Image Encryption and Compression

A quick overview of this tool is that it encrypts images and compresses them, moreover it allows the images to be only accessible by an authorized user using a specific password.

This allows the user to post images online without the worry of them getting misused by users that don't have access.

The Algorithm used here is called "**Linear Feedback Shift Register**" (LFSR). Also, we used a couple of different tools to help us in the process like Huffman Trees and other tools, which will all be explained later.

We'll discuss the main functions of the project one by one.

Image Encryption/Decryption

- **LSFR (Linear Feedback Shift Register)**

It takes as parameters : size of the seed (referred to as "n"), the seed itself represented as a binary array (referred to "seed"), tap position (referred to as "tapPosition").

This process happens eight times, it aims to change the color of each pixel so that the resulting image is just noise and has absolutely no meaning unless you have the password to reverse the encrypting process.

It takes the most significant bit of the original seed and uses the exclusive-or operation on it with the bit of index of "tapPosition" and stores it in a temporary variable and then shifts the seed to the left by one bit and concatenates the temporary variable to the shifted seed.

Time Complexity O(n)

- **EncryptImage Function**

It takes as parameters the targeted image (referred to as "image"), the size of the seed (referred to as "seedSize"), the seed itself (referred to as "seed") and the tap position (referred to as "tapPosition).

The function iterates over the pixels of the image and calls the EncryptPixel function.

Time Complexity O($H \cdot W \cdot n$)

- **EncryptPixel Function**

It takes as parameters the targeted pixel (referred to as "pixel"), the size of the seed (referred to as "seedSize"), the seed itself (referred to as "seed") and the tap position (referred to as "tapPosition).

This function just calls the LSFR function on each color of the main three colors of the targeted pixel (Red, Green and Blue).

Time Complexity O(n)

- **BruteColorAvg**

It takes as parameters the targeted image (referred to as "image"), the size of the seed (referred to as "n") and tap position (referred to as "tapPosition").

This function iterates over all the pixels ($H \cdot W$) and tries to apply the current seed and tap position to the image and then returns the average color value of the summation of the values of red, green and blue over all pixels.

$$\frac{\left(\sum_{i=1}^{H}\sum_{j=1}^{W}\left(pixel_{ij_{red}} + pixel_{ij_{green}} + pixel_{ij_{blue}}\right)\right)}{H \cdot W \cdot 3}$$

Time Complexity O($H \cdot W$)

- **BruteHelper function**

It takes as parameters the targeted image (referred to as "image"), the size of the seed (referred to as "n") and keeps track of the best tapPosition and current best value (which deviated the most from 128).

The function just brute forces all the possible seeds (2^nseeds) with all the possible tap positions (n). On each iteration it calls the BruteColorAvg function which calculates for the current seed and tap position what is the deviation from it and 128 and if it's greater than the current best answer it updates it and updates the best seed and best tap position.

Time Complexity O($2^N \cdot N \cdot H \cdot W$)

- **HashSeed**

The main objective of this function is to convert the seed from a string into a binary number.

It takes as parameters a string that represents the seed and converts it into a unique integer using a **polynomial rolling hash function**, it stores the result of this hash in a temporary variable and then converts this hash into a binary number.

Time Complexity O(N)

Image Compression

It is a type of compression applied to digital images, to reduce their cost for storage or transmission. One of the common data compression methods is 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 values using more bits.

- **MinHeap**
  Used MinHeap as a priority queue to extract lower frequencies first to ensure logarithmic time for the algorithm.

  The way it works is when inserting or popping an element from the heap, it extracts "highest priority element" and puts it at the "peek" position, in the case of the min-heap the highest priority element is the minimum element.

  Time Complexity O($\log(n)$)

- **Build**
  We calculate the two least frequent colors and merge them into one node, each node has left and right children, pointing toward an internal node having the sum of the two frequencies we calculated, we repeat the following step until the heap has one element remaining which is the root. Now we have created the Huffman Tree. Since in every iteration we add a node and remove 2 then the total number of iterations will be $n - 1$

  Time Complexity O($N \cdot log(N)$)

- **Traverse**
  We start traversing the tree from the root, if we visit the left child, we add 0, if we visit the right child, we add 1, we stop the traversal once we reach a leaf (no left child and no right child) and the color is then encoded into a BitArray.

  Note that: the function traverses the entire tree, and the number of nodes of the tree is at most around $2 \cdot n - 1$

  Time Complexity O(n)

- **Freq_RED**
  Calculates frequencies for each color value of the red channel.

