Image Encryption and Compression

[**Sample Input/Output 3**](#_heading=h.30j0zll)

[**Problem Definition 4**](#_heading=h.1fob9te)

[**Image Encryption 4**](#_heading=h.3znysh7)

[**Image Compression 4**](#_heading=h.2et92p0)

[**Some Usages 4**](#_heading=h.tyjcwt)

[**Related Image Terminologies 5**](#_heading=h.3dy6vkm)

[**FIRST: Encryption/Decryption 7**](#_heading=h.1t3h5sf)

[**Linear Feedback Shift Register 7**](#_heading=h.4d34og8)

[**Image Encoding/Decoding 9**](#_heading=h.2s8eyo1)

[**SECOND: Compression/Decompression 9**](#_heading=h.17dp8vu)

[**Compression by Huffman Coding 9**](#_heading=h.3rdcrjn)

[**Decompression using the Huffman Tree 11**](#_heading=h.lnxbz9)

[**Project Requirements 12**](#_heading=h.35nkun2)

[**Required Implementation 12**](#_heading=h.1ksv4uv)

[**Input & Output 13**](#_heading=h.44sinio)

[**Test Cases 13**](#_heading=h.z337ya)

[**Deliverables 13**](#_heading=h.3j2qqm3)

[**Implementation (60%) 13**](#_heading=h.1y810tw)

[**Document (40%) 14**](#_heading=h.4i7ojhp)

[**Allowed Codes 14**](#_heading=h.2xcytpi)

[**Milestones 14**](#_heading=h.1ci93xb)

[**BONUSES 14**](#_heading=h.3whwml4)

[**Appendix: Template Code Description 15**](#_heading=h.qsh70q)

# Sample Input/Output





# Problem Definition

## Image Encryption

It is the process of encoding an image in such a way that only authorized person can access it. This is usually done by using a key (i.e. password). Anyone knowing this password can get the original image back, but another password won't work. Thus, for example, you could post an encrypted image on the web, but only friends who have the password (and your program) can see the original. Here, a simple algorithm called **Linear Feedback Shift Register** is used for this purpose. It will be explained later.

## Image Compression

It is a type of data compression applied to digital images, to reduce their cost for storage or transmission. One of the common data compression methods is the **Huffman Coding**. Its basic idea is that instead of storing each color channel as an 8-bit value, it stores the more frequently occurring color values using fewer bits and less frequently occurring color values using more bits.

For example, assume a gray-scale image of size 64×64 and the following pixel values:

| Grayscale value | 10 | 100 | 200 | 255 |
| --- | --- | --- | --- | --- |
| Number of pixels | 1792 | 256 | 128 | 1920 |

Without compression, each pixel value will be represented by 8 bits (1 byte). The total size is:

Size without compression = 64 × 64 × 8 bits = 32,768 bits = 4 KB

In Huffman Coding, the more frequently values will be represented by less number of bits, as follows:

| Grayscale value | 10 | 100 | 200 | 255 |
| --- | --- | --- | --- | --- |
| Binary represented value | (10)2 | (110)2 | (111)2 | (0)2 |

Thus, the total required size in this case will be:

Size with compression = 1792 × 2 bits + 256 × 3 bits + 128 × 3 bits + 1920 × 1 bit = 6,656 bits ≈ 0.8 KB

So, the compression ratio in this case will be:

## Some Usages

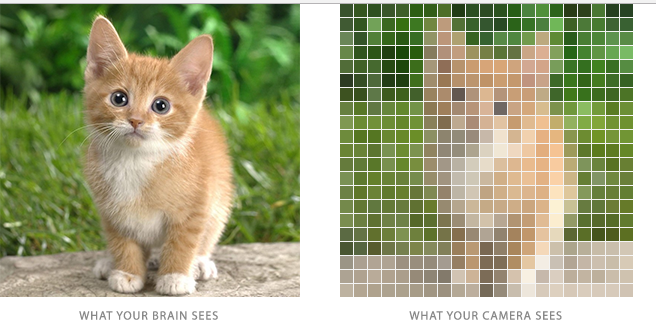
1. Secured and fast image transmission (e.g. MMS)
2. Secured image storage (e.g. on mobile phones)

## Related Image Terminologies

### Digital Image

It’s an electronic snapshot taken of a scene or scanned from documents, such as photographs, manuscripts, printed texts, and artwork.

