# TEAM MEMBERS :-

# Ahmed Mostafa El Moslemany.

Section 3.

ID:- 2014170061

* **Ahmed Naser Abdel Baky Khalaf**.

Section 3.

ID:- 2015170054

# Ahmed Kamal El Din El Sayed Abdel Rahman.

Section 2.

ID:- 2015170037

* **Ahmed Hamdy Abdel Fattah Ibrahim.**

Section 1.

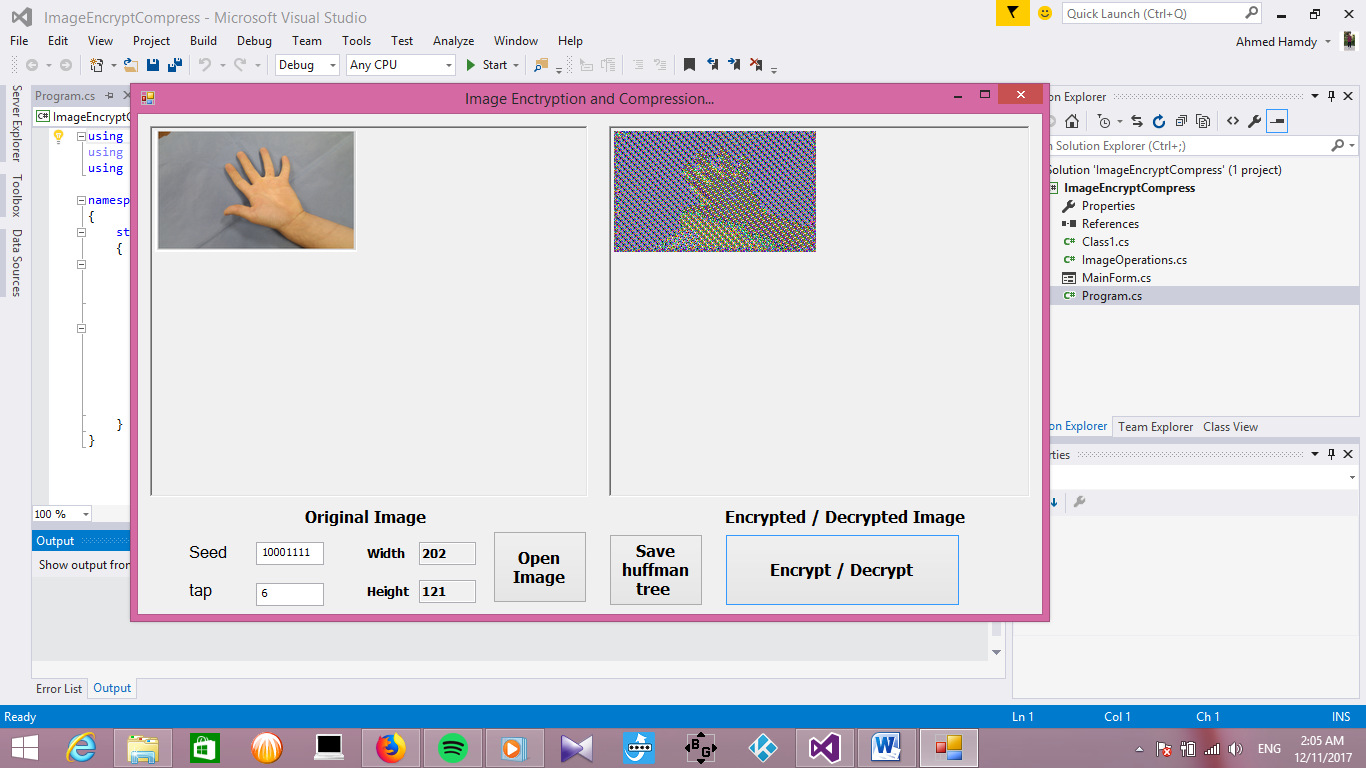
ID:- 2014170021

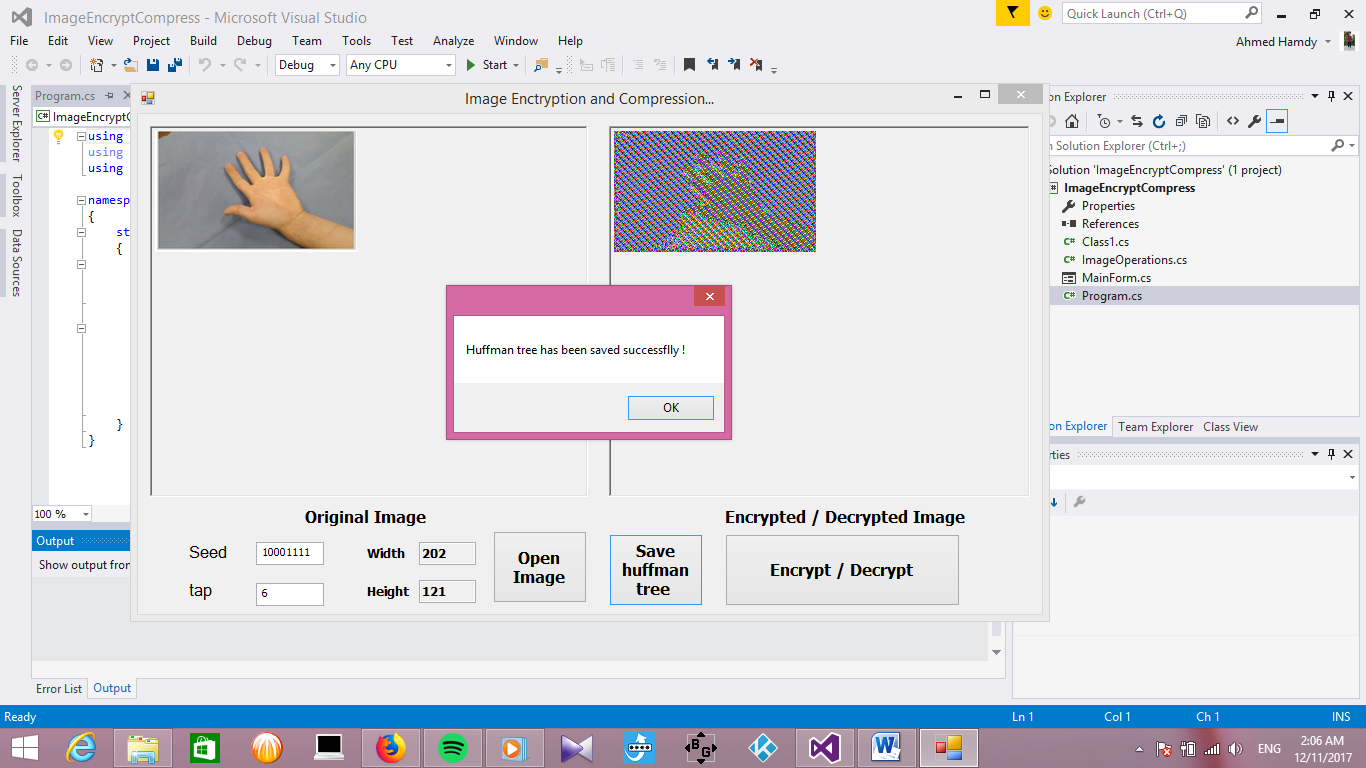
* **Ahmed Essmat Ahmed Mohammed.**

Section 2.

ID:- 2015170032

# IMAGE ENCRYPTION PROJECT





# Used Data Structure : Priority Queue :

- Priority Queue is an extension of queue but with added some properties like :  
1) Every item has a specific priority.  
2) An element with high priority is dequeued before an element with low priority.  
3) If two elements have the same priority we deal with them according to their order in the queue.

- A typical priority queue supports some operations like enqueue & dequeue ,etc…

- When a priority queue is implemented using a heap, the worst-case times for both insert and removeMax are logarithmic in the number of values in the priority queue.

- Complexity of priority queue operations :

Insert [ O(lg N)].

Remove [O(lg N)].

DeleteMin [O(lg N)].

BuildHeap Insert [O(N)].

How to implement it :-

* implementation of a Priority Queue structure built on a SortedDicitionary.
* the Huffmann implementation exposes a constructor which takes a Dictionary that is the frequency mapping for each value.
* It also exposes Dictionary CreateEncodings() which creates the binary encoding for each value in the map.
* The HuffmanNode is a basic tree node structure that keeps track of parent, childs, the value it represents and the value's frequency.

# Huffman Coding (Huffman Tree):-

# 

# 

The steps involved in Huffman coding are the following:

1-Examine the frequency and value for each pixel.

2-Place each value and its frequency into a sorted priority queue.

