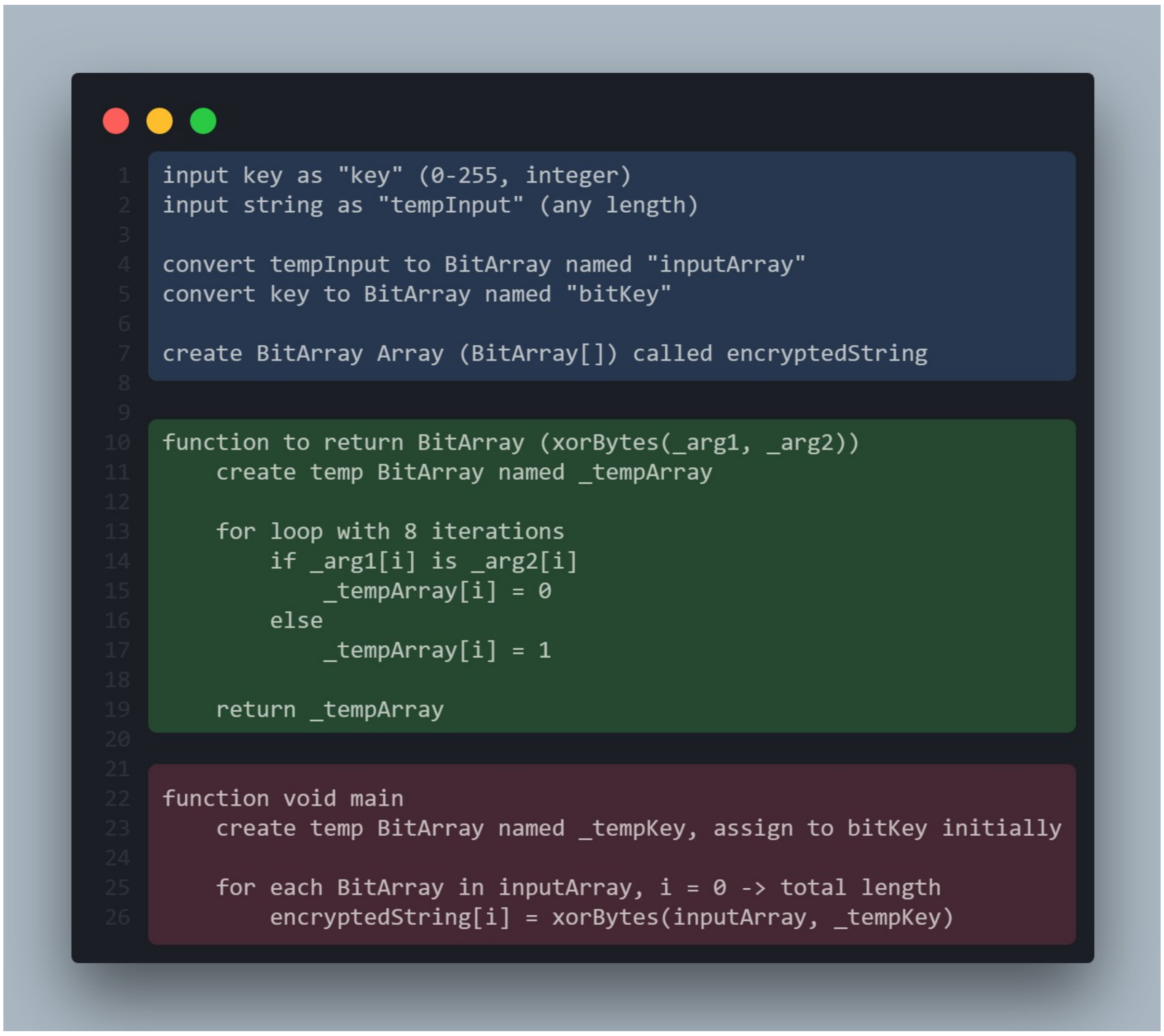
XOR ENCRYPTION

XOR Encryption is a very basic encryption standard that requires using the XOR operator on each bit of the target data using a key, which is a set of bits with n length (GeeksforGeeks, n.d.). The way this works is that the private key, as well as the original data, is masked by the XOR operation, and can be difficult to work out from the output alone. The simplistic algorithm of what this involves is as follows:

$A \wedge B = C$

A being the unencrypted data, B being the key, and C being the encrypted data (Microsoft, n.d.). This method of encryption is vulnerable to brute-forcing, however, which involves generating lots of keys and trying to decrypt C with them, however this is slow and even more difficult on small data sets. This is because one notable solution relies upon frequency analysis, which is less accurate on small data sets. (Zus, n.d.)

