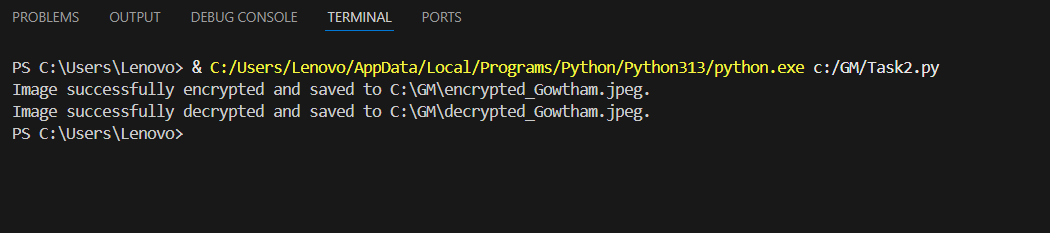
**Task 2:** Develop a simple image encryption tool using pixel manipulation. Support operations like swapping pixel values or applying a basic mathematical operation to each pixel.

Input code:



Output:



What all operations, features are included in the code and working of it ?

The operations:

1. **Bitwise XOR Operation (**numpy.bitwise\_xor**)**:

* **Purpose**: This function performs an element-wise XOR (exclusive OR) operation between two arrays. In the code, it's used to modify the pixel values of the image by combining them with a generated key stream. This process is reversible, allowing the original pixel values to be restored during decryption.
* **How It Works**:

1. Each pixel's color value (an integer) is combined with the corresponding value from the key stream using the XOR operation.
2. In binary terms, XOR outputs 1 if the corresponding bits of the operands differ, and 0 if they are the same.
3. This effectively encrypts the pixel data. During decryption, applying the same XOR operation with the same key stream restores the original pixel values.
4. **Shuffling Pixel Positions (**numpy.random.shuffle**)**:

* **Purpose**: This function randomly rearranges the elements of an array along the first axis. In this code, it's used to shuffle the positions of the pixels in the image, adding another layer of obfuscation.
* **How It Works**:

1. A seed value is generated from the key stream to ensure that the shuffling can be reproduced during decryption.
2. The pixel indices are then shuffled based on this seed, rearranging the pixel positions in a deterministic yet seemingly random manner.
3. This shuffled arrangement is saved during encryption so that it can be reversed during decryption.
4. During decryption, the saved permutation is used to restore the pixels to their original positions.

By combining these two operations bitwise XOR for modifying pixel values and shuffling for rearranging pixel positions the code effectively encrypts the image. The process is reversible, allowing the original image to be recovered when the same key is used for decryption. Here the Bitwise XOR is a mathematical operation and the Shuffling pixel positions is a permutation operation often used in image processing and encryption.

The working:

1. **Loading and Preparing the Image:**

* The program opens the specified image file and converts it to RGB format to ensure consistency.​
* It then transforms the image into a NumPy array, allowing for efficient numerical operations on pixel data.​

1. **Handling the Encryption Key:**

* The user provides a key as a string (e.g., '131202').
* This key is converted into bytes, which will be used to generate a unique sequence for encrypting or decrypting the image.​

1. **Generating a Key Stream:**

* A sequence of bytes (key stream) is created by repeatedly hashing the key combined with a counter using the SHA-256 algorithm.​
* This process continues until the key stream is long enough to match the total number of pixel values in the image.​

1. **Shuffling Pixel Positions:**

* A seed for the random number generator is derived from the first few bytes of the key stream.​
* Using this seed, the program generates a random permutation of pixel indices, effectively shuffling the pixel positions in a consistent and reversible manner.​

1. **Encrypting the Image:**

* The pixels are rearranged according to the shuffled indices.​
* Each pixel's color values are then modified by performing a bitwise XOR operation with the corresponding values from the key stream. This step obscures the pixel data, making the image appear noisy and unrecognizable.​
* The permutation indices are saved to a file (e.g.,'encrypted\_Gowtham.jpeg\_permutation.npy') to facilitate accurate decryption later.

​

1. **Decrypting the Image:**

* To revert the image to its original form, the program reads the saved permutation indices.​
* It applies the XOR operation again using the same key stream, effectively reversing the encryption.​
* The pixels are then rearranged back to their original positions using the inverse of the initial shuffle, restoring the image to its original appearance.​

1. **Error Handling:**

* The code includes mechanisms to catch and report errors, such as issues with file paths, missing permutation files, or invalid modes (i.e., not 'encrypt' or 'decrypt').​

1. **Saving the Processed Image:**

* After processing, whether encryption or decryption, the image is reshaped to its original dimensions.​
* The final image is then saved to the specified output path, allowing the user to view the encrypted or decrypted image as needed.

Conclusion:

This code provides a method to secure images by scrambling their pixel positions and altering their color values based on a user-provided key. The same key is essential for both encrypting and decrypting the image, ensuring that only authorized users can access the original content.