  Time Complexity: O($H \cdot W$)

- **Freq_GREEN**
  Calculates frequencies for each color value of the green channel.

  Time Complexity: O($H \cdot W$)

- **Freq_BLUE**
  Calculates frequencies for each color value of the blue channel.

  Time Complexity: O($H \cdot W$)

- **BuildHuffman_red**
  Stores the frequency of the red channel in a dictionary and builds Huffman Tree for it.

  The function each of: Freq_RED, build (in Huffman tree), and traverse_set (in Huffman tree) which is $O(H \cdot W) + O(n) + O\big(nlog(n)\big)$

  Time Complexity: $O\big(H \cdot W + nlog(n)\big)$

- **BuildHuffman_green**
  Stores the frequency of the green channel in a dictionary and builds Huffman Tree for it.

  The function each of: Freq_GREEN, build (in Huffman tree), and traverse_set (in Huffman tree) which is $O(H \cdot W) + O(n) + O\big(nlog(n)\big)$

  Time Complexity: $O\big(H \cdot W + nlog(n)\big)$

- **BuildHuffman_blue**
  Stores the frequency of the blue channel in a dictionary and builds Huffman Tree for it.

  The function each of: Freq_BLUE, build (in Huffman tree), and traverse_set (in Huffman tree) which is $O(H \cdot W) + O(n) + O\big(nlog(n)\big)$

  Time Complexity: $O\big(H \cdot W + nlog(n)\big)$

- **CompressImage**
  Takes an image as a parameter, it creates 3 Huffman trees (one for each color channel), then we create three dictionaries using the Huffman trees to store the encode of each color channel in it.

  Then we iterate over the image merging the BitArrays of the encodes of every pixel into one big BitArray for the entire picture.

  Time Complexity: O($H \cdot W \cdot k_{ij} + Nlog(N)$) where k_{ij} is the size of the encode of the $pixel_{ij}$

Decompression

This is where we restore the original image using a compressed image so that we're able to properly display and use it normally again.

- **DecompressImage**
  Takes an image as a parameter, it works by holding an index (used to point at the image BitArray) and iterating over the dimensions of the image, we first search for the red channel value by using the BitArray values to traverse the red channel Huffman tree (where if the value pointed at by the index is 0 it traverses left, and if it's 1 it traverses right), it does that till it finds a leaf (a node that doesn't have either a left or a right node) and sets the red value for the $pixel_{ij}$ to the corresponding value in the tree, then it does the same for the other 2 channels (green then blue) and once its done with the blue one the main loop advances to the next pixel to repeat the algorithm till the end of the picture.

  Time Complexity: O($H \cdot W \cdot k_{ij}$) where k_{ij} is the size of the encode of the $pixel_{ij}$

`Codes

```
public static byte LSFR(int n, BitArray seed, int tapPosition)
{
    byte ret = 0;
    for(byte i = 0; i < 8; i++)
    {
        bool value = seed[n - 1] ^ seed[tapPosition];
        ret |= Convert.ToByte(Convert.ToByte(value) << (7-i));
        for (int j = n - 1; j > 0; j--)
            seed[j] = seed[j - 1];
        seed[0] = value;
    }

    return ret;
}
```

The loop in the code loops a constant 8 iterations, the inner loop though loops on the size of the seed n (could be further optimized to $O(1)$ with bitwise shift if the size of the seed would be guaranteed to fit in an int/long datatype), everything else is just constants, thus the complexity of the function is $\boldsymbol{O(n)}$

```
public static void EncryptPixel(ref RGBPixel pixel, int seedSize, BitArray
seed, int tapPosition)
{
    pixel.red ^= LSFR(seedSize, seed, tapPosition);
    pixel.green ^= LSFR(seedSize, seed, tapPosition);
    pixel.blue ^= LSFR(seedSize, seed, tapPosition);
    return;
}
```

The function just uses LSFR on all the 3 color channels, thus the complexity is the same as LSFR $O(n)$.

```
public static void EncryptImage(ref RGBPixel[,] image, int seedSize,
BitArray seed, int tapPosition)
{
    int h = GetHeight(image);
    int w = GetWidth(image);
    for(int i = 0; i < h; i++)
    {
        for(int j = 0; j < w; j++)
        {
            EncryptPixel(ref image[i, j], seedSize, seed, tapPosition);
        }
    }
}
```

