has a place value of 26=642^6 = 6426=64. It strongly impacts the pixel value.

- **Effect on Image:** Bit Plane 6 is crucial for capturing high levels of detail and contributes significantly to image clarity.

8. Bit Plane 7:

- **Represents**: The most significant bit (MSB) of each pixel's binary value.
- **Value Contribution:** This bit has a place value of 27=1282^7 = 12827=128. It contributes the most to the pixel value.
- **Effect on Image:** Bit Plane 7 has the most significant impact on the overall appearance of the image. It determines the major tonal variations and is essential for the most significant image details.

Here are some examples :



After deleting from the 1st to the 4th bit planes we see that we can also have a good image quality and this the principle used in the compression

original

Reconstruction from levels 5-8

Reconstruction from levels 6-8

original

Reconstruction from levels 5-8

Reconstruction from levels 6-8

Image encryption and decryption :

My code :

```
from PIL import Image
import numpy as np
```

```
def encrypt_img(img_path) :
    public_key = np.random.randint(0, 256)       #  Generate a random public key

    while public_key % 2 == 0:   # Check if the number is even
        public_key = np.random.randint(0, 256)  # Generate a new number if even

    print("Your public key is : ", public_key)

    private_key = 0      #Initiate the private key with a value of 0 to enter the loop
    while private_key % 2 == 0:    # Check if the number is even
        private_key = int(input("Enter your private key. It should be odd : "))      #  Prompt user
to enter their private key

    img = Image.open(img_path)      # Open the image using PIL
    img1 = img.convert("RGB")     #  Convert the image to a 'RGB' mode (if it's not already)

    # Convert the image to a NumPy array (3D array: height x width x color channels)
    img_array = np.array(img1)

    width,height = img1.size       # Get image dimensions


    # First layer of encryption: Adjust color values using the private key :
    for y in range(height) :
        for x in range(width) :
            r,g,b = img_array[y,x]
            r = (r + private_key - 100) % 256
            g = (g + private_key) % 256
            b = (b + private_key + 150) % 256
            img_array[y, x] = [r, g, b]


    # Second layer of encryption: Apply different operations to every third pixel using the private
or public key :
    for y in range(height) :
        for x in range(0,int(width - 1), 3) :
            r,g,b = img_array[y,x]
            r = (r * private_key) % 256
            g = (g * private_key) % 256
            b = (b * private_key) % 256
            img_array[y, x] = [r, g, b]

        for x in range(0,int(width - 1), 3) :
            r,g,b = img_array[y,x + 1]
            r = (r * public_key) % 256
            g = (g - public_key) % 256
            b = (b - public_key) % 256
```

```
            img_array[y, x + 1] = [r, g, b]


    # Third layer of encryption: Swap pixels in groups of 10 :
    for y in range(height) :
        for x in range(0,int(width - 10), 10) :
            img_array[y,x],img_array[y,x + 8] = img_array[y,x + 8],img_array[y,x]
            img_array[y,x + 9],img_array[y,x + 5] = img_array[y,x + 5],img_array[y,x + 9]
            img_array[y,x + 2],img_array[y,x + 7] = img_array[y,x + 7],img_array[y,x + 2]


    # Fourth layer of encryption: Combine private and public keys for further modification :
    for y in range(height) :
        for x in range(0,int(width - 1), 2) :
            r,g,b = img_array[y,x]
            r = (r * private_key * public_key) % 256
            g = (g * private_key * public_key) % 256
            b = (b * private_key * public_key) % 256
            img_array[y, x] = [r, g, b]

        for x in range(0,int(width - 1), 2) :
            r,g,b = img_array[y,x + 1]
            r = (r * private_key + public_key) % 256
            g = (g * private_key + public_key) % 256
            b = (b * private_key + public_key) % 256
            img_array[y,x + 1] = [r, g, b]


    # Convert the NumPy array back to an image
    img_modified = Image.fromarray(img_array,"RGB")

    # display and save the modified image
    img_modified.show()
    img_modified.save(output_path)


