Image Encryption & Compression Project

TA:Mohamed Magdy

Team Members :

- **Bishoy Sedra Saber 2021170130**
- **Maya Mohamed Mohamed 2021170439**
- **Farah Moataz Mohamed 2021170393**
- **Mazen Mostafa Hanafy 2021170438**
- **Moamen Mohamed Ahmed 2021170418**
- **Mohamed Ibrahim Gamal 2021170445**

1. Code of Xor Helper function :

```
// function to do XORing between two characters
// O(1) as it is a constant time operation
1 reference
public static char XOR(char bit_1, char bit_2) // O(1)
{
    return (bit_1 == bit_2) ? '0' : '1'; // O(1)
}
```

2. Code of GetKbitsLFSR function:

```
// function to get the k-bit shift register for each color channel
// O(k * 3) where k here is always 8, so O(24) = O(1)
1 reference
public static string[] GetKbitSLFSR(string initialSeed, int tapPosition, int k)
{
    // 0 == > red, 1 ==> green, 2 ==> blue
    string[] results = new string[3]; // O(1)

    // create string builder to store the answer
    StringBuilder answer = new StringBuilder(); // O(1)
    StringBuilder seedBuilder = new StringBuilder(initialSeed); // O(1)

    // convert the initial seed to binary
    int size = initialSeed.Length; // O(1)
    char tapPositionBit; // O(1)
    char firstBit; // O(1)
    int cnt = 0; // O(1)
    for (int i = 1; i <= k * 3; i++) // O(k * 3) where k here is always 8, so O(24) = O(1)
    {

        tapPositionBit = seedBuilder[size - tapPosition - 1]; // O(1)
        firstBit = seedBuilder[0]; // O(1)
        char newBit = XOR(firstBit, tapPositionBit); // O(1)

        answer.Append(newBit); // O(1)

        if (i % k == 0) // O(1)
        {
            results[cnt] = answer.ToString(); // O(1)
            answer.Clear(); // O(1)
            cnt++; // O(1)
        }

        // shifting the bits to the right
        seedBuilder.Remove(0, 1); // O(1)
        seedBuilder.Append(newBit); // O(1)

        // printing the new seed
        //Console.Write(i + 1 + " ==> ");
        //Console.WriteLine(initialSeed);
    }

    seedValue = seedBuilder.ToString(); // O(1)

    return results; // O(1)
}
```

3. Code of AlphaNumeric bonus function :

```
// [BONUS] function to convert the string to binary
// O(n) where n is the length of the string
1 reference
public static string AlphanumericConvertion(string data)
{
    //string binary = string.Empty;
    StringBuilder binary = new StringBuilder(); // O(1)
    int data_size = data.Length; // O(1)

    // if it has 0's or 1's in data string, store it in binary
    foreach (char c in data) // O(n) where n is the length of the string
    {
        if (c == '0' || c == '1')
        {
            binary.Append(c); // O(1)
        }
    }

    int binary_size = binary.Length; // O(1)

    if (binary_size == data_size)
    {
        return binary.ToString(); // O(1)
    }

    //string converted_result = String.Empty;
    StringBuilder converted_result = new StringBuilder(); // O(1)
    for (int i = 0; i < data_size; i++) // O(n) where n is the length of the string
    {
        // convert the character to binary
        string binary_char = Convert.ToString(data[i], 2); // O(log(n))
        //Console.WriteLine("Character: " + data[i] + " Binary: " + binary_char);
        //converted_result += binary_char;
        converted_result.Append(binary_char); // O(1)
    }

    return converted_result.ToString(); // O(1)
}
```

4. Code of Encrypt&Decrypt function :

```
// function to encrypt the image using the LFSR algorithm
// O(n * m) where n is the height of the image and m is the width of the image
7 references
public static RGBPixel[,] EncryptDecryptImage(RGBPixel[,] imageMatrix, string initialSeed, int tapPosition)
{
    // convert the initial seed to binary
    initialSeed = BONUS_Functions.AlphanumericConvertion(initialSeed); // O(n)

    // Generate LFSR keys for encryption
    seedValue = initialSeed;  // O(1)
    seedKey = tapPosition;   // O(1)

    int Height = GetHeight(imageMatrix); // O(1)
    int Width = GetWidth(imageMatrix);  // O(1)
    string[] keys; // O(1)

    // Ensure that the dimensions of the encrypted image match the dimensions of the original image
    RGBPixel[,] encryptedImage = new RGBPixel[Height, Width]; // O(1)

    // iterate over the image matrix and encrypt each pixel
    // O(n * m) where n is the height of the image and m is the width of the image
    for (int i = 0; i < Height; i++) // O(n)
    {
        for (int j = 0; j < Width; j++) // O(m)
        {
            // get the LFSR keys for each color channel
            keys = GetKbitSLFSR(seedValue, tapPosition, 8); // O(1)

            // get the RGB values of the pixel
            byte red = imageMatrix[i, j].red; // O(1)
            byte green = imageMatrix[i, j].green; // O(1)
            byte blue = imageMatrix[i, j].blue; // O(1)

            // convert the RGB values to binary strings
            byte redBinaryByte = Convert.ToByte(keys[0], 2); // O(1)
            byte greenBinaryByte = Convert.ToByte(keys[1], 2); // O(1)
            byte blueBinaryByte = Convert.ToByte(keys[2], 2); // O(1)

            byte encryptedRedByte = (byte)(red ^ redBinaryByte); // O(1)
            byte encryptedGreenByte = (byte)(green ^ greenBinaryByte); // O(1)
            byte encryptedBlueByte = (byte)(blue ^ blueBinaryByte); // O(1)

            // update the encrypted image matrix with the encrypted pixel
            encryptedImage[i, j].red = encryptedRedByte; // O(1)
            encryptedImage[i, j].green = encryptedGreenByte; // O(1)
            encryptedImage[i, j].blue = encryptedBlueByte;  // O(1)
        }
    }

    return encryptedImage; // O(1)
}
```