There's a few steps required in making an XOR cipher in software, namely converting the input data into a set of bytes (or bits), using the XOR operator between those bytes/bits and a key, then outputting all of the encrypted data to a string or a file.



```
input key as "key" (0-255, integer)
input string as "tempInput" (any length)

convert tempInput to BitArray named "inputArray"
convert key to BitArray named "bitKey"

create BitArray Array (BitArray[]) called encryptedString


function to return BitArray (xorBytes(_arg1, _arg2))
    create temp BitArray named _tempArray

    for loop with 8 iterations
        if _arg1[i] is _arg2[i]
            _tempArray[i] = 0
        else
            _tempArray[i] = 1

    return _tempArray


function void main
    create temp BitArray named _tempKey, assign to bitKey initially

    for each BitArray in inputArray, i = 0 -> total length
        encryptedString[i] = xorBytes(inputArray, _tempKey)
```

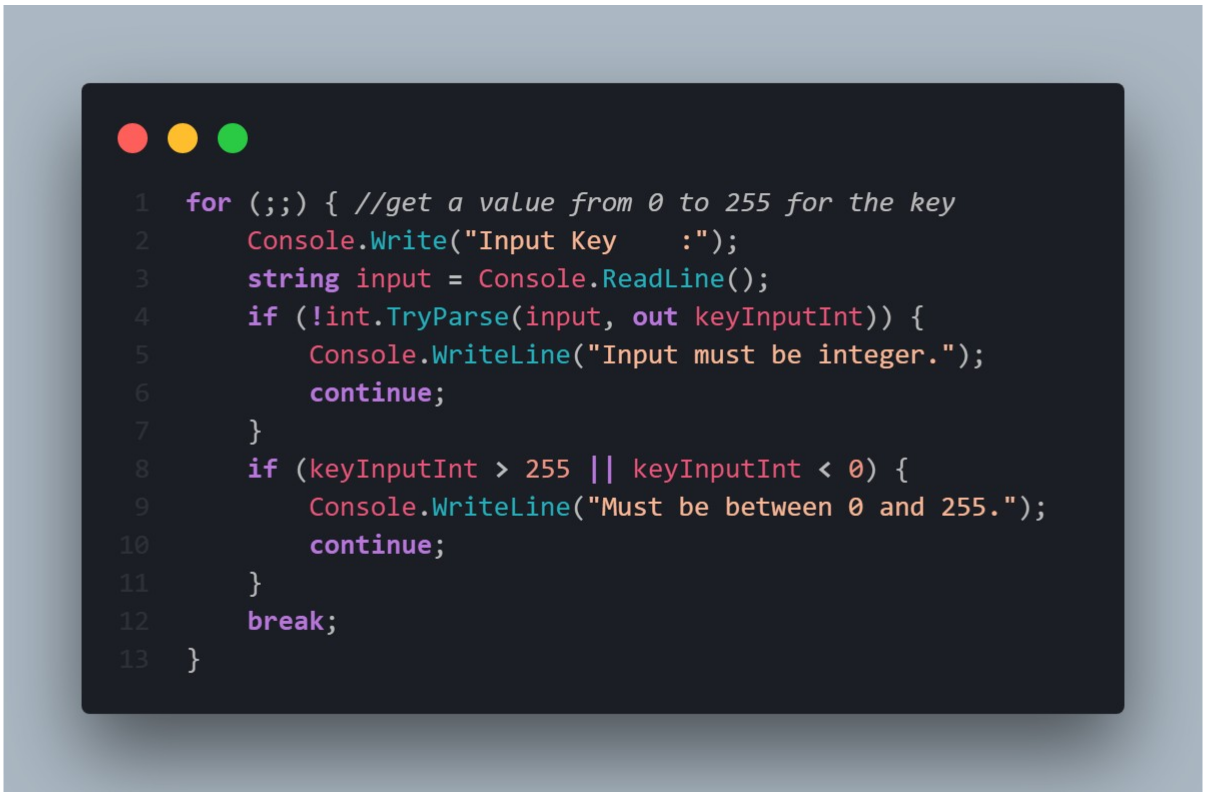
The three sections in the code screenshot above are colour coded as such:

- Blue – Creates a set of BitArrays - A, B and C from the basic algorithm defined earlier. This section may include some looped segments, converting strings and integers into BitArrays.
- Green – This section handles the XOR operation that'll be used on each byte (8 bits) in the input BitArray.
- Red – This section handles the loop that performs the Green section on every byte in the input string, as well as outputting the encrypted string.

Overall, these segments are simplistic as they oversimplify the type conversions required for this kind of program to work. I expect that the final code implementation would be much more complex than the pseudocode above. I'm unsure as to how the specific types will be used, as it's not clear whether BitArrays are the ideal object type for the XOR operation in the green function.

