Encryption code snippet

1 - we have a my_seed struct that contains

```
public long SEED;
public char[] s;
public int sz, TAB, pStart, pEnd, pTab;
```

2 - the struct has a constructor which initialize the struct attribute (TAB , sz ,....,pTab)

```
public my_seed(TextBox tabBox, TextBox seedBox)
{
   TAB = int.Parse(tabBox.Text.ToString());
   string temp = seedBox.Text.ToString();
   s = new char[temp.Length];
   for (int i = 0; i < temp.Length; i++)
       s[i] = temp[i];
   long ret = 0;
   for (int i = 0; i < temp.Length; i++)
       ret = ret * 2 + (temp[i] - '0');
   SEED = ret;
   sz = s.Length;
   pStart = 0;
   pEnd = sz - 1;
    pTab = TAB;
}
```

3 - the struct has a function called go_next which returns the next msk by LFSR which has two states dealing (binary , alphanumeric)

```
public byte go_next(bool f)
{
   if (f)
   {
       byte ret = 0;
       for (int i = 7; i >= 0; i--)
       {
           long x = (SEED >> (sz - 1)) & 1L;
           long y = SEED >> TAB & 1L;
```

```
            SEED *= 2;
            SEED += x ^ y;
            ret += (byte)((x ^ y) << i);
        }

        return ret;
    }
    else
    {
        byte ret = (byte)(s[pTab] ^ s[pEnd]);
        pEnd = (pEnd - 1 + sz) % sz;
        pTab = (pTab - 1 + sz) % sz;
        pStart = (pStart - 1 + sz) % sz;
        s[pStart] = (char)ret;
        byte one = 1;
        int pos = (pStart + 1) % sz;
        ret = (byte)(ret | ((s[pos] & one) << 7));
        return ret;
    }
}
```

Explanation of bonus part (dealing with alphanumeric)

Generating the next character :

- The function calculates the next pseudo-random bit by performing an exclusive OR (XOR) operation on the values of two taps in the LSFR (`s[pTab]` and `s[pEnd]`). The XOR operation returns character

Updating indices:

- The code then updates the positions of the taps (`pTab`, `pEnd`, and `pStart`) by subtracting 1 and performing a modulo operation with the length of the shift register (`sz`). This effectively shifts the contents of the register by one position.

Updating the value of seed:

- The line `s[pStart] = (char)ret;` in the provided code updates the Linear Feedback Shift Register (LSFR) with the newly generated pseudo-random bit.

Return:

- The function returns the generated pseudo-random byte.

- The same function and struct will be used in decryption

Compress and decompress code snippet

1 - we have a class to carry the node info called node

```
public class Node : IComparable<Node>
{
    public int Left, Right, Leaf, Frq;

    public Node(int left, int right, int leaf, int frq)
    {
        this.Left = left;
        this.Right = right;
        this.Leaf = leaf;
        this.Frq = frq;
    }

    public int CompareTo(Node other)
    {
        return this.Frq.CompareTo(other.Frq);
    }
}
```