5. Code of Save Image function :

```
// function to export the image to a file
// O(n * m) where n is the height of the image and m is the width of the image
3 references
public static void SaveImage(RGBPixel[,] ImageMatrix, string FilePath)
{
    int Height = GetHeight(ImageMatrix); // O(1)
    int Width = GetWidth(ImageMatrix); // O(1)

    Bitmap ImageBMP = new Bitmap(Width, Height, PixelFormat.Format24bppRgb); // O(1)

    unsafe
    {
        BitmapData bmd = ImageBMP.LockBits(new Rectangle(0, 0, Width, Height), ImageLockMode.ReadWrite, ImageBMP.PixelFormat); // O(1)
        int nWidth = 0; // O(1)
        nWidth = Width * 3; // O(1)
        int nOffset = bmd.Stride - nWidth; // O(1)
        byte* p = (byte*)bmd.Scan0; // O(1)
        // iterate over the image matrix and write the pixel values to the image
        // O(n * m) where n is the height of the image and m is the width of the image
        for (int i = 0; i < Height; i++) // O(n)
        {
            for (int j = 0; j < Width; j++) // O(m)
            {
                p[2] = ImageMatrix[i, j].red; // O(1)
                p[1] = ImageMatrix[i, j].green; // O(1)
                p[0] = ImageMatrix[i, j].blue; // O(1)
                p += 3;     // O(1)
            }

            p += nOffset; // O(1)
        }
        ImageBMP.UnlockBits(bmd); // O(1)
    }

    // save the image to the file
    ImageBMP.Save(FilePath); // O(1)
}
```

6. Code of WriteHuffmanDict function :
Base case : theta(1), Non-recursive code: theta(1). T(N) = 2T(N-1) + theta(1).

```
// helper function to write the huffman tree to a file
// O(n) where n is the number of nodes in the huffman tree
// T(n) = 2T(n-1) + O(1) => O(n)
5 references
public static void WriteHuffmanDict(HuffmanNode root, string s, Dictionary<int, string> dict, ref long Total_Bits)
{
    // Assign 0 to the left node and recur
    if (root.Left != null)
    {
        s += '0'; // O(1)
        //                    arr[top] = 0;
        WriteHuffmanDict(root.Left, s, dict, ref Total_Bits);

        // backtracking
        s = s.Remove(s.Length - 1); // O(1)
    }

    // Assign 1 to the right node and recur
    if (root.Right != null)
    {
        s += '1'; // O(1)
        //                    arr[top] = 1;
        WriteHuffmanDict(root.Right, s, dict, ref Total_Bits);

        // backtracking
        s = s.Remove(s.Length - 1); // O(1)
    }

    // base case: if the node is a leaf node
    // O(1)
    if (root.Left == null && root.Right == null)
    {
        dict.Add(root.Pixel, s); // O(1)

        int bittat = s.Length * root.Frequency; // O(1)

        Total_Bits += bittat; // O(1)

    }
}
```