def decrypt_img(img_path) :
    private_key = int(input("enter your private key : "))
    public_key = int(input("enter your public key : "))

    img1 = Image.open(img_path)      # Open the image using PIL


    # Convert the image to a NumPy array (3D array: height x width x color channels)
    img_array = np.array(img1)

    width,height = img1.size
```

```
    # Reverse of the fourth layer of encryption: Combine private and public keys for further
modification :
    for y in range(height) :
        for x in range(0,int(width - 1), 2) :
            r,g,b = img_array[y,x]
            r = (r  * pow(private_key * public_key, -1, 256)) % 256
            g = (g  * pow(private_key * public_key, -1, 256)) % 256
            b = (b  * pow(private_key * public_key, -1, 256)) % 256
            img_array[y, x] = [r, g, b]

        for x in range(0,int(width - 1), 2) :
            r,g,b = img_array[y,x + 1]
            r = (r - public_key) * pow(private_key, -1, 256)  % 256
            g = (g - public_key) * pow(private_key, -1, 256)  % 256
            b = (b - public_key) * pow(private_key, -1, 256)  % 256
            img_array[y,x + 1] = [r, g, b]


    #  Reverse of the third layer of encryption: Swap pixels in groups of 10 :
    for y in range(height) :
        for x in range(0,int(width - 10), 10) :
            img_array[y,x],img_array[y,x + 8] = img_array[y,x + 8],img_array[y,x]
            img_array[y,x + 9],img_array[y,x + 5] = img_array[y,x + 5],img_array[y,x + 9]
            img_array[y,x + 2],img_array[y,x + 7] = img_array[y,x + 7],img_array[y,x + 2]



    # Reverse of the second layer of encryption: Apply different operations to every third pixel
using the private or public key :
    for y in range(height) :
        for x in range(0,int(width - 1), 3) :
            r,g,b = img_array[y,x]
            r = r * pow(private_key, -1, 256) % 256
            g = g * pow(private_key, -1, 256) % 256
            b = b * pow(private_key, -1, 256) % 256
            img_array[y, x] = [r, g, b]

        for x in range(0,int(width - 1), 3) :
            r,g,b = img_array[y,x + 1]
            r = (r * pow(public_key, -1 ,256)) % 256
            g = (g + public_key) % 256
            b = (b + public_key) % 256
            img_array[y, x + 1] = [r, g, b]


    # Reverse of the first layer of encryption: Adjust color values using the private key :
```

```
    for y in range(height) :
        for x in range(width) :
            r,g,b = img_array[y,x]
            r = (r - private_key + 100) % 256
            g = (g - private_key) % 256
            b = (b - private_key - 150) % 256
            img_array[y, x] = [r, g, b]


    # Convert the NumPy array back to an image
    img_modified = Image.fromarray(img_array,"RGB")

    # display ans save modified image
    img_modified.show()
    img_modified.save(output_path)



print('''
Welcome to Prodigy, Prodigy is a simple image encryption and decryption tool. Feel free to try it.
 ''')
user_choice = int(input("If you want to encrypt the image insert 1, inseert 2 for decryption : "))

print(r'''
Make sure to remove any quotation marks " or ' from the path before you enter it to avoid problems.

For example : C:\users\lenovo\desktop\test.png"
''')

img_path = input("insert your image's path : ")
output_path = input("insert the path where you want to save the encrypted or decrypted image : ")
if user_choice == 1 :
    encrypt_img(img_path)
elif user_choice == 2 :
    decrypt_img(img_path)
else :
    print("error")
```

In our code we used 2 keys, one is generated randomly while the other is given by the user

Using ‘pow’ for Modular Inverse:

Python's built-in pow function can calculate the modular inverse directly. When you multiply a number by its modular inverse under a specific modulus, the result is 1. This

property is crucial in cryptography, where operations must be reversible. By using pow() with a negative exponent (-1), you ensure that the multiplication performed during encryption can be undone, effectively decrypting the image.

If during encryption, a pixel value was modified by multiplying it with the private_key, then during decryption, you need to multiply the altered value by the modular inverse of private_key to retrieve the original pixel value.