CODE PRODUCTION

INPUTS



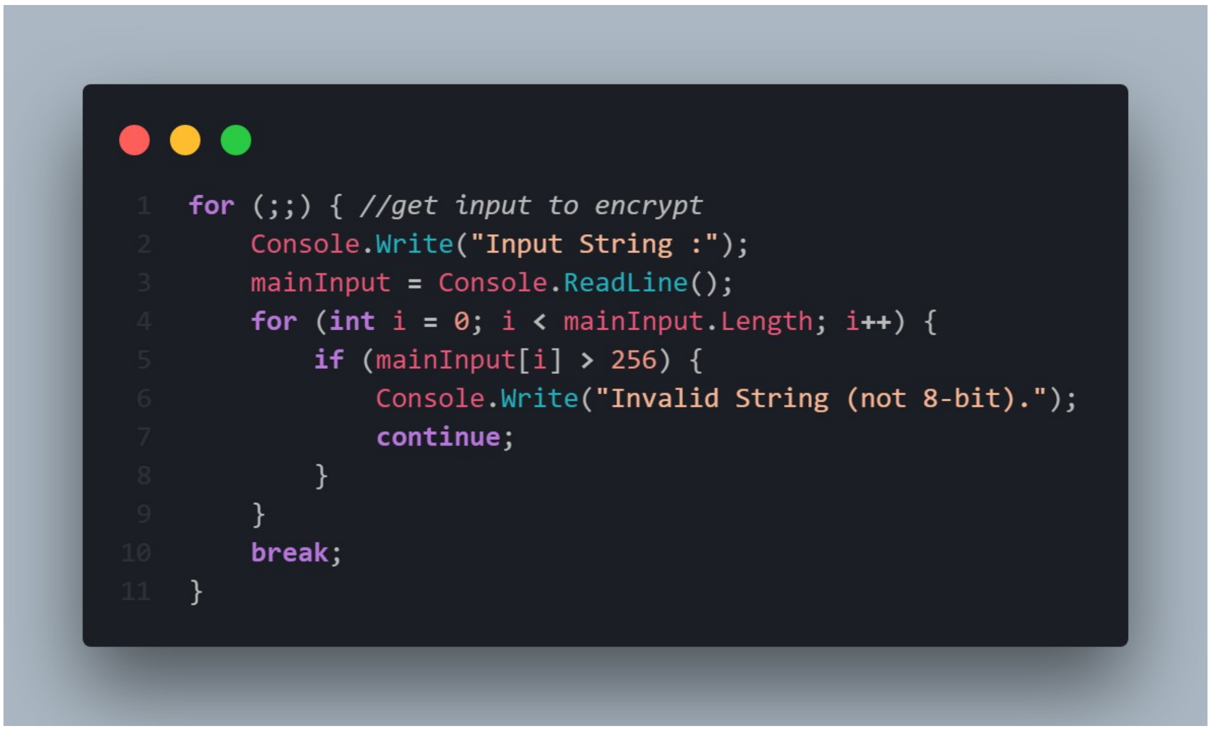
```
for (;;) { //get a value from 0 to 255 for the key
    Console.Write("Input Key    :");
    string input = Console.ReadLine();
    if (!int.TryParse(input, out keyInputInt)) {
        Console.WriteLine("Input must be integer.");
        continue;
    }
    if (keyInputInt > 255 || keyInputInt < 0) {
        Console.WriteLine("Must be between 0 and 255.");
        continue;
    }
    break;
}
```

Inputs in C# (using the console) rely on the function Console.ReadLine(). This takes in text input as a string, relying on extra code to convert the inputs into other types. The issue arises, in this case, from trying to convert strings into integers or into a set of 8 bits. For the first example, the user could input a letter, which cannot be cast to an integer, causing the program to crash. For the second example, should the user input a character that falls outside of an 8-bit character set (i.e. a Unicode character), the conversion from each letter to 8 bits will fail. For these reasons, I needed to add in type conversion checks inside a loop so that the user can input many potentially invalid inputs without needing to restart the script.

This is made possible because of a few features. The first problem can be remedied with int.TryParse(), which is an explicit conversion from a set of possible types into an integer. TryParse is interesting, because it is a function that attempts to parse the value into an integer but returns a Boolean. It performs these tasks simultaneously, assigning the variable and outputting true, or failing and outputting false.

The second segment to this input section involves checking whether the integer inputted is between 0 and 255, as the key in this software is going to be a simple 8 bit key, as this allows for each character to be encoded with a key without having to worry about bit deltas caused by key-string length differences.

Both if statements have a continue; line, as this restarts the loop from the top when encountered. Similarly, the break; at the end escapes the loop instead of continuing from the top. If the parsing fails (!int.TryParse()) or if the key is greater than 255 or less than 0, then the loop resets and the user has to input again. Otherwise, the loop is escaped, and the code continues with a valid input for the key.