7. Code of Compress Image function :

```
// function to compress the image using huffman encoding
// O(n*m + clog(c)) where n is the height of the image, m is the width of the image, and c is the number of colors
2 references
public static KeyValuePair<long,double> CompressImage(RGBPixel[,] ImageMatrix, int tapPosition, string seedValue)
{
    // get the height and width of the image
    int Height = GetHeight(ImageMatrix); // O(1)
    int Width = GetWidth(ImageMatrix); // O(1)

    // initializing the frequency of each color bit
    int[] redFreq = new int[256]; // O(1)
    int[] blueFreq = new int[256]; // O(1)
    int[] greenFreq = new int[256]; // O(1)

    // calculate the frequency of each color bit
    // O(n * m) where n is the height of the image and m is the width of the image
    for (int i = 0; i < Height; i++)    // O(n)
    {
        for (int j = 0; j < Width; j++) // O(m)
        {
            redFreq[ImageMatrix[i, j].red]++; // O(1)
            blueFreq[ImageMatrix[i, j].blue]++; // O(1)
            greenFreq[ImageMatrix[i, j].green]++; // O(1)
        }
    }

    // 3 priority queues for each color
    PriorityQueue pq_red = new PriorityQueue(); // O(1)
    PriorityQueue pq_green = new PriorityQueue(); // O(1)
    PriorityQueue pq_blue = new PriorityQueue(); // O(1)

    // iterate over the frequency arrays and insert the non-zero frequencies into the priority queue
    // O(256 * log n) = O(log(n))
    for (int i = 0; i < 256; i++) // O(256) = O(1)
    {
        // ================= RED CHANNEL =================
        if (redFreq[i] != 0)
        {
            HuffmanNode node = new HuffmanNode // O(1)
            {
                Pixel = i, // O(1)
```

```
                Frequency = redFreq[i] // O(1)
            };

            node.Left = node.Right = null; // O(1)
            pq_red.Push(node); // O(log n)
        }
        // ================= END RED CHANNEL =================


        // ================= BLUE CHANNEL =================
        if (blueFreq[i] != 0)
        {
            HuffmanNode node = new HuffmanNode // O(1)
            {
                Pixel = i, // O(1)
                Frequency = blueFreq[i] // O(1)
            };

            node.Left = node.Right = null; // O(1)
            pq_blue.Push(node); // O(log n)
        }
        // ================= END BLUE CHANNEL =================

        // ================= GREEN CHANNEL =================
        if (greenFreq[i] != 0)
        {
            HuffmanNode node = new HuffmanNode // O(1)
            {
                Pixel = i, // O(1)
                Frequency = greenFreq[i] // O(1)
            };

            node.Left = node.Right = null; // O(1)
            pq_green.Push(node); // O(log n)
        }
        // ================= END GREEN CHANNEL =================
    }

    // construct the huffman tree for the red channel
    // O(c * log c) where c is the number of nodes in the huffman tree
    while (pq_red.Count != 1) // O(c)
    {
```

```
        HuffmanNode node = new HuffmanNode(); // O(1)
        HuffmanNode smallFreq = pq_red.Pop(); // O(log c)
        HuffmanNode largeFreq = pq_red.Pop(); // O(log c)

        node.Frequency = smallFreq.Frequency + largeFreq.Frequency; // O(1)
        node.Left = largeFreq; // O(1)
        node.Right = smallFreq; // O(1)
        pq_red.Push(node); // O(log c)
    }

    // construct the huffman tree for the green channel
    // O(c * log c) where c is the number of nodes in the huffman tree
    while (pq_green.Count != 1) // O(c)
    {
        HuffmanNode node = new HuffmanNode(); // O(1)
        HuffmanNode smallFreq = pq_green.Pop(); // O(log c)
        HuffmanNode largeFreq = pq_green.Pop(); // O(log c)

        node.Frequency = smallFreq.Frequency + largeFreq.Frequency; // O(1)
        node.Left = largeFreq; // O(1)
        node.Right = smallFreq; // O(1)
        pq_green.Push(node); // O(log c)
    }

    // construct the huffman tree for the blue channel
    // O(c * log c) where c is the number of nodes in the huffman tree
    while (pq_blue.Count != 1) // O(c)
    {
        HuffmanNode node = new HuffmanNode(); // O(1)
        HuffmanNode smallFreq = pq_blue.Pop(); // O(log c)
        HuffmanNode largeFreq = pq_blue.Pop(); // O(log c)

        node.Frequency = smallFreq.Frequency + largeFreq.Frequency; // O(1)
        node.Left = largeFreq; // O(1)
        node.Right = smallFreq; // O(1)
        pq_blue.Push(node); // O(log c)
    }

    // get the root node of the huffman tree for each channel
    HuffmanNode theRootNode = pq_red.Pop(); // O(log n)
    HuffmanNode theRootNode2 = pq_blue.Pop(); // O(log n)
    HuffmanNode theRootNode3 = pq_green.Pop();  // O(log n)
```

```
    long red_total_bits = 0; // O(1)
    long green_total_bits = 0; // O(1)
    long blue_total_bits = 0; // O(1)

    // dictionaries to store the huffman representation of each color bit
    Dictionary<int, string> red_dict = new Dictionary<int, string>(); // O(1)
    Dictionary<int, string> blue_dict = new Dictionary<int, string>(); // O(1)
    Dictionary<int, string> green_dict = new Dictionary<int, string>(); // O(1)

    // stream writer to write the huffman tree to file
    //StreamWriter stream = new StreamWriter(CompressionPath);

    // write the initial seed and tap position to the file
    //stream.WriteLine("Initial Seed: " + seedValue);
    //stream.WriteLine("Tap Position: " + tapPosition);

    // write the huffman tree to a file with red channel
    //            stream.WriteLine("Red - Frequency - Huffman Representation - Total Bits");
    //WriteHuffmanDict(theRootNode, null, red_dict, ref red_total_bits, stream);
    WriteHuffmanDict(theRootNode, null, red_dict, ref red_total_bits); // O(c)

    // write the huffman tree to a file with blue channel
    //            stream.WriteLine("Blue - Frequency - Huffman Representation - Total Bits");
    //WriteHuffmanDict(theRootNode2, null, blue_dict, ref blue_total_bits, stream);