2 - first we create arrays and build it

1. sz[] for the number of masks for each color
2. frq[][] for color frequency
3. newmask[] for the mask of the color frequency in Huffman tree
4. szOfnewmask[] For the size of this mask
5. huffmanTree[] to save the info of Huffman tree and we build

```
public static void CompressImage(RGBPixel[,] ImageMatrix, String fileName)
{
    int n = GetHeight(ImageMatrix), m = GetWidth(ImageMatrix);
    int[][] frq = new int[3][];
    for (int i = 0; i < 3; i++)
        frq[i] = new int[1 << 8];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
```

```
        {
            frq[0][ImageMatrix[i, j].red]++;
            frq[1][ImageMatrix[i, j].green]++;
            frq[2][ImageMatrix[i, j].blue]++;
        }
    }

    Node[][] huffmanTree = new Node[3][];
    int[] sz = new int[3];
    int[][] newMask = new int[3][];
    int[][] szOfNewMask = new int[3][];
    for (int i = 0; i < 3; i++)
    {
        newMask[i] = new int[(1 << 8)];
        szOfNewMask[i] = new int[(1 << 8)];
    }

    for (int col = 0; col < 3; col++)
    {
        for (int i = 0; i < (1 << 8); i++)
        {
            if (frq[col][i] > 0)
                sz[col]++;
        }

        huffmanTree[col] = new Node[2 * sz[col] + 10];
        int idx = 0;
        for (int i = 0; i < (1 << 8); i++)
        {
            if (frq[col][i] > 0)
            {
                huffmanTree[col][idx++] = new Node(-1, -1, i, frq[col][i]);
            }
        }

        Array.Sort<Node>(huffmanTree[col], 0, idx);

        Queue<Tuple<int, int>>[] Qt = new Queue<Tuple<int, int>>[2];
        for (int i = 0; i < 2; i++)
            Qt[i] = new Queue<Tuple<int, int>>();
        for (int i = 0; i < idx; i++)
        {
            Qt[0].Enqueue(new Tuple<int, int>(i, huffmanTree[col][i].Frq));
        }

        while (Qt[0].Count + Qt[1].Count >= 2)
        {
            Tuple<int, int>[] t = new Tuple<int, int>[2];
            for (int i = 0; i < 2; i++)
                t[i] = new Tuple<int, int>(-1, -1);
            for (int _ = 0; _ < 2; _++)
            {
```

```
                int who = 1;
                if (Qt[1].Count == 0 || (Qt[0].Count != 0 && Qt[0].First().Item2 <
Qt[1].First().Item2))
                {
                    who = 0;
                }

                for (int i = 0; i < 2; i++)
                    if (t[i].Item1 == -1)
                    {
                        t[i] = Qt[who].Dequeue();
                        break;
                    }
            }

            huffmanTree[col][idx] = new Node(t[0].Item1, t[1].Item1, -1,
t[0].Item2 + t[1].Item2);
            Qt[1].Enqueue(new Tuple<int, int>(idx, t[0].Item2 + t[1].Item2));
            idx++;
        }

        Queue<Tuple<int, int, int>> Q = new Queue<Tuple<int, int, int>>();
        Q.Enqueue(new Tuple<int, int, int>(idx - 1, 0, 0));
        while (Q.Count != 0)
        {
            Tuple<int, int, int> t = Q.Dequeue();
            int u = t.Item1, curmsk = t.Item2, depth = t.Item3;
            if (huffmanTree[col][u].Leaf != -1)
            {
                newMask[col][huffmanTree[col][u].Leaf] = curmsk;
                szOfNewMask[col][huffmanTree[col][u].Leaf] = depth;
                continue;
            }

            Q.Enqueue(new Tuple<int, int, int>(huffmanTree[col][u].Left, curmsk,
depth + 1));
            Q.Enqueue(new Tuple<int, int, int>(huffmanTree[col][u].Right, curmsk |
(1 << depth), depth + 1));
        }
    }
```

3 - we will inialize trie data structure and we will read the data from the file to build the trie and use this trie to decompress the original masks

```
public static RGBPixel[,] DecompressImage(String fileName)
```

```
{
   List<Node>[] trie = new List<Node>[3];
   for (int i = 0; i < 3; i++)
   {
       trie[i] = new List<Node>();
       trie[i].Add(new Node(-1, -1, -1, 0));
   }

   RGBPixel[,] ImageMatrix;
   using (var stream = File.Open(
              $"G:\\Project\\[1] Image Encryption and Compression\\Sample
Test\\wla\\{fileName}_Com.bin",
              FileMode.Open))
   {
       using (var reader = new BinaryReader(stream, Encoding.UTF8, false))
       {
           int n = reader.ReadInt32();
           int m = reader.ReadInt32();
           ImageMatrix = new RGBPixel[n, m];
           for (int col = 0; col < 3; col++)
           {
               int sz = reader.ReadInt32();
               for (int _ = 0; _ < sz; _++)
               {
                   int oriMask = (int)reader.ReadByte();
                   int nwmask = (int)reader.ReadByte();
                   int szOfnwmsk = (int)reader.ReadByte();
                   int curNode = 0;
                   for (int c = 0; c < szOfnwmsk; c++)
                   {
                       int ch = (nwmask >> c) & 1;
                       if (ch == 0 && trie[col][curNode].Left == -1)
                       {
                           trie[col][curNode].Left = trie[col].Count;
                           trie[col].Add(new Node(-1, -1, -1, 0));
                       }

                       if (ch == 1 && trie[col][curNode].Right == -1)
                       {
                           trie[col][curNode].Right = trie[col].Count;
                           trie[col].Add(new Node(-1, -1, -1, 0));
                       }

                       if (ch == 0)
                           curNode = trie[col][curNode].Left;
                       else curNode = trie[col][curNode].Right;
                   }

                   trie[col][curNode].Leaf = oriMask;
               }
           }
           byte b = 0;
```

```
            int idx = -1;
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < m; j++)
                {
                    ImageMatrix[i, j] = new RGBPixel();
                    for (int col = 0; col < 3; col++)
                    {
                        int curNode = 0;
                        while (trie[col][curNode].Leaf == -1)
                        {
                            int curBit = readBit(ref b, ref idx, reader);
                            if (curBit == 0)
                                curNode = trie[col][curNode].Left;
                            else
                                curNode = trie[col][curNode].Right;
                        }

                        if (col == 0)
                            ImageMatrix[i, j].red = (byte)trie[col][curNode].Leaf;
                        else if (col == 1)
                            ImageMatrix[i, j].green =
(byte)trie[col][curNode].Leaf;
                        else
                            ImageMatrix[i, j].blue =
(byte)trie[col][curNode].Leaf;
                    }
                }
            }
        }
    }

    return ImageMatrix;
}
```