The function iterates over the entire image ($O(H \cdot W)$) and applies the EncryptPixel function which is $O(N)$ thus in total the complexity is $O(H \cdot W \cdot N)$

```
public class HuffmanTree
{
    public HuffmanNode root { get; set; }
    public void build(Dictionary<byte, int> FreqTable)
    {
        MinHeap pq = new MinHeap(FreqTable.Count);

        foreach (var row in FreqTable)
        {
            HeapNode node = new HeapNode(new HuffmanNode(row.Key), row.Value);
            pq.Add(node);
        }

        while (pq._size > 1)
        {
            HeapNode left = pq.Pop();
            HeapNode right = pq.Pop();

            HeapNode internal_node = new HeapNode(new HuffmanNode(0), left.freq +
right.freq);

            internal_node.node.left = left.node;
            internal_node.node.right = right.node;

            pq.Add(internal_node);
        }

        root = pq.Peek().node;
    }

    public void traverse_set(Dictionary<byte, BitArray> encode, HuffmanNode node,
BitArray byt, int idx = 0)
    {
        // root is doz
        if (node == null) return;

        if (node.left == null && node.right == null)
        {
            // leaf dozer
            encode[node.color] = new BitArray(idx);
            for (int i = 0; i < idx; i++)
                encode[node.color][i] = byt[i];
            return;
        }

        traverse_set(encode, node.left, byt, idx + 1);
        byt[idx] = true;
        traverse_set(encode, node.right, byt, idx + 1);
        byt[idx] = false;
    }

}
```

The Huffman tree consist mainly of 2 functions, the build which builds the Huffman tree using a frequency table (dictionary), it inserts all unique color values with their frequencies in the minheap used as a priority queue $\left(O\left(Nlog(N)\right)\right)$ then a while loop that persists until the priority queue has only 1 element left in it, and since in every iteration we remove to elements and add one new it's equivalent to reducing the size by 1 every iteration, thus total iterations is $n-1$ thus $O(N)$, in every iteration we used the priority queue via popping and insertion which are both $\log(N)$ operations, thus the total complexity of the loop is $O(Nlog(N))$. Thus, the entire complexity is $\boldsymbol{O\left(Nlog(N)\right)}$.

For the traverse_set function it's noted that it recursively traverses the tree (similar to DFS) while backtracking the value of the BitArray to set it depending on whether the traversal is right or left, since the function traverses every node once, and the total number of nodes is at most around $2 \cdot n - 1$ (according to the way we built the tree initially), then the complexity is $O(n)$

```
public static Dictionary<byte, int> Freq_RED(RGBPixel[,] image)
{
    Dictionary<byte, int> freq_red = new Dictionary<byte, int>();

    int height = ImageOperations.GetHeight(image);
    int width = ImageOperations.GetWidth(image);

    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            byte red = image[i, j].red;
            if (freq_red.ContainsKey(red))
                freq_red[red]++;
            else
                freq_red.Add(red, 1);
        }
    }

    return freq_red;
}
```

The Freq_RED function is one of 3 functions that calculate the frequency of each channel color values, the function works by iterating over the entire image ($O(H \cdot W)$) and checking if the color value appeared as a key in the frequency dictionary (hashmap) before, if not it creates it with frequency 1, otherwise int increments the frequency by 1 ($O(1)$). Thus, the total complexity of the function is $\boldsymbol{O(H \cdot W)}$.

```
    public static void BuildHuffman_red(Dictionary<byte, BitArray> encode,
RGBPixel[,] image, ref HuffmanTree huffmanTree_red)
    {
        Dictionary<byte, int> freqred = Freq_RED(image);
        BitArray b = new BitArray(64);
        huffmanTree_red.build(freqred);
        huffmanTree_red.traverse_set(encode, huffmanTree_red.root, b, 0);

    }
```