    WriteHuffmanDict(theRootNode2, null, blue_dict, ref blue_total_bits); // O(c)

    // write the huffman tree to a file with green channel
    //            stream.WriteLine("Green - Frequency - Huffman Representation - Total Bits");
    //WriteHuffmanDict(theRootNode3, null, green_dict, ref green_total_bits, stream);
    WriteHuffmanDict(theRootNode3, null, green_dict, ref green_total_bits); // O(c)

    // calculate the total bytes of the image for each channel

    // red channel
    long red_rem = (red_total_bits % 8); // O(1)
    // if there are remaining bits, add an extra byte
    // O(1)
    long red_bytes; // O(1)
    if (red_rem != 0) {
        red_total_bits += 8; // O(1)
        red_bytes = red_total_bits / 8; // O(1)
        red_total_bits -= 8; // O(1)
```

```
        red_total_bits -= 8; // O(1)
    }
    else {
        red_bytes = red_total_bits / 8; // O(1)
    }



    // green channel
    long green_rem = (green_total_bits % 8); // O(1)
    // if there are remaining bits, add an extra byte
    // O(1)
    long green_bytes; // O(1)
    if (green_rem != 0) {
        green_total_bits += 8; // O(1)
        green_bytes = green_total_bits / 8;
        green_total_bits -= 8; // O(1)
    } else {
        green_bytes = green_total_bits / 8;
    }


    // blue channel
    long blue_rem = (blue_total_bits % 8); // O(1)
    // if there are remaining bits, add an extra byte
    // O(1)
    long blue_bytes;
    if (blue_rem != 0) {
        blue_total_bits += 8; // O(1)
        blue_bytes = blue_total_bits / 8;
        blue_total_bits -= 8;
    } else {
        blue_bytes = blue_total_bits / 8;
    }

    // calculate the total bytes of the image
    long total_bytes = red_bytes + blue_bytes + green_bytes; // O(1)

    // write the total bytes of the image
    //stream.WriteLine("total bytes: " + total_bytes);
    // write the compression ratio of the image
    long total_bits = red_total_bits + blue_total_bits + green_total_bits; // O(1)

    // product by 24 for the 3 channels (red, green, blue) and each channel has 8 bits (1 byte)
    long image_size = Height * Width * 24;
    double compression_ratio = (double)total_bits / image_size; // O(1)
    compression_ratio *= 100; // to get the percentage
    //stream.WriteLine("Compression Ratio: " + compression_ratio * 100 + "%");

    // close the stream writer
    //stream.Close();


    // law fy remainder ehgez bytes + 1

    // byte array to store the binary representation of the image of size total_bytes for each channel
    byte[] redBinaryRepresentationToWriteInFile = new byte[red_bytes]; // O(1)
    byte[] blueBinaryRepresentationToWriteInFile = new byte[blue_bytes]; // O(1)
    byte[] greenBinaryRepresentationToWriteInFile = new byte[green_bytes]; // O(1)

    // variables to store the remaining bits in the byte
    int byte_remainder1 = 8; //O(1)
    int byte_remainder2 = 8; //O(1)
    int byte_remainder3 = 8; //O(1)

    // variables to store the index of the byte array
    int redIndex = 0; //O(1)
    int blueIndex = 0; //O(1)
    int greenIndex = 0; //O(1)

    // variables to store the huffman representation of the pixel
    string huffman_string, huffman_substr;
    int huffman_string_length;
    for (int i = 0; i < Height; i++) // O(n)
    {
        for (int j = 0; j < Width; j++) // O(m)
        {

            // ================= RED CHANNEL =================
            // get the huffman representation of the pixel
            huffman_string = red_dict[ImageMatrix[i, j].red]; // O(1)
```

```
huffman_string_length = huffman_string.Length; // O(1)
// if the length of the huffman representation is less than the remaining bits in the byte
if (huffman_string_length < byte_remainder1)
{
    redBinaryRepresentationToWriteInFile[redIndex] <<= huffman_string_length;  // O(1)
    redBinaryRepresentationToWriteInFile[redIndex] += Convert.ToByte(huffman_string, 2);  // O(1)
    byte_remainder1 -= huffman_string_length;  // O(1)
}
else if (huffman_string_length == byte_remainder1)
{
    redBinaryRepresentationToWriteInFile[redIndex] <<= huffman_string_length; // O(1)
    redBinaryRepresentationToWriteInFile[redIndex] += Convert.ToByte(huffman_string, 2); // O(1)
    redIndex++; // O(1)
    byte_remainder1 = 8; // O(1)
}
else
{
    huffman_substr = huffman_string.Substring(0, byte_remainder1); // O(1)
    redBinaryRepresentationToWriteInFile[redIndex] <<= byte_remainder1; // O(1)
    redBinaryRepresentationToWriteInFile[redIndex] += Convert.ToByte(huffman_substr, 2); // O(1)
    redIndex++; // O(1)
    huffman_string = huffman_string.Substring(byte_remainder1, huffman_string.Length - byte_remainder1); // O(1)

    // iterate over the huffman representation and store the binary representation in the byte array
    // O(n) where n is the length of the huffman representation
    while (huffman_string.Length >= 8)
    {
        huffman_substr = huffman_string.Substring(0, 8); // O(1)
        redBinaryRepresentationToWriteInFile[redIndex] <<= 8; // O(1)
        redBinaryRepresentationToWriteInFile[redIndex] += Convert.ToByte(huffman_substr, 2); // O(1)
        redIndex++; // O(1)
        huffman_string = huffman_string.Substring(8, huffman_string.Length - 8); // O(1)
    }

    if (huffman_string.Length != 0)
    {
        redBinaryRepresentationToWriteInFile[redIndex] <<= huffman_string.Length; // O(1)
        redBinaryRepresentationToWriteInFile[redIndex] += Convert.ToByte(huffman_string, 2); // O(1)
        byte_remainder1 = 8 - huffman_string.Length; // O(1)
    }
    else
    {
        byte_remainder1 = 8; // O(1)
    }
}
// ================= END RED CHANNEL =================

// ================= BLUE CHANNEL =================
huffman_string = blue_dict[ImageMatrix[i, j].blue]; // O(1)
huffman_string_length = huffman_string.Length; // O(1)
if (huffman_string_length < byte_remainder2)
{
    blueBinaryRepresentationToWriteInFile[blueIndex] <<= huffman_string_length; // O(1)
    blueBinaryRepresentationToWriteInFile[blueIndex] += Convert.ToByte(huffman_string, 2); // O(1)