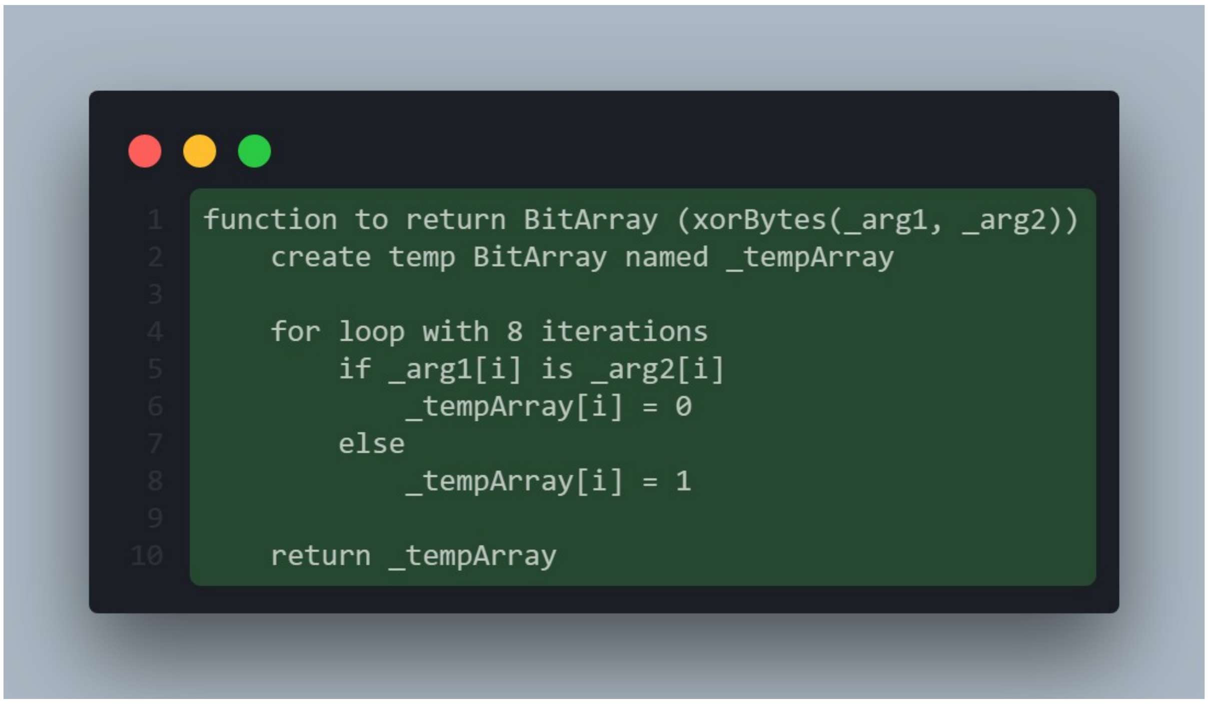
```
for (;;) { //get input to encrypt
    Console.Write("Input String :");
    mainInput = Console.ReadLine();
    for (int i = 0; i < mainInput.Length; i++) {
        if (mainInput[i] > 256) {
            Console.Write("Invalid String (not 8-bit).");
            continue;
        }
    }
    break;
}
```

The same process applies for the input for the string to encrypt. This process is simpler as there’s no type conversion, however there’s still a check to make sure that each character is representable in an 8-bit format. I assume that the input character set depends on the console that the user is using, which could potentially result in issues should one person use extended ASCII while someone else uses UTF-8. For default settings, however, this process should work fine.

Overall, this section expanded in the actual codebase compared to the pseudocode by a large margin. The pseudocode only had two lines for this whole segment, whereas here there’s 24 lines in total. This system is much more resilient than a single input and a simple type parse which might fail, however.

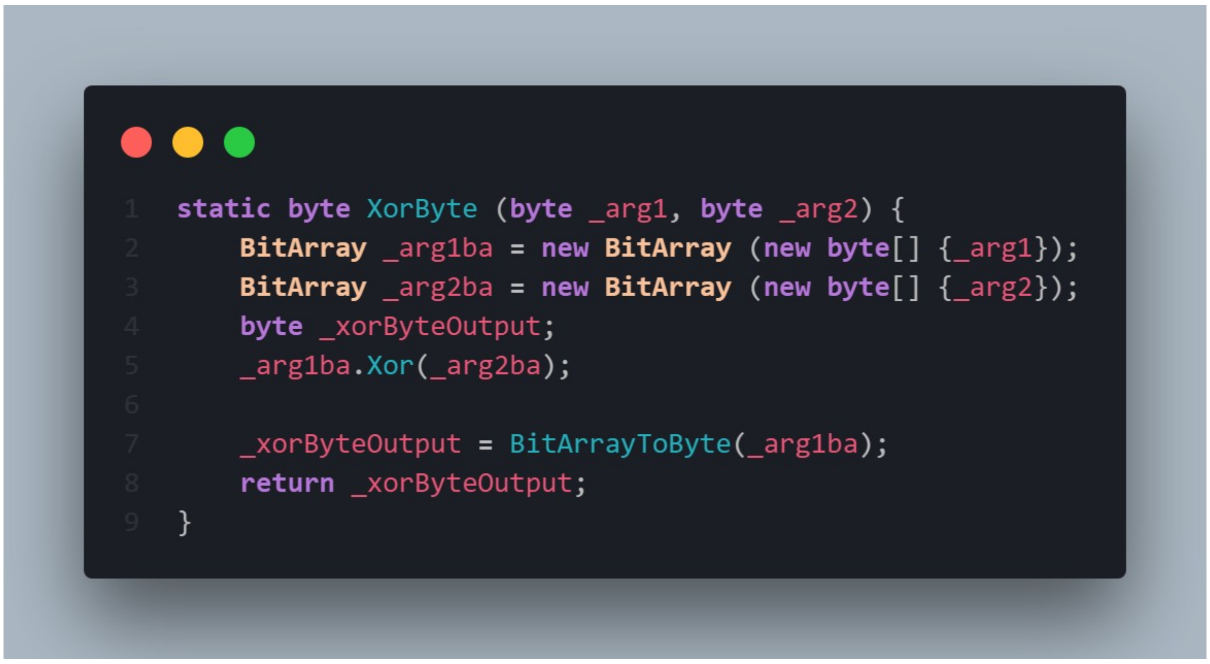
FUNCTION TO XOR BYTES

This portion of pseudocode shows the XOR process being performed on two input bytes. These inputs come from the text input string (n bytes) and the key (1 byte).