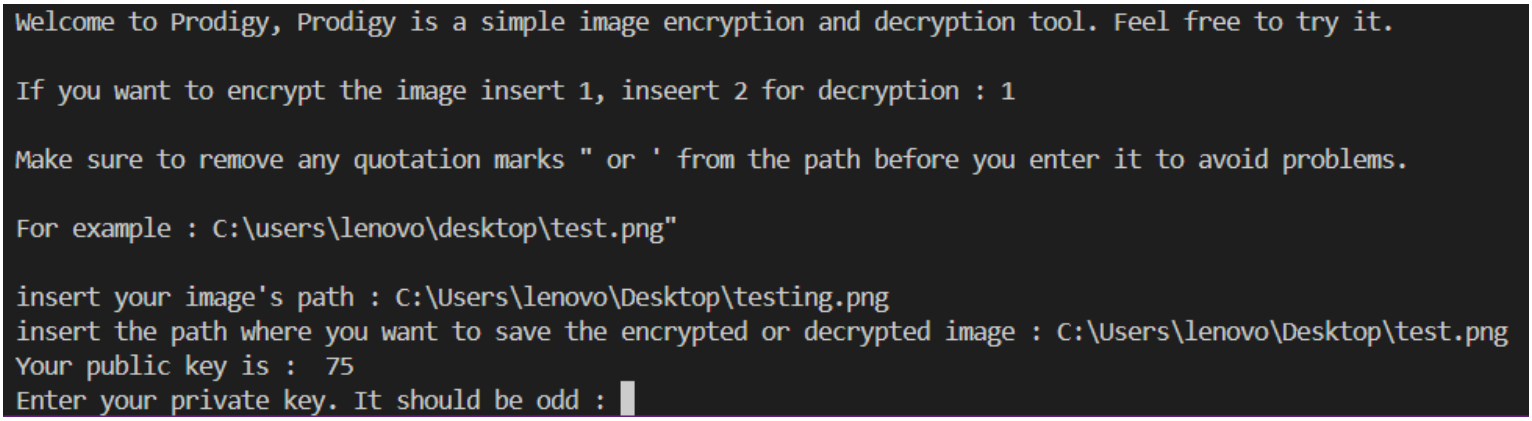
Suppose during encryption you had:
$\text{encrypted\_value} = (\text{original\_value} \times \text{private\_key} \times \text{public\_key}) \bmod 256$

To decrypt: $\text{original\_value} = (\text{encrypted\_value} \times \text{modular\_inverse}) \bmod 256$

The modular inverse of a number a modulo m exists if and only if a and m are coprime (i.e., gcd(a, m) = 1).

And 256 is an even number so it can only be coprime with odd numbers that's why we only used the odd numbers as keys.

The result :



Original image :

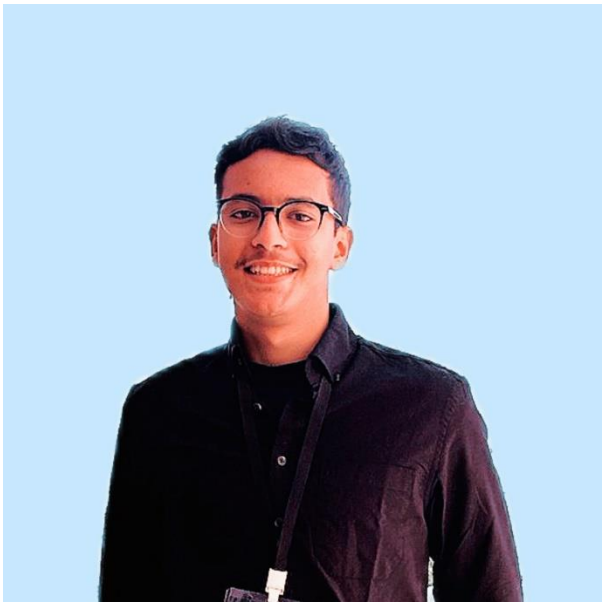


Image after encryption :