    byte_remainder2 -= huffman_string_length; // O(1)
}
else if (huffman_string_length == byte_remainder2)
{
    blueBinaryRepresentationToWriteInFile[blueIndex] <<= huffman_string_length; // O(1)
    blueBinaryRepresentationToWriteInFile[blueIndex] += Convert.ToByte(huffman_string, 2); // O(1)
    blueIndex++; // O(1)
    byte_remainder2 = 8; // O(1)
}
else
{
    huffman_substr = huffman_string.Substring(0, byte_remainder2); // O(1)
    blueBinaryRepresentationToWriteInFile[blueIndex] <<= byte_remainder2; // O(1)
    blueBinaryRepresentationToWriteInFile[blueIndex] += Convert.ToByte(huffman_substr, 2);  // O(1)
    blueIndex++; // O(1)
    huffman_string = huffman_string.Substring(byte_remainder2, huffman_string_length - byte_remainder2); // O(1)

    while (huffman_string.Length >= 8)
    {
        huffman_substr = huffman_string.Substring(0, 8); // O(1)
        blueBinaryRepresentationToWriteInFile[blueIndex] <<= 8; // O(1)
        blueBinaryRepresentationToWriteInFile[blueIndex] += Convert.ToByte(huffman_substr, 2); // O(1)
        blueIndex++; // O(1)
        huffman_string = huffman_string.Substring(8, huffman_string.Length - 8); // O(1)
    }
    if (huffman_string.Length != 0)
    {
```

```
            {
                blueBinaryRepresentationToWriteInFile[blueIndex] <<= huffman_string.Length; // O(1)
                blueBinaryRepresentationToWriteInFile[blueIndex] += Convert.ToByte(huffman_string, 2); // O(1)
                byte_remainder2 = 8 - huffman_string.Length; // O(1)
            }
            else {
                byte_remainder2 = 8; // O(1)
            }
        }
        // ================= END BLUE CHANNEL =================

        // ================= GREEN CHANNEL ====================
        huffman_string = green_dict[ImageMatrix[i, j].green]; // O(1)
        huffman_string_length = huffman_string.Length; // O(1)
        if (huffman_string_length < byte_remainder3)
        {
            greenBinaryRepresentationToWriteInFile[greenIndex] <<= huffman_string_length;  // O(1)
            greenBinaryRepresentationToWriteInFile[greenIndex] += Convert.ToByte(huffman_string, 2);  // O(1)
            byte_remainder3 -= huffman_string_length; // O(1)
        }
        else if (huffman_string_length == byte_remainder3)
        {
            greenBinaryRepresentationToWriteInFile[greenIndex] <<= huffman_string_length;  // O(1)
            greenBinaryRepresentationToWriteInFile[greenIndex] += Convert.ToByte(huffman_string, 2); // O(1)
            greenIndex++;  // O(1)
            byte_remainder3 = 8;  // O(1)
        }
        else
        {
            huffman_substr = huffman_string.Substring(0, byte_remainder3); // O(1)
            greenBinaryRepresentationToWriteInFile[greenIndex] <<= byte_remainder3; // O(1)
            greenBinaryRepresentationToWriteInFile[greenIndex] += Convert.ToByte(huffman_substr, 2); // O(1)
            greenIndex++; // O(1)
            huffman_string = huffman_string.Substring(byte_remainder3, huffman_string_length - byte_remainder3);// O(1)

            while (huffman_string.Length >= 8)
            {
                huffman_substr = huffman_string.Substring(0, 8); // O(1)
                greenBinaryRepresentationToWriteInFile[greenIndex] <<= 8; // O(1)
                greenBinaryRepresentationToWriteInFile[greenIndex] += Convert.ToByte(huffman_substr, 2); // O(1)
                greenIndex++; // O(1)
                huffman_string = huffman_string.Substring(8, huffman_string.Length - 8); // O(1)
```

```
            }

            if (huffman_string.Length != 0)
            {
                greenBinaryRepresentationToWriteInFile[greenIndex] <<= huffman_string.Length; // O(1)
                greenBinaryRepresentationToWriteInFile[greenIndex] += Convert.ToByte(huffman_string, 2); // O(1)
                byte_remainder3 = 8 - huffman_string.Length; // O(1)
            }
            else
            {
                byte_remainder3 = 8; // O(1)
            }
        }
        // ================== END GREEN CHANNEL ==================
    }
}

byte[] redFreqByteArr = new byte[1024]; // O(1)
byte[] greenFreqByteArr = new byte[1024]; // O(1)
byte[] blueFreqByteArr = new byte[1024]; // O(1)

// convert the frequencies to byte array
// O(256) = O(1)
for (int i = 0; i < 256; i++) // O(256) = O(1)
{
    Array.Copy(BitConverter.GetBytes(redFreq[i]), 0, redFreqByteArr, i * 4, 4); // O(1)
    Array.Copy(BitConverter.GetBytes(greenFreq[i]), 0, greenFreqByteArr, i * 4, 4); // O(1)
    Array.Copy(BitConverter.GetBytes(blueFreq[i]), 0, blueFreqByteArr, i * 4, 4); // O(1)
}

//FileStream ffs = new FileStream(compressedImageDataPath, FileMode.Truncate); // O(1)
//StreamWriter ffss = new StreamWriter(ffs); // O(1)

global_red_bytes = red_bytes; // O(1)
global_green_bytes = green_bytes; // O(1)
global_blue_bytes = blue_bytes; // O(1)

//ffss.WriteLine(red_bytes); // O(1)
//ffss.WriteLine(green_bytes); // O(1)
//ffss.WriteLine(blue_bytes); // O(1)

//ffss.WriteLine(red_rem); // O(1)
```

```
    //ffss.WriteLine(red_rem); // O(1)
    //ffss.WriteLine(green_rem); // O(1)
    //ffss.WriteLine(blue_rem); // O(1)

    //ffss.Close(); // O(1)
    //ffs.Close(); // O(1)

    FileStream ss = new FileStream(BinaryWriterPath, FileMode.Truncate); // O(1)
    BinaryWriter binWriter = new BinaryWriter(ss); // O(1)


    binWriter.Write(redFreqByteArr); // O(1)
    binWriter.Write(greenFreqByteArr); // O(1)
    binWriter.Write(blueFreqByteArr); // O(1)

    binWriter.Write(redBinaryRepresentationToWriteInFile); // O(1)
    binWriter.Write(greenBinaryRepresentationToWriteInFile); // O(1)
    binWriter.Write(blueBinaryRepresentationToWriteInFile); // O(1)

    binWriter.Write(seedValue); // O(1)
    binWriter.Write(tapPosition); // O(1)

    binWriter.Write(Width); // O(1)
    binWriter.Write(Height); // O(1)

    binWriter.Close(); // O(1)
    ss.Close(); // O(1)

    KeyValuePair<long, double> result = new KeyValuePair<long, double>(total_bytes, compression_ratio); // O(1)

    return result; // O(1)
}
```