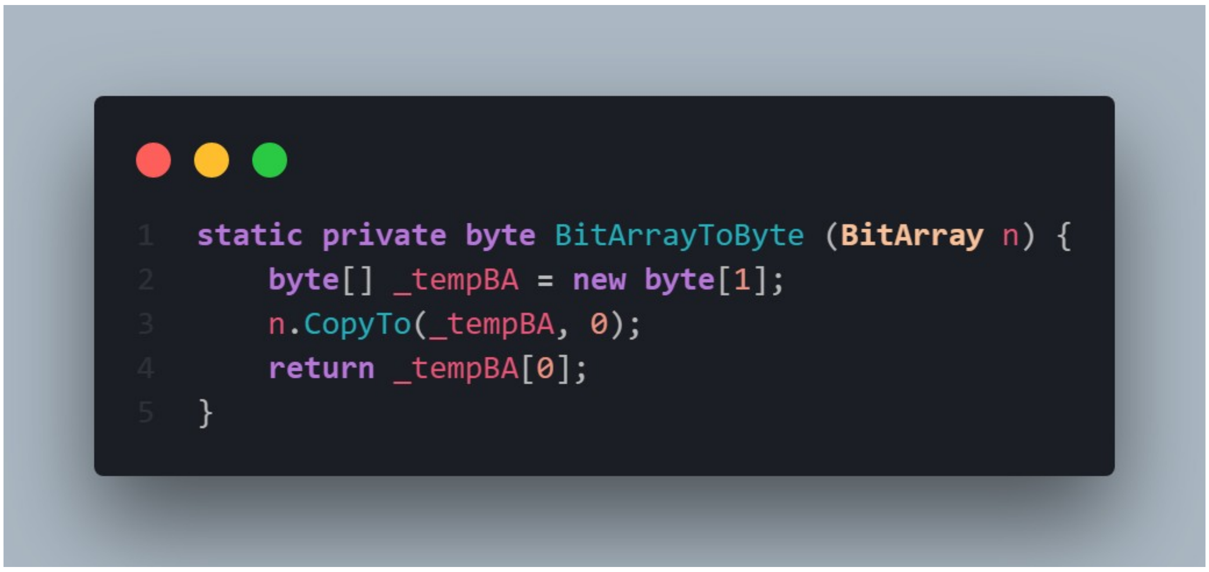


```
function to return BitArray (xorBytes(_arg1, _arg2))
    create temp BitArray named _tempArray

    for loop with 8 iterations
        if _arg1[i] is _arg2[i]
            _tempArray[i] = 0
        else
            _tempArray[i] = 1

    return _tempArray
```

Below is the same pseudocode produced in the actual codebase, simplifying the process somewhat by using the BitArray method Xor, which performs the XOR operation using the parent BitArray and the argument provided, updating the parent. In this case, _arg1ba becomes the output. I used a function called BitArrayToByte, which is a simple byte method that converts a BitArray (of 8 bits) into a byte.



```
static byte XorByte (byte _arg1, byte _arg2) {
    BitArray _arg1ba = new BitArray (new byte[] {_arg1});
    BitArray _arg2ba = new BitArray (new byte[] {_arg2});
    byte _xorByteOutput;
    _arg1ba.Xor(_arg2ba);

    _xorByteOutput = BitArrayToByte(_arg1ba);
    return _xorByteOutput;
}
```



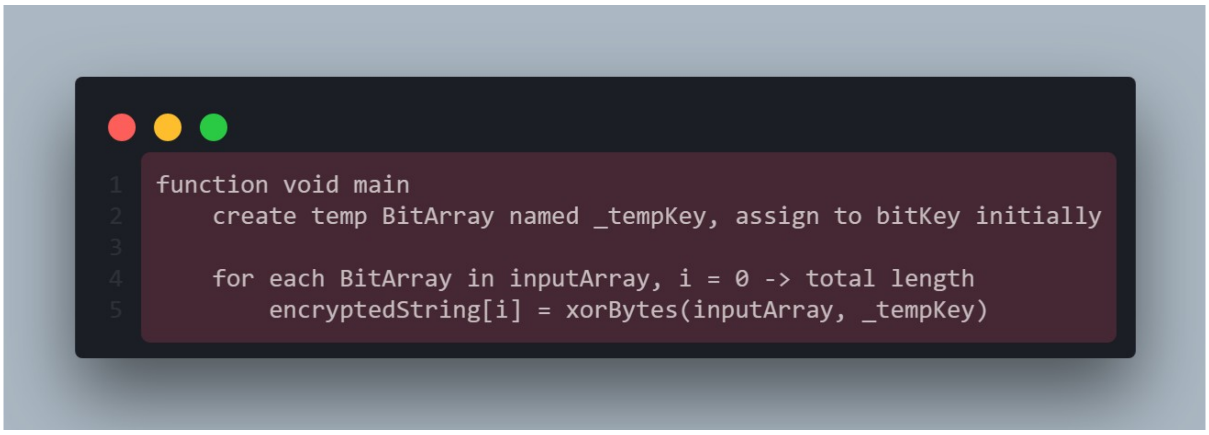
```
static private byte BitArrayToByte (BitArray n) {
    byte[] _tempBA = new byte[1];
    n.CopyTo(_tempBA, 0);
    return _tempBA[0];
}
```