It’s made of picture elements called pixels. Typically, pixels are organized in an ordered rectangular array. Each pixel has its own intensity value, or brightness which is represented in binary code (zeros and ones).



### Color depth

Also known as **bit depth**, is the number of bits used to indicate the color of a single pixel.

The size of an image is determined by two factors:

1. Dimensions: width and height of the 2D pixel array.
2. Color depth: number of bits per pixels

### Intensity

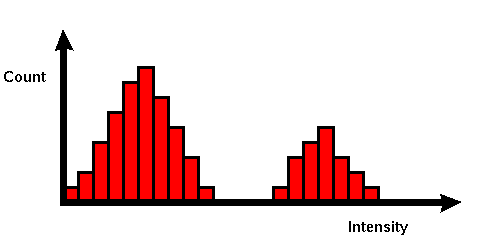
The intensity of a pixel is expressed within a given range between a minimum and a maximum value [Inclusive], based on the color depth of the pixel.

True Color images have intensity from the darkest (0) and lightest (255) of three different color channels, **R**ed, **G**reen, and **B**lue. Each channel has a range from 0 to 255 as shown in Figure below. So we need 8+8+8=24 bits to represent 1 pixel color which means we have 224 = 16,777,216 different colors.

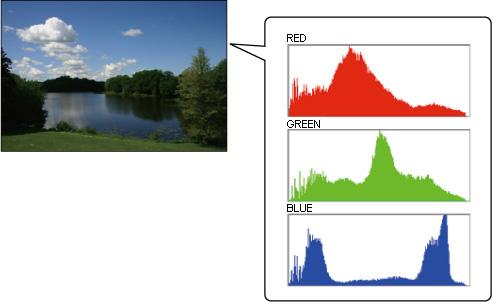


### Histogram

An **image histogram** is a graph showing the number of pixels (i.e. frequencies) in an image at each different intensity value found in that image.



Usually, there’s a different histogram for each channel (**R**ed, **G**reen, and **B**lue). Since each channel is represented by 8 bits, so, there are 256 possible values on x-axis. The histogram will graphically display 256 numbers showing the distribution of pixels amongst those values.



# FIRST: Encryption/Decryption

## Linear Feedback Shift Register

A linear feedback shift register is a simple algorithm that can be used to implement a simple encode/decode facility for images. For example, in figure 1, a sample input image is shown with its encoding. As shown the encoded image shows nothing from the original image.



*Original After Encoding After Decoding*

**Figure 1: Example of encryption and decryption using the linear feedback shift register**

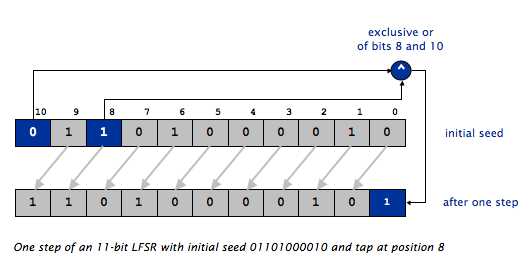
How Linear Feedback Shift Register works?

A linear feedback shift register is a register of bits that performs discrete step operations that shift all the bits one position to the left and replace the vacated bit by the exclusive or (XOR) of the bit shifted off and the bit at a given **tap position** in the register.