- **Freq array nested loop theta(n*m)**
- **construct tree: theta(clogc)**
- **traverse tree: theta(n) where n is number of nodes in tree**
- **store Huffman representation in byte array: theta(n*m)**

Total: theta(n*m + clogc)

8. Code of Decompress Image function :

```
// function to decompress the image using the huffman tree and the binary file
// O(n * m) where n is the height of the image and m is the width of the image
2 references
public static RGBPixel[,] DecompressImage()
{
    // declare the binary file stream and the binary reader
    //FileStream readingStream = new FileStream(compressedImageDataPath, FileMode.Open); // O(1)
    //StreamReader stream_reader = new StreamReader(readingStream); // O(1)

    // stream carries:
    // rgb length (3 lines)
    // binary carries:
    // (1) rgb frequencies (each one 1024 byte)
    // (2) huffman representations (3) seed (4) tap position (5) width (6) height

    // lengths of rgb bytes
    //int red_length = Convert.ToInt32(stream_reader.ReadLine()); // O(1)
    //int green_length = Convert.ToInt32(stream_reader.ReadLine()); // O(1)
    //int blue_length = Convert.ToInt32(stream_reader.ReadLine()); // O(1)

    int red_length = (int)global_red_bytes; // O(1)
    int green_length = (int)global_green_bytes; // O(1)
    int blue_length = (int)global_blue_bytes; // O(1)

    //int red_extra_bits = Convert.ToInt32(stream_reader.ReadLine());
    //int green_extra_bits = Convert.ToInt32(stream_reader.ReadLine());
    //int blue_extra_bits = Convert.ToInt32(stream_reader.ReadLine());

    //stream_reader.Close(); // O(1)
    //readingStream.Close(); // O(1)

    FileStream binaryReadingStream = new FileStream(BinaryWriterPath, FileMode.Open); // O(1)
    BinaryReader binary_reader = new BinaryReader(binaryReadingStream); // O(1)

    // frequency arrs for carrying freqs that are the
    byte[] redFreqInBytes = binary_reader.ReadBytes(1024); // O(1)
    byte[] greenFreqInBytes = binary_reader.ReadBytes(1024); // O(1)
    byte[] blueFreqInBytes = binary_reader.ReadBytes(1024); // O(1)

    // rgb int arrs to store frequencies read from binary file
    int[] redFreq = new int[256]; // O(1)
    int[] greenFreq = new int[256]; // O(1)
    int[] blueFreq = new int[256]; // O(1)

    // priority queues for rgb to construct the tree
    PriorityQueue pq_red = new PriorityQueue(); // O(1)
    PriorityQueue pq_green = new PriorityQueue(); // O(1)
    PriorityQueue pq_blue = new PriorityQueue(); // O(1)

    // convert the frequencies from byte array to int array
    // O(1024) = O(1)
    for (int i = 0; i < 1024; i += 4) // O(1)
    {
        redFreq[i / 4] = BitConverter.ToInt32(redFreqInBytes, i); // O(1)
        greenFreq[i / 4] = BitConverter.ToInt32(greenFreqInBytes, i); // O(1)
        blueFreq[i / 4] = BitConverter.ToInt32(blueFreqInBytes, i); // O(1)
    }

    // iterate over the frequencies and insert the non-zero frequencies into the priority queue
    // O(256 * log n) = O(log n)
    for (int i = 0; i < 256; i++)
    {

        // ================= RED CHANNEL =================
        if (redFreq[i] != 0)
        {
            HuffmanNode node = new HuffmanNode // O(1)
            {
                Pixel = i,
                Frequency = redFreq[i]
            };
            node.Left = node.Right = null; // O(1)
            pq_red.Push(node); // O(log n)
        }
        // ================= END RED CHANNEL =================
```

```
        // ================= GREEN CHANNEL =================
        if (greenFreq[i] != 0)
        {
            HuffmanNode node = new HuffmanNode // O(1)
            {
                Pixel = i,
                Frequency = greenFreq[i]
            };

            node.Left = node.Right = null; // O(1)
            pq_green.Push(node); // O(log n)
        }
        // ================= END GREEN CHANNEL =================


        // ================= BLUE CHANNEL =================
        if (blueFreq[i] != 0)
        {
            HuffmanNode node = new HuffmanNode // O(1)
            {
                Pixel = i,
                Frequency = blueFreq[i]
            };
            node.Left = node.Right = null; // O(1)
            pq_blue.Push(node); // O(log n)
        }
        // ================= END BLUE CHANNEL =================
    }
```

```
    // construct the huffman tree for the red channel
    // O(c * log c) where c is the number of nodes in the huffman tree
    while (pq_red.Count != 1) // O(c)
    {
        HuffmanNode node = new HuffmanNode(); // O(1)
        HuffmanNode smallFreq = pq_red.Pop(); // O(log c)
        HuffmanNode largeFreq = pq_red.Pop(); // O(log c)

        node.Frequency = smallFreq.Frequency + largeFreq.Frequency; // O(1)
        node.Left = largeFreq; // O(1)
        node.Right = smallFreq; // O(1)
        pq_red.Push(node); // O(log c)
    }

    // construct the huffman tree for the green channel
    // O(c * log c) where c is the number of nodes in the huffman tree
    while (pq_green.Count != 1) // O(c)
    {
        HuffmanNode node = new HuffmanNode(); // O(1)
        HuffmanNode smallFreq = pq_green.Pop(); // O(log c)
        HuffmanNode largeFreq = pq_green.Pop(); // O(log c)
```

```
        node.Frequency = smallFreq.Frequency + largeFreq.Frequency; // O(1)
        node.Left = largeFreq; // O(1)
        node.Right = smallFreq; // O(1)
        pq_green.Push(node); // O(log c)
    }

    // huffman tree for the blue channel
    // O(C * log C) where C is the number of nodes in the huffman tree
    while (pq_blue.Count != 1)
    {
        HuffmanNode node = new HuffmanNode(); // O(1)
        HuffmanNode smallFreq = pq_blue.Pop(); // O(log c)
        HuffmanNode largeFreq = pq_blue.Pop(); // O(log c)

        node.Frequency = smallFreq.Frequency + largeFreq.Frequency; // O(1)
        node.Left = largeFreq; // O(1)
        node.Right = smallFreq; // O(1)
        pq_blue.Push(node);     // O(log c)
    }

    // extract the roots of the rgb trees
    HuffmanNode rootNodeRed = pq_red.Pop(); // theta(1)
    HuffmanNode rootNodeGreen = pq_green.Pop(); // theta(1)
    HuffmanNode rootNodeBlue = pq_blue.Pop(); // theta(1)

    //read from the file the red, green, blue bytes compressed values
    byte[] compressed_red = binary_reader.ReadBytes(red_length); // O(1)
    byte[] compressed_green = binary_reader.ReadBytes(green_length); // O(1)
    byte[] compressed_blue = binary_reader.ReadBytes(blue_length); // O(1)

    seedValue = binary_reader.ReadString(); // O(1)
    seedKey = binary_reader.ReadInt32(); // O(1)

    int Width = binary_reader.ReadInt32(); // O(1)
    int Height = binary_reader.ReadInt32(); // O(1)

    binary_reader.Close(); // O(1)
    binaryReadingStream.Close(); // O(1)
    //Tape_Position = tap_position;//o(1) save it to he global value
    //Initial_Seed = seed;//o(1) save it to the global value
```

```
    // 3 lists to save colors values that would end in 3*(N^2) space
    List<int> redPixels = new List<int>();  // O(1)
    List<int> greenPixels = new List<int>();  // O(1)
    List<int> bluePixels = new List<int>();  // O(1)

    byte byteValue = 128; // O(1)
    // 1st iter: 1000 0000, 2nd: 0100 0000, 3rd: 0010 0000, 4th: 0001 0000, 5th: 0000 1000, 6th: 0000 0100
    int currBitCount = 0; // O(1)
    HuffmanNode rootNode1 = rootNodeRed; // O(1)

    int cnt = 0; // O(1)
    long crl = compressed_red.Length; // O(1)

    // iterate over the compressed array and decompress it
    // O(n) where n is the length of the compressed array which is equal to number of pixels in image O(N*M)
    while (cnt < crl)
    {
        while (currBitCount < 8) // O(1)
        {
            byte hettaOS = (byte)(compressed_red[cnt] & byteValue);  // O(1)
            HuffmanNode tempNode; // O(1)
            if (hettaOS == 0)
            {
                tempNode = rootNode1.Left; // O(1)
            }
            else {
                tempNode = rootNode1.Right; // O(1)
            }

            if (tempNode.Left == null && tempNode.Right == null)
            {
                redPixels.Add(tempNode.Pixel); // O(1)
                rootNode1 = rootNodeRed; // O(1)
            }
            else
            {
                rootNode1 = tempNode; // O(1)
            }

            // divide to compare with the bit on the right
            byteValue /= 2; // O(1)
```

```
            currBitCount++;  // O(1)
        }

        cnt++; // O(1)
        currBitCount = 0; // O(1)
        byteValue = 128;  // O(1)
    }

    byteValue = 128; // O(1)
    currBitCount = 0; // O(1)
    rootNode1 = rootNodeGreen; // O(1)
    cnt = 0; // O(1)
    long cgl = compressed_green.Length; // O(1)

    // iterate over the compressed array and decompress it
    // O(n) where n is the length of the compressed array which is equal to number of pixels in image O(N*M)
    while (cnt < cgl)
    {
        while (currBitCount < 8) // O(1)
        {
            byte hettaOS = (byte)(compressed_green[cnt] & byteValue);  // O(1)
            HuffmanNode tempNode; // O(1)
            if (hettaOS == 0)
            {
                tempNode = rootNode1.Left; // O(1)
            }
            else {
                tempNode = rootNode1.Right; // O(1)
            }

            if (tempNode.Left == null && tempNode.Right == null)
            {
                greenPixels.Add(tempNode.Pixel); // O(1)
                rootNode1 = rootNodeGreen; // O(1)
            }
            else
            {
                rootNode1 = tempNode; // O(1)
            }
```

```
            // divide to compare with the bit on the right
            byteValue /= 2; // O(1)
            // increment counter to go compare the bit to the right
            currBitCount++; // O(1)

        }
        cnt++; // O(1)
        currBitCount = 0; // O(1)
        byteValue = 128; // O(1)
    }

    byteValue = 128; // O(1)
    currBitCount = 0; // O(1)
    rootNode1 = rootNodeBlue; // O(1)
    cnt = 0; // O(1)
    long cbl = compressed_blue.Length; // O(1)

    // iterate over the compressed array and decompress it
    // O(n) where n is the length of the compressed array which is equal to number of pixels in image O(Width*Height)
    while (cnt < cbl)
    {
        while (currBitCount < 8) // O(1)
        {
            byte hettaOS = (byte)(compressed_blue[cnt] & byteValue); // O(1)
            HuffmanNode tempNode; // O(1)
            if (hettaOS == 0)
            {
                tempNode = rootNode1.Left; // O(1)
            }
            else {
                tempNode = rootNode1.Right; // O(1)
            }

            if (tempNode.Left == null && tempNode.Right == null)
            {
                bluePixels.Add(tempNode.Pixel); // O(1)
                rootNode1 = rootNodeBlue;  // O(1)
            }
            else
            {
```

```
                rootNode1 = tempNode; // O(1)
            }

            // divide to compare with the bit on the right
            byteValue /= 2; // O(1)
            // increment counter to go compare the bit to the right
            currBitCount++; // O(1)
        }

        cnt++; // O(1)
        currBitCount = 0; // O(1)
        byteValue = 128; // O(1)
    }

    RGBPixel[,] decompressedPicture = new RGBPixel[Height, Width]; // O(1)
    int index = 0; // O(1)

    int redLength = redPixels.Count; // O(1)
    int greenLength = greenPixels.Count; // O(1)
    int blueLength = bluePixels.Count; // O(1)

    // iterate over the decompressed pixels and store them in the decompressed picture
    // O(n * m) where n is the height of the image and m is the width of the image
    for (int i = 0; i < Height; i++) // O(n)
    {
        for (int j = 0; j < Width; j++) // O(m)
        {
            if (index < redLength && index < greenLength && index < blueLength)
            {
                decompressedPicture[i, j].red = (byte)redPixels[index]; // O(1)
                decompressedPicture[i, j].green = (byte)greenPixels[index]; // O(1)
                decompressedPicture[i, j].blue = (byte)bluePixels[index]; // O(1)
            }
            index++; // O(1)
        }
    }

    return decompressedPicture; // O(1)
  }
}
```