The pseudocode and the actual code solution differ by having an extra type conversion at the top, taking in an input of bytes rather than two BitArrays. As well as this, it outputs a byte object rather than a BitArray. This helps keep the BitArray manipulation contained within these functions, which prevents any non-8 length BitArrays coming up anywhere else in the script.

ENCRYPTION LOOP

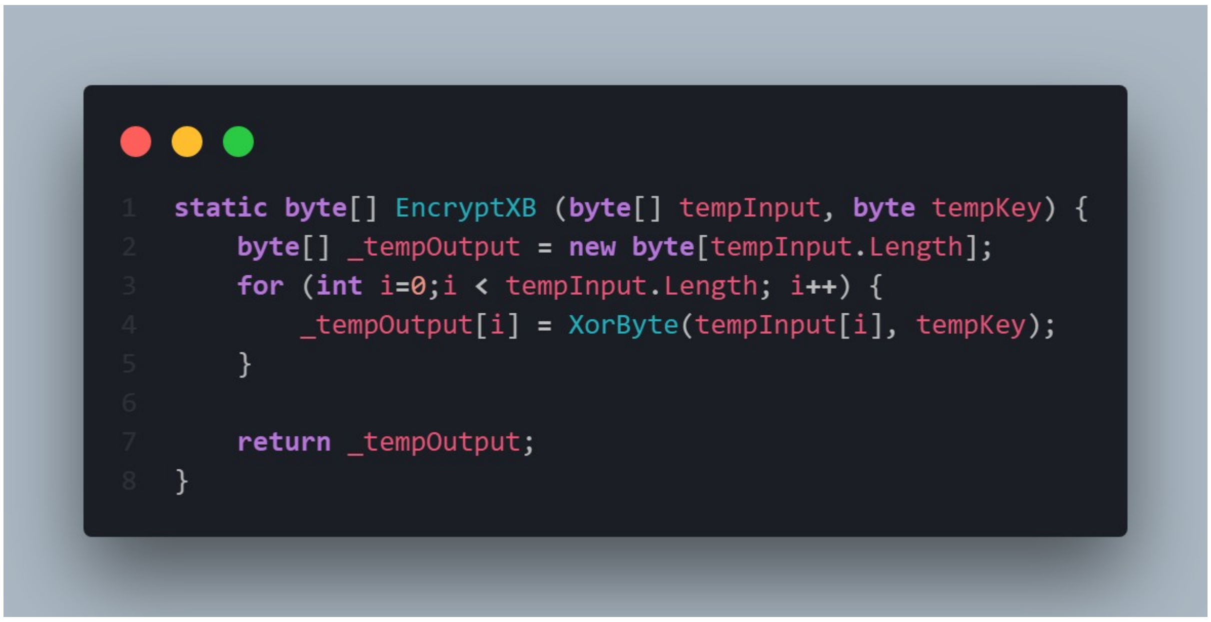
This portion of the pseudocode outlines the main loop that encrypts each byte using the key. The pseudocode glosses over a type conversion (xorBytes returns a BitArray not a string) and assigning a BitArray to a string using an index[i] wouldn't work. It covers the core requirement of having each BitArray (byte) be encrypted and outputted, however. The inputArray is an array of BitArrays (BitArray[]), which is not a very good solution and was changed in the final codebase.

This section of the pseudocode is called main, however the actual code has this be a separate function, with different things going on in the main loop.



```
function void main
    create temp BitArray named _tempKey, assign to bitKey initially

    for each BitArray in inputArray, i = 0 -> total length
        encryptedString[i] = xorBytes(inputArray, _tempKey)
```