A LFSR has three parameters that characterize the sequence of bits it produces:

1. The number of bits N
2. The tap position: tap
3. The initial seed (the sequence of bits that initializes the register)

For example, if the initial seed has **01101000010** and tap position is **8** after one step it will be **11010000101** as shown in figure 2



**Figure 2 LFSR example**

LFSR can be used in 2 modes

1. *Simulate one step*

In this mode, only 1 step as shown above is done and the bit is returned. Here is an example of calling the function multiple times with initial seed **01101000010**

|  | **Register value** | **Returned Value** |
| --- | --- | --- |
| After 1st call | 11010000101 | 1 |
| After 2nd call | 10100001011 | 1 |
| After 3rd call | 01000010110 | 0 |
| After 4th call | 10000101100 | 0 |
| After 5th call | 00001011001 | 1 |
| After 6th call | 00010110010 | 0 |
| After 7th call | 00101100100 | 0 |
| After 8th call | 01011001001 | 1 |
| After 9th call | 10110010010 | 0 |
| After 10th call | 01100100100 | 0 |

1. Extracting multiple bits

It takes an integer **k** as an argument and returns a k-bit integer obtained by simulating k steps of the LFSR. Here is an example of calling the function multiple times with initial seed **01101000010** and k = **5**

|  | **Register value** | **Returned Value (k-bit binary)** | **Returned Value (decimal)** |
| --- | --- | --- | --- |
| After 1st call | 00001011001 | 11001 | 25 |
| After 2nd call | 01100100100 | 00100 | 4 |
| After 3rd call | 10010011110 | 11110 | 30 |

## Image Encoding/Decoding

* Given an image, an initial seed and tap position,
* For each pixel (x, y), in the order (0, 0), (0, 1), (0, 2), ...,

1. Extract the **red, green, and blue** components of the color (note that each component is an integer between 0 and 255).
2. XOR the red component with 8 newly generated bits. Do the same for the green (using another 8 newly generated bits) and, similarly, the blue. Create a new color using the result of the xor operations.
3. Set the pixel to that color.

Since the algorithm is based on XOR so the same steps are done in encoding and decoding

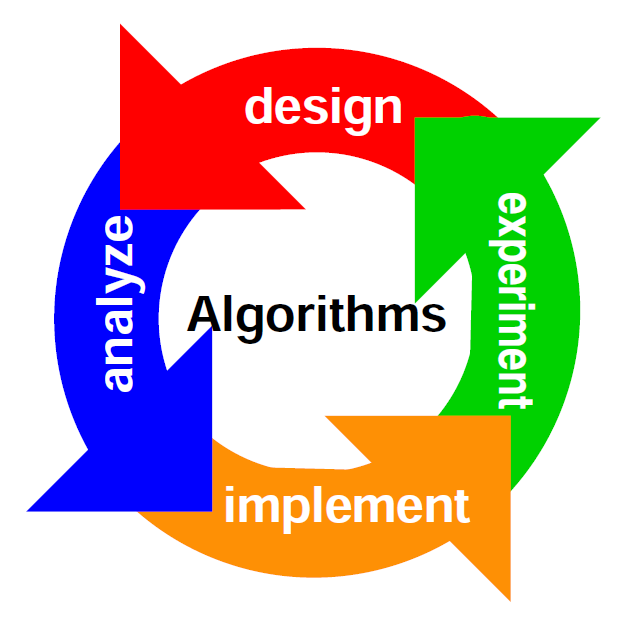
# SECOND: Compression/Decompression

## Compression by Huffman Coding

To encode the color values using Huffman Coding, **FIVE** steps are required per color channel (i.e. **red**, **green** and **blue**):

1. Calculate the frequency (i.e. number of pixels) for each value
2. Use a Greedy algorithm to build up a Huffman Tree, such that
   1. smaller frequencies at bottom of the tree while larger frequencies at top
   2. assign codes to the tree by **placing** a **0** on every **left** branch and a **1** on every **right** branch
   3. use **priority queue** for efficient implementation of selecting the minimum at each time
3. A **traversal** of the tree from root to leaf give the Huffman code for that particular leaf value
4. Replace each value by its corresponding Huffman code
5. Store the generated Huffman code stream together with the Huffman Tree… **WHY the tree?**