- **Freq array nested loop theta(n*m)**
- **construct tree: theta(clogc)**
- **traverse tree: theta(n) where n is number of nodes in tree**
- **store values in rgb arrs: theta(n*m)**
- **get back original image: theta(n*m)**

9. Code of Get Combinations helper bonus function :

```
// function to get all the possible combinations of a given length
// O(2^n * n) where n is the length of the binary string
1 reference
public static string[] GetCombinations(int length)
{

    // total number of combinations
    int total = (int)Math.Pow(2, length); // O(1) or O(log(n))

    // array to store the combinations
    string[] combinations = new string[total]; // O(1)

    // iterate over all the possible combinations
    // O(2^n * n) where n is the length of the binary string
    for (int i = 0; i < total; i++)
    {
        // convert the number to binary
        string binary = Convert.ToString(i, 2); // O(log(n))

        // get the length of the binary string
        int binary_length = binary.Length; // O(1)

        // check if the length of the binary string is less than the required length
        if (binary_length < length) // O(1)
        {
            // add zeros to the left of the binary string to make it the required length
            binary = binary.PadLeft(length, '0'); // O(n)
        }

        // add the binary string to the combinations array
        combinations[i] = binary; // O(1)
    }

    // return the combinations array
    return combinations; // O(1)
}
```