3-Convert the contents of this priority queue into a binary tree.

4-Traverse the tree to discover the binary encoding of each character

5-save the encoded binary version of that value to the destination file.

* implementation of a Priority Queue structure built on a SortedDicitionary.
* the Huffmann implementation exposes a constructor which takes a Dictionary that is the frequency mapping for each value.
* It also exposes Dictionary CreateEncodings() which creates the binary encoding for each value in the map.
* The HuffmanNode is a basic tree node structure that keeps track of parent, childs, the value it represents and the value's frequency.
* The algorithm places the values and it's frequency into binary tree nodes.
* The nodes are put into a priority queue, which keeps them in sorted order with smaller frequencies at the front of the queue.
* Now the algorithm repeatedly removes two nodes from the front of the queue.
* A new node is created and the two nodes are set as children for the new node; the first node becomes the left child, and the second the right.
* The frequency variable of the new node is set to the accumulated value of the two nodes.
* if the frequency of node n1 was 5 and node n2 was 7 then the frequency of the new node is set to 12.
* The new node is then reinserted into the queue. This process is repeated until the queue contains only one node. This node is the root of the finished Huffman tree.
* The Priority Queue structure is up next. A typical priority queue let's you enqueue an arbitrary set of objects, each if which are associated with a value or priority.
* It then let's you dequeue one object at the time and it automatically makes sure that it's the item with the lowest priority that are dequeued.
* Next step of the algorithm is calculate the path to discover the binary encoding and by calculating the path from the root down we can create a path from root to node.
* we assigned left and right with '0' and '1' we can now construct a binary representation of the path.
* By using the CreateEncodings() method of HoffmanTree we can create a mapping for each value and binary path.
* This function won't serve more purpose than displaying that the tree has been built correctly.

Used Function:-

The Encryption and Decryption Part :

-First :-

the function " GET\_Key(ref string seed, int tap)" it takes a reference from the input seed an the tap as parameters ,, it return a variable "key" containing the new key produced from xor process.   
Xoring : int res = (seed[0] - 48 ^ seed[seed.Length - tap - 1] - 48),,  
this line take the last bit in the register and xor it with the bit of the tap and store the result in a variable "res".

Shifting : seed = seed.Substring(1, seed.Length - 1) , this line shifts the string left removing the left most bit.

-Second:-

the function "RGBPixel[,] encrypt\_image(RGBPixel[,] ImageMatrix, string seed, int tap)" , it is a function which return the new encrypted image "matrix of pixels" ,

it takes the matrix of the image's pixels , the initial seed and the tap as parameters.

firstly we should get the key of each color in the image, these 3 lines shows that ,,

the red color key :

string Rkey = GET\_Key(ref seed, tap)

the green color key :

string Gkey = GET\_Key(ref seed, tap)

the blue color key :

string Bkey = GET\_Key(ref seed, tap)

-here we first get the height and the width of the matrix "image" using these functions :

int hight = GetHeight(ImageMatrix);

int width = GetWidth(ImageMatrix);

-The "Bonus Part" , the for loop is moving on each character in the seed string , and the the if condition checks if the character is not '0' or '1' -->"used for binary representation of the seed" , we convert this character to binary value and put the new binary bit in the "seed\_alpha" string , and the return it back "the new binary seed stored in seed\_alpha" to the seed variable used for encryption.

for (int i = 0; i < seed.Length; i++)

{

if (seed[i] != '0' && seed[i] != '1')

{

string seed\_alpha = "";

for (int j = 0; j < seed.Length; j++)

seed\_alpha += Convert.ToString(seed[j], 2);

seed = seed\_alpha;

break;

}

}

and then ,,

the for loop gets each pixel in the image then ,

we xor each color in each pixel with its key and convert it to byte.

for (int i = 0; i < hight; i++)

{

for (int j = 0; j < width; j++)

{

string Rkey = GET\_Key(ref seed, tap);

string Gkey = GET\_Key(ref seed, tap);

string Bkey = GET\_Key(ref seed, tap);

ImageMatrix[i, j].red = (byte)(ImageMatrix[i, j].red ^ Convert.ToByte(Rkey, 2));

ImageMatrix[i, j].green = (byte)(ImageMatrix[i, j].green ^ Convert.ToByte(Gkey, 2));

ImageMatrix[i, j].blue = (byte)(ImageMatrix[i, j].blue ^ Convert.ToByte(Bkey, 2));

}

}