For example, assume we need to compress the following 128×128 colored image:



**Figure 3: Colored image of size 128x128**

**The first step**, calculate the frequencies of each color channel:

| **RED** | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | 0 | 32 | 64 | 96 | 128 | 160 | 166 | 192 | 224 | 255 |
| **#Pixels** | 3354 | 9 | 11 | 29 | 52 | 8 | 21 | 28 | 1664 | 11208 |

| **GREEN** | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | 0 | 32 | 64 | 96 | 128 | 160 | 192 | 202 | 220 | 224 | 251 | 255 |
| **#Pixels** | 3311 | 4 | 34 | 39 | 42 | 1563 | 36 | 21 | 15 | 1676 | 127 | 9516 |

| **BLUE** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Value** | 0 | 64 | 128 | 192 | 240 | 255 |
| **#Pixels** | 4830 | 67 | 80 | 140 | 148 | 11119 |

**Second**, construct the Huffman Coding Tree for each color channel, use **priority queue** to select the minimum at each time:



**Figure 4: Huffman tree of the red channel for the image in Figure 3**

Do the same for the **green** and the **blue** channels…

**Third**, traverse each tree to get the corresponding binary code for each color value:

| **RED** | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | 0 | 32 | 64 | 96 | 128 | 160 | 166 | 192 | 224 | 255 |
| **Binary code** | (10)2 | (1111110)2 | (111110)2 | (11110)2 | (11100)2 | (1111111)2 | (111010)2 | (111011)2 | (110)2 | (0)2 |
| **#Pixels** | 3354 | 9 | 11 | 29 | 52 | 8 | 21 | 28 | 1664 | 11208 |

Do the same for the **green** and the **blue** channels…

**Fourth**, replace each color value with its corresponding binary code. Thus, for **red** channel, all pixels with value 0 will be replaced by (10)2, all pixels with value 32 will be replaced by (1111110)2 and so on…

For example:

0, 255, 32, 64, 0, 64, 64 … 🡺 (100111111011111010111110111110…)2

The first 10 encode '0', then the next 0 encode the '255', then 1111110 encode 32, and so on as follows:

10|0 |1111110|111110|10|111110|111110 …

0 |255|32 |64 |0 |64 |64 …

So, the compression of the **red** channel will be as follows:

Original size = 128 × 128 × 8 bits = 16 KB

Compressed size = #pixels of 0 × 2 bits + #pixels of 32 × 7 bits + #pixels of 64 × 6 bits +…

Compressed size = 3354×2 + 9×7 + 11×6 + 29×5 + 52×5 + 8×7 + 21×6 + 28×6 + 1664×3 + 11208×1 = 2.9 KB

Compression ratio of **red** channel only = 2.9 KB / 16 KB × 100 ≈ **18.2%**

## Decompression using the Huffman Tree

To decompress the image, **TWO** steps are required for each color channel (i.e. **red**, **green** and **blue**):

1. Reconstruct the Huffman Tree from the compressed file
2. Use this tree to decode the stored binary stream as follows:
   1. Start at the root of the tree.
   2. Repeat until you reach an external leaf node.
      1. Read one bit from the stream.
      2. Take the left branch in the tree if the bit is 0; take the right branch if it is 1.
   3. Print the value in that external node

Following are examples of traversing the **red** channel tree to construct the **first three values**:



**Figure 5: Example of decoding the first three values using the Huffman Tree**

# Project Requirements

## Required Implementation

| **Requirement** | **Performance** |
| --- | --- |
| 1. Image encryption using Linear Feedback Shift Register | **Time:** should be **bounded by O(N2)**, N is one image dimension (width/height) |
| 1. Image decryption using Linear Feedback Shift Register | **Time:** should be **bounded by O(N2)**, N is one image dimension (width/height) |
| 1. Image compression using Huffman Coding with **priority queue** | **Time:** should be **bounded by O(N2 + C log(C)**), N is one image dimension (width/height). C is the number of distinct values |
| 1. Image decompression using Huffman Coding | **Time:** should be **bounded by O(N2)**, N is one image dimension (width/height) |
| 1. Save and load the binary encoded and compressed file | **Time:** should be **bounded by O(N2)**, N is one image dimension (width/height) |