10. Code of Test Identical Bonus function :

```
// function to test the identicality of two images
// O(n * m) where n is the height of the image and m is the width of the image
2 references
public static bool TestIdenticality(RGBPixel[,] ImageMatrix1, RGBPixel[,] ImageMatrix2)
{
    // get the dimensions of the two images
    int Height1 = ImageOperations.GetHeight(ImageMatrix1); // O(1)
    int Width1 = ImageOperations.GetWidth(ImageMatrix1); // O(1)

    int Height2 = ImageOperations.GetHeight(ImageMatrix2); // O(1)
    int Width2 = ImageOperations.GetWidth(ImageMatrix2); // O(1)


    // checking if the dimensions of the two images are not the same
    if (Height1 != Height2 || Width1 != Width2) // O(1)
    {
        return false;
    }

    // checking the identicality of the two images for each pixel
    // O(n * m) where n is the height of the image and m is the width of the image
    for (int i = 0; i < Height1; i++) // O(n)
    {
        for (int j = 0; j < Width1; j++)  // O(m)
        {
            if (ImageMatrix1[i, j].red != ImageMatrix2[i, j].red || ImageMatrix1[i, j].green != ImageMatrix2[i, j].green || ImageMatrix1[i, j].blue != ImageMatrix2[i, j].blue) // O(1)
            {
                return false;
            }
        }
    }

    return true;
}
```

11. Code of Attack bonus function :

```
// [BONUS] function to attack the encrypted image
// O(2^n * n * m) where n is the length of the binary string and m is the size of the image
1 reference
public static Tuple<string, int> Attack(RGBPixel[,] EncryptedImageMatrix, RGBPixel[,] DesiredImageMatrix, int Nbits) // O(2^n * n * m)
{
    // get all the possible combinations of the initial seed
    string[] combinations = GetCombinations(Nbits); // O(2^n * n)

    // iterate over all the possible combinations
    // O(2^n * n * m) where n is the length of the binary string and m is the size of the image
    foreach (string combination in combinations) // O(2^n)
    {
        // loop through all the tap positions
        for (int tapPosition = 0; tapPosition < Nbits; tapPosition++) // O(n)
        {
            // decrypt the image using the current combination and tap position
            RGBPixel[,] DecryptedImageMatrix = ImageOperations.EncryptDecryptImage(EncryptedImageMatrix, combination, tapPosition); // O(n * m)

            // check if the decrypted image is identical to the desired image
            if (TestIdenticality(DecryptedImageMatrix, DesiredImageMatrix)) // O(n * m)
            {
                // return the combination and tap position
                return new Tuple<string, int>(combination, tapPosition); // O(1)
            }
        }
    }

    // return null if no combination and tap position were found
    return null;

}
```

12. Code of Priority queue class :

```
public class PriorityQueue
{
    private List<HuffmanNode> list; // O(1)
    14 references
    public int Count { get { return list.Count; } } // O(1)
    public readonly bool IsDescending; // O(1)

    8 references
    public PriorityQueue() // O(1)
    {
        list = new List<HuffmanNode>(); // O(1)
    }

    0 references
    public PriorityQueue(bool isdesc) // O(1)
        : this()
    {
        IsDescending = isdesc; // O(1)
    }

    0 references
    public PriorityQueue(int capacity) // O(n)
        : this(capacity, false) // O(n)
    { }

    0 references
    public PriorityQueue(IEnumerable<HuffmanNode> collection) // O(n)
        : this(collection, false) // O(n)
    { }

    1 reference
    public PriorityQueue(int capacity, bool isdesc) // O(n)
    {
        list = new List<HuffmanNode>(capacity); // O(n)
        IsDescending = isdesc; // O(1)
    }
```

```
1 reference
public PriorityQueue(IEnumerable<HuffmanNode> collection, bool isdesc) // O(n)
    : this()
{
    IsDescending = isdesc; // O(1)
    // O(n log n)
    foreach (var item in collection)  // O(n)
    {
        Push(item); // O(log n)
    }
}

// O(log n)
13 references
public void Push(HuffmanNode x)
{
    list.Add(x); // O(1)
    int i = Count - 1; // O(1)

    while (i > 0) // O(log n)
    {
        int p = (i - 1) / 2; // O(1)
        if ((IsDescending ? -1 : 1) * list[p].Frequency.CompareTo(x.Frequency) <= 0) // O(1)
        {
            break; // O(1)
        }

        list[i] = list[p]; // O(1)
        i = p; // O(1)
    }

    if (Count > 0) // O(1)
    {
        list[i] = x; // O(1)
    }
}
```

```
18 references
public HuffmanNode Pop() // O(log n)
{
    HuffmanNode target = Top(); // O(1)
    HuffmanNode root = list[Count - 1]; // O(1)
    list.RemoveAt(Count - 1); // O(1)

    int i = 0; // O(1)
    while (i * 2 + 1 < Count)
    {
        int a = i * 2 + 1; // O(1)
        int b = i * 2 + 2; // O(1)
        int c = b < Count && (IsDescending ? -1 : 1) * list[b].Frequency.CompareTo(list[a].Frequency) < 0 ? b : a; // O(1)

        // O(1)
        if ((IsDescending ? -1 : 1) * list[c].Frequency.CompareTo(root.Frequency) >= 0) {
            break; // O(1)
        }

        list[i] = list[c]; // O(1)
        i = c; // O(1)
    }

    // O(1)
    if (Count > 0) {
        list[i] = root; // O(1)
    }
    return target; // O(1)
}

1 reference
public HuffmanNode Top() // O(1)
{
    // O(1)
    if (Count == 0) {
        throw new InvalidOperationException("Queue is empty."); // O(1)
    }
    return list[0]; // O(1)
}
```

```
// O(1)
0 references
public void Clear()
{
    list.Clear(); // O(1)
}
}
```

Push and Pop are of theta(log n) complexity

Compression Ratio Table:

Initial seed	Tap position	Image	With encryption	Without encryption
0011001101010101	8	Small_case1	100%	79.763%
111111110000000011111111000000001111111100000000	47	Small_case2	24.905%	24.905%
01101000010100010000	16	Medium_case1	100%	85.224%
00101	0	Medium_case2	86.412%	61.032%
00101	3	Large_case1	94.336%	73.312%
0000	3	Large_case2	90.786%	90.786%