Complexity analysis for encryption

1. **Constructor Initialization**:
    - Parsing tap position and seed value: O(n)
    - Converting seed to binary representation: O(len of the seed)
    - Total: O(n)
2. **Encryption/Decryption**:
    - O(1) per byte for both encryption and decryption operations

3. **Method go_next**

- O(1) per byte

Complexity Analysis of CompressImage function

This function compresses an RGB image using Huffman coding for each color channe and generate huffman tree for each colorl:

1. Frequency Table Creation (O(n * m))

- Loops iterate over each pixel in the image (`n * m` times).
- For each pixel, the corresponding color channel's frequency table is incremented (`O(1)` per pixel).

2. Huffman Tree Construction per Channel (O(k * log k))

- This part happens three times (once for each color channel).
- Finding non-zero frequencies in the frequency table (`O(k)` where k is the number of possible values (256 for each channel)).
- Creating nodes and sorting them based on frequency (`O(k log k)` using sorting algorithms like quicksort or merge sort).
- Building the Huffman tree using a two queues method (`O(k log k)`).

3. Generating Code Masks and Sizes (O(k))

- This part also happens three times (once per channel).
- Traverses the Huffman tree using a queue (`O(k)` in the worst case, where each node is visited once).
- Assigns the new mapped code masks and sizes of each new mask to leaf nodes in the tree (`O(1)` per leaf node).

Overall Complexity:

The dominant factors are the frequency table creation (`O(n * m)`) and Huffman tree construction (`O(3 * k * log k)`).Since `n * m` is typically much larger than `k` as k can be maximum of 256, the overall complexity can be approximated as: O(n * m) + O(3 * k * log k)

Complexity Analysis of decompressImage function

1. Trie Initialization:
    - Initializing each trie tree(prefix tree) list and adding a node: O(sz of the new mapped mask) per channel (constant time complexity).
    - Total: O(sum of sizes of the new mapped masks) wich can be at maximum 256 * 8 for each color.
2. Reading Compressed Data:
    - Reading image dimensions (n and m): O(1) (constant time complexity).
    - Reading size of distinct masks for each color channel: O(1) (constant time complexity).
    - Parsing compressed data and building trie:
        - For each color channel (red, green, blue):
            - Reading each compressed data entry: O(sz) where sz is the size of the compressed data for the specific channel.
            - Building the trie: O(sz) in the worst case, where sz is the size of the compressed data.
        - Total: O(sz)
3. Reconstructing Image:
    - Iterating through each pixel (n * m):
        - For each pixel:
            - Traversing the trie to find the original pixel value: O(sz of the color mask for the pixel wich can be at maximum 8) per color channel.
    - Total: O(n * m)

Tests cases

Test (Small case 1):

- **Without encryption:** 65,384 / 73,622
- **With encryption:** 82,562 / 73,622

Test (Small case 2):

- **Without encryption:** 196,027 / 750,138

- **With encryption:** 196, 027/ 750,138

Test (Medium case 1):

- **Without encryption:** 8,050,431 / 1,506,250
- **With encryption:** 9,199,981 / 1,506,250

Test (Medium case 2):

- **Without encryption:** 9,199,891 / 8,274,238
- **With encryption:** 21,511,344 / 8,274,238

Test (Large case 1):

- **Without encryption:** 97,050,715 / 132,385,526
- **With encryption:** 124,885,874/ 132,385,526

Test (Large case 2):

- **Without encryption:** 195,192,447 / 214,990,902
- **With encryption:** 195,192,447 / 214,990,902