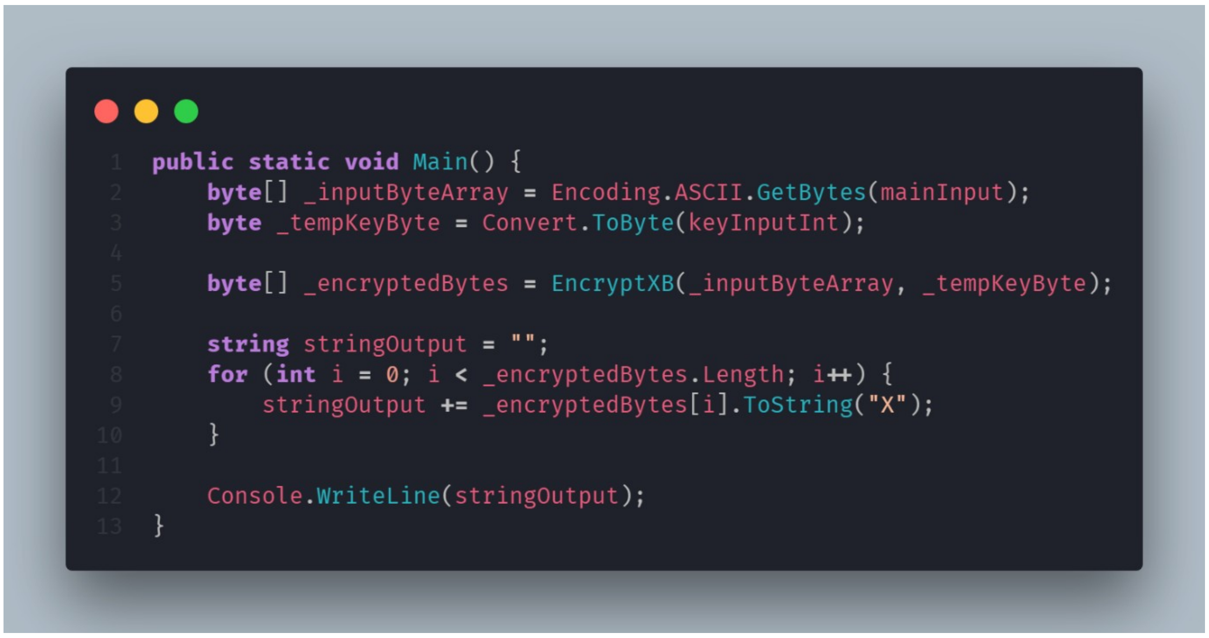
This section of the actual code is an implementation of the loop that encrypts each byte. Having the byte array as an argument allows for a loop inside the function that compares each of those bytes with the key byte. This function relies on XorByte(), described above, which compares the two bytes. It also has a temporary output byte as well. This method is very simple as it has very few steps included, and it doesn't really differ from the pseudocode apart from having a return value and being in a separate method.



```
static byte[] EncryptXB (byte[] tempInput, byte tempKey) {
    byte[] _tempOutput = new byte[tempInput.Length];
    for (int i=0;i < tempInput.Length; i++) {
        _tempOutput[i] = XorByte(tempInput[i], tempKey);
    }

    return _tempOutput;
}
```