## Input & Output

**Stage1:** Image Encoding and Compression

1. Input
   * Image (2D array of pixels)
   * Initial seed
   * Tap position
2. Output

Binary file contains the following:

* 1. Initial seed value and tap position
  2. Huffman Tree
  3. Compressed image

**Stage2:** Image Decompression and Decoding

1. Input

Binary file that is output from stage1

1. Output

Image (should be the same image before encoding)

## Test Cases

### Sample Test:

* **Goal:** test the correctness
* **Given:** six sample images with the expected O/P of both phases (Encryption & Compression)
* **Check the** [**attached folder**](about:blank)

### Complete Test:

* **Goal:** test the efficiency (beside correctness)
* **Given:** 3 levels
  1. Small: image size O(700 KBs)
  2. Medium: image size O(8 MBs)
  3. Large: image size O(200 MBs)
* **Check the** [**attached folder**](about:blank)

# Deliverables

## Implementation (60%)

1. Image encryption using Linear Feedback Shift Register
2. Image decryption using Linear Feedback Shift Register
3. Image compression using Huffman Coding with **priority queue**
4. Image decompression using Huffman Coding
5. Save and load the binary encoded and compressed file

## Document (40%)

1. Code of image encryption.
2. Code for constructing the Huffman Tree.
3. Code for image compression.
4. Code for image decompression.
5. Detailed analysis of the above codes.
6. Compression ratio of complete test cases **WITH** and **WITHOUT** encryption.

## Allowed Codes

1. Open-source code for **Priority Queue [**You **MUST** **understand** and **analyze** it**]**
2. Given template to open and display the images. (check the [**attached folder**](about:blank))

Refer to [**Appendix: Template Code Description**](#_heading=h.2bn6wsx) for more details.

# Milestones

|  | **Deliverables** | **Due to** |
| --- | --- | --- |
| **Milestone1** | 1. Image encryption using Linear Feedback Shift Register 2. Image decryption using Linear Feedback Shift Register 3. Construction of the Huffman Tree using priority queue 4. Image compression using Huffman Coding with priority queue 5. Image decompression using Huffman Coding 6. Save and load the binary encoded and compressed file 7. Documentation | Final Delivery  [LAB EXAM WEEK] |
| * + **MUST** deliver the required tasks and **ENSURE** it’s worked correctly   + **MUST** deliver in your scheduled time (TO BE ANNOUNCED) | | |

# BONUSES

1. Image encryption: use **alphanumeric password** instead of binary password (since binary password provides weak protection and inconvenient)
2. Image encryption: write a code to **break the encrypted** image. Assume that you know the password has N bits, write a program that tries all possible seeds and all possible taps to decrypt the image.

Hint: all but the correct seed and tap will produce pseudo-random colors, so you can track the frequencies of each 8-bit value and pick the seed and tap that results in the frequencies that deviate the most from 128.

# Appendix: Template Code Description

C# Code contains **ImageOperations** class with the following functionalities:

1. Open image & load it in a 2D array stored in a global variable of type MyImage[[1]](#footnote-0) [,]called ImageMatrix

MyImage [,] OpenImage(string ImagePath)

1. Get width and height of the image matrix

int GetHeight(MyImage[,] ImageMatrix)

int GetWidth(MyImage[,] ImageMatrix)

1. Display an image on a given PictureBox control

void DisplayImage(MyImage[,] ImageMatrix,PictureBox PicBox)

1. MyImage is a structure defined in the code to hold the Red, Green, Blue values of each pixel [↑](#footnote-ref-0)