The BuildHuffman_red function is one of 3 functions for building the Huffman tree which takes as arguments the image, a dictionary for the encodes, and the Huffman tree to be created, which simply just calls the Freq_RED function ($O(H \cdot W)$), creates a BitArray of fixed size big enough to store the maximum possible encode $\big(O(1)\big)$, creates the Huffman tree $(O\big(Nlog(N)\big)$, then passes the created bitset to the traverse function to calculate the encodes and store them in the encode dictionary. Thus, total complexity is $\boldsymbol{O\big(H \cdot W + Nlog(N)\big)}$.

```
    public static CompressedImage CompressImage(RGBPixel[,] image)
    {
        int height = ImageOperations.GetHeight(image);
        int width = ImageOperations.GetWidth(image);

        Dictionary<byte, BitArray> encodeR = new Dictionary<byte, BitArray>();
        Dictionary<byte, BitArray> encodeG = new Dictionary<byte, BitArray>();
        Dictionary<byte, BitArray> encodeB = new Dictionary<byte, BitArray>();


        HuffmanTree red_h = new HuffmanTree();
        HuffmanTree green_h = new HuffmanTree();
        HuffmanTree blue_h = new HuffmanTree();

        BuildHuffman_red(encodeR, image, ref red_h);
        BuildHuffman_Green(encodeG, image, ref green_h);
        BuildHuffman_Blue(encodeB, image, ref blue_h);

        CompressedImage compressedImage = new CompressedImage(height, width, red_h,
green_h, blue_h);

        List<bool> imageBuilder = new List<bool>(height * width);
        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                byte red = image[i, j].red;
                byte green = image[i, j].green;
                byte blue = image[i, j].blue;

                BitArray redBits = encodeR[red];
                BitArray greenBits = encodeG[green];
                BitArray blueBits = encodeB[blue];

                for (int k = 0; k < redBits.Length; k++)
                    imageBuilder.Add(redBits[k]);

                for (int k = 0; k < greenBits.Length; k++)
                    imageBuilder.Add(greenBits[k]);

                for (int k = 0; k < blueBits.Length; k++)
                    imageBuilder.Add(blueBits[k]);
            }
        }

        compressedImage.image = Conversions.ToBitArray(imageBuilder);

        return compressedImage;
    }
```

The compress function basically uses the previous to functions to build the tree and get the encodes ($O(Nlog(N) + H \cdot W)$ then it prepares to create the compressed image (which is basically a struct that contains the 3 Huffman trees, the length, the width, and the BitArray of the compressed image), it starts by initializing a list named imageBuilder to push all the bit values (the list is initially set with capacity $H \cdot W$ since that's the minimum length of the final BitArray would be $H \cdot W$ thus lowering the number of possible expansions of the underlying List $\big(O(H \cdot W)\big)$, then we start iterating over all pixels $\big(O(H \cdot W)\big)$, getting the encoded value of its 3 channels via the hashmap $\big(O(1)\big)$ and appending all the bit values of all the 3 channels all together in the imageBuilder ($O\big(k_{ij}\big)$ where k_{ij} indicates the encode size of the pixel indexed with i and j), then the imageBuilder list is converted into a BitArray via a function which basically iterates over the List and copies the values into a BitArray $\Big(O\big(H \cdot W \cdot k_{ij}\big)\Big)$. Thus the overall complexity of the function is $\boldsymbol{O\left(H \cdot W \cdot k_{ij} + Nlog(N)\right).}$

```
public static RGBPixel[,] DecompressImage(ref CompressedImage image)
{

    RGBPixel[,] ret = new RGBPixel[image.length, image.width];


    int idx = 0;
    for (int i = 0; i < image.length; i++)
    {
        for (int j = 0; j < image.width; j++)
        {
            HuffmanNode it = image.redTree.root;

            while (it.left != null && it.right != null)
            {
                it = (image.image[idx]) ? it.right : it.left;
                idx++;
            }
            ret[i, j].red = it.color;

            it = image.greenTree.root;
            while (it.left != null && it.right != null)
            {
                it = (image.image[idx]) ? it.right : it.left;
                idx++;
            }
            ret[i, j].green = it.color;

            it = image.blueTree.root;
            while (it.left != null && it.right != null)
            {
                it = (image.image[idx]) ? it.right : it.left;
                idx++;
            }
            ret[i, j].blue = it.color;
        }
    }
    return ret;
}
```

The DecompressImage function first initializes an image with the proper size and an index to use for the image BitArray, then it starts iterating over all the pixels of the image $\left(O(H \cdot W)\right)$ and for each pixel it starts using the bit values in the BitArray to first traverse the red channel Huffman tree where if the bit is 1 it traverses right, and if it's 0 it traverses left, it does that till it finds a leaf (a node with no left or right), then it sets the red channel of that pixel with the value in the node, and it does that for the other 2 channels as well (green and blue) $\left(O\left(k_{ij}\right)\right)$, then the loop moves to the next pixel repeating the same algorithm till the end of the picture (thus any trailing zeros at the end are ignored). Thus, the total complexity of the algorithm is $\boldsymbol{O\left(H \cdot W \cdot k_{ij}\right)}$

Test Name	Ratio (without encryption)	Ratio (with encryption)
Easy 1	81.8%	102.7%
Easy 2	25.2%	25.2
Mid 1	534%	626.7%
Mid 2	183.6%	259.9%
Large 1	73.3%	94.3%
Large 2	90.8%	90.8%