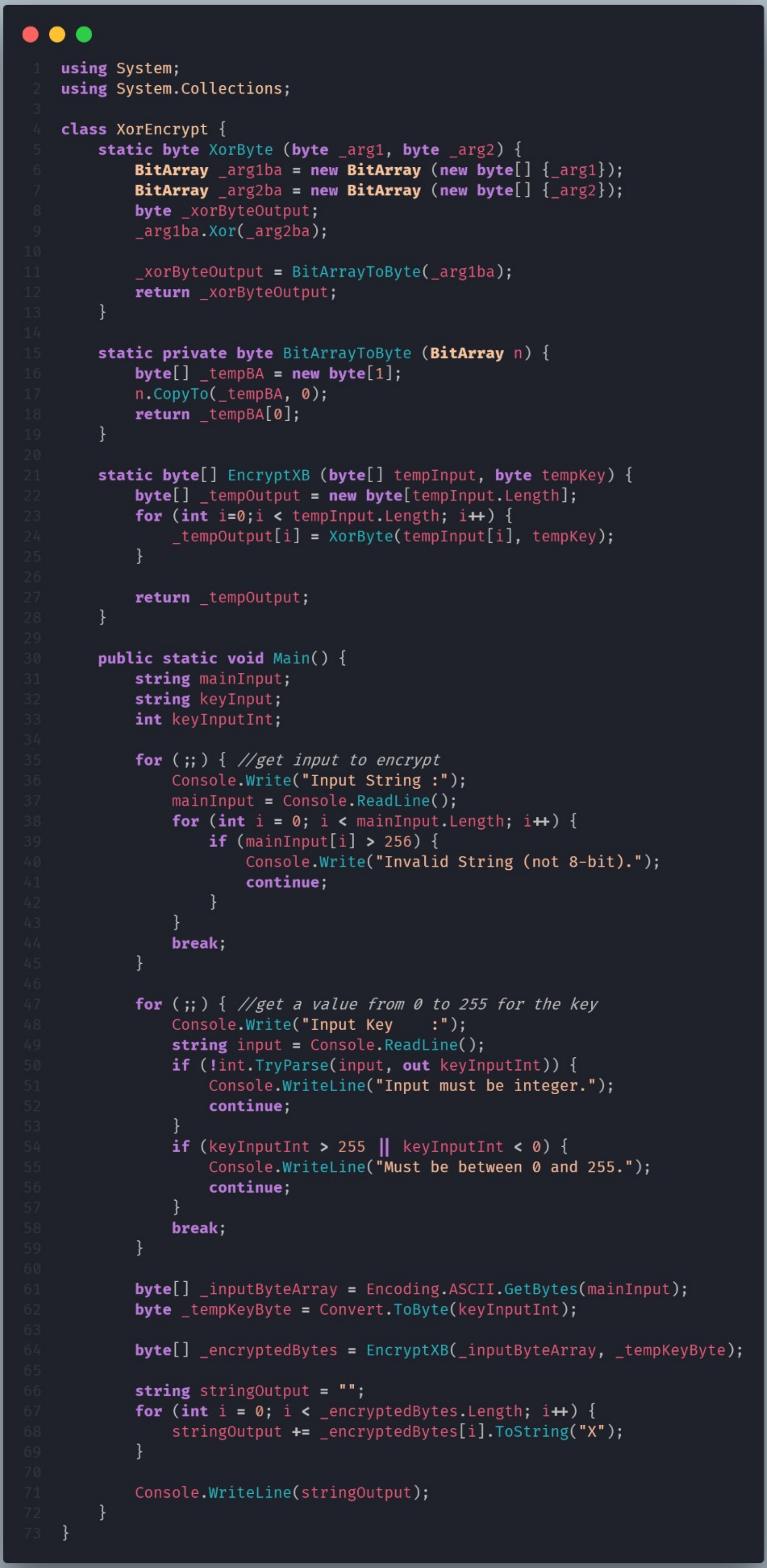
MAIN METHOD

This section doesn't necessarily have an analogue in the pseudocode; however, it extends the EncryptXB method and calls it directly. The inputs recorded at the start of the code are converted into bytes, and a temporary array of bytes for the output is created then filled with the output of the EncryptXB portion. The output of this is then put into a string using a for loop, and written to the output.



```
public static void Main() {
    byte[] _inputByteArray = Encoding.ASCII.GetBytes(mainInput);
    byte _tempKeyByte = Convert.ToByte(keyInputInt);

    byte[] _encryptedBytes = EncryptXB(_inputByteArray, _tempKeyByte);

    string stringOutput = "";
    for (int i = 0; i < _encryptedBytes.Length; i++) {
        stringOutput += _encryptedBytes[i].ToString("X");
    }

    Console.WriteLine(stringOutput);
}
```

FULL CODE



```csharp
using System;
using System.Collections;

class XorEncrypt {
    static byte XorByte (byte _arg1, byte _arg2) {
        BitArray _arg1ba = new BitArray (new byte[] {_arg1});
        BitArray _arg2ba = new BitArray (new byte[] {_arg2});
        byte _xorByteOutput;
        _arg1ba.Xor(_arg2ba);

        _xorByteOutput = BitArrayToByte(_arg1ba);
        return _xorByteOutput;
    }

    static private byte BitArrayToByte (BitArray n) {
        byte[] _tempBA = new byte[1];
        n.CopyTo(_tempBA, 0);
        return _tempBA[0];
    }

    static byte[] EncryptXB (byte[] tempInput, byte tempKey) {
        byte[] _tempOutput = new byte[tempInput.Length];
        for (int i=0;i < tempInput.Length; i++) {
            _tempOutput[i] = XorByte(tempInput[i], tempKey);
        }

        return _tempOutput;
    }

    public static void Main() {
        string mainInput;
        string keyInput;
        int keyInputInt;

        for (;;) { //get input to encrypt
            Console.Write("Input String :");
            mainInput = Console.ReadLine();
            for (int i = 0; i < mainInput.Length; i++) {
                if (mainInput[i] > 256) {
                    Console.Write("Invalid String (not 8-bit).");
                    continue;
                }
            }
            break;
        }

        for (;;) { //get a value from 0 to 255 for the key
            Console.Write("Input Key    :");
            string input = Console.ReadLine();
            if (!int.TryParse(input, out keyInputInt)) {
                Console.WriteLine("Input must be integer.");
                continue;
            }
            if (keyInputInt > 255 || keyInputInt < 0) {
                Console.WriteLine("Must be between 0 and 255.");
                continue;
            }
            break;
        }

        byte[] _inputByteArray = Encoding.ASCII.GetBytes(mainInput);
        byte _tempKeyByte = Convert.ToByte(keyInputInt);

        byte[] _encryptedBytes = EncryptXB(_inputByteArray, _tempKeyByte);

        string stringOutput = "";
        for (int i = 0; i < _encryptedBytes.Length; i++) {
            stringOutput += _encryptedBytes[i].ToString("X");
        }

        Console.WriteLine(stringOutput);
    }
}
```

BIBLIOGRAPHY

GeeksforGeeks. (n.d.). XOR Cipher - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/xor-cipher/ [Accessed 7 Nov. 2019].

Microsoft (n.d.). Bitwise and shift operators - C# reference. *[online] Docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#logical-exclusive-or-operator- [Accessed 7 Nov. 2019].*

Zus, J. (n.d.). Breaking a XOR cipher of known key length? [online] Cryptography Stack Exchange. Available at: https://crypto.stackexchange.com/questions/56281/breaking-a-xor-cipher-of-known-key-length [Accessed 7 Nov. 2019].