1 Consignes de rendu

Après avoir réalisé ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- tp6.sln*
    |-- Ex1
        |-- Ex1.csproj
        |-- Tout sauf bin/ et obj/
    |-- Photoshop
        |-- Photoshop.csproj
        |-- BMPReader.cs
        |-- Filters.cs
        |-- Stegano.cs
        |-- Tout sauf bin/ et obj/
    |-- Crypto
        |-- Crypto.csproj
        |-- Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer "`login_x`" par votre propre login. Les fichiers annotés d'une astérique sont *OBLIGATOIRES*.

N'oubliez pas de vérifier les points suivants avant de rendre :

— Le fichier `AUTHORS` doit être au format habituel (une *, un espace, votre login et un retour à la ligne).
— Pas de dossiers `bin` ou `obj` dans le projet.
— **Le code doit compiler !**

2 Introduction

Durant ce Tp vous aurez l'occasion de vous familiariser avec différents concepts informatiques tel que le traitement d'images, la stéganographie, ainsi que le chiffrement.

3 Cours

3.1 Opérateurs bitwise

Commençons par le commencement. De quoi un fichier est-il constitué ? Un fichier n'est en réalité qu'une longue suite d'octets, chacun composés de 8 bits, s'ajoutant pour une valeur variant entre 0 et 255 inclus. Ces octets (ou *bytes* en anglais) sont le support matériel de la donnée en informatique. Dans ce tp, vous apprendrez à manipuler les fichiers, et devrez donc aussi manipuler les octets.
Pour ce faire, le c# nous fournit des outils que l'on appelle *opérateurs bitwise*. Ils sont les opérateurs NOT, AND, OR, XOR, SHIFT.

Vous devriez déjà connaître les opérateurs AND, OR, et XOR. Le NOT quant à lui, représente le complément à 1 du byte voulu. Il s'utilise grâce à un '~' placé devant le byte. Pour plus d'informations, allez fouiller la MSDN.

Le SHIFT est une notion importante pour le tp de cette semaine. Son effet est de déplacer tous les bits du byte du nombre de places voulues. Voici un exemple d'utilisation :

```
byte b = 60;        // b1 = 00111100 = 60
byte b2 = b << 1;   // b2 = 01111000 = 120
byte b3 = b2 << 3; // b3 = 11000000 = 192
byte b4 = b3 >> 3; // b4 = 00011000 = 24
byte b5 = b4 >> 1; // b5 = 00001100 = 12
```

Il y a plusieurs choses à remarquer ici. Premièrement, vous pouvez voir qu'il existe deux shifts différents : le gauche et le droit. De plus, vous pouvez remarquez que si vous décalez trop vos bits d'un coté, vous les perdez ! En effet, les bits qui sont poussés en dehors de l'octet lors du shift sont perdus à tout jamais ! Faites donc attention à cette subtilité. Troisièmement, vous pouvez remarquer qu'un shift à gauche revient à une multiplication par deux, et un shift vers la droite une division par deux. Ceci est normal étant donné que nous travaillons ici en base 2. Notez que ces shifts sont plus rapides à effectuer par votre ordinateur que des multiplications classiques.

3.2 Manipulation de fichiers

Comme l'indique son titre, ce tp portera principalement sur la manipulation de fichiers, chose essentielle en programmation. Les deux notions principales que l'on va aborder sont l'écriture et la lecture. Afin d'exécuter ces actions, vous aurez à votre disposition des outils tout prêts que sont les *FileStreams*. Vous vous en servirez comme 'intermédiaire' entre vous et le fichier. Vous devrez l'ouvrir, faire vos manipulations, puis le fermer. (N'oubliez surtout pas de le fermer ou vous devrez faire face à des erreurs très étranges !)
Ces *FileStreams* servant principalement à manipuler des octets, ils restent peu pratiques à utiliser pour écrire du texte... Aussi a-t-on à notre disposition les *StreamReaders* et les *StreamWriters*. Ces deux outils nous permettront de respectivement lire et écrire dans les fichiers beaucoup plus facilement !

3.2.1 Lecture de fichiers

Afin de lire dans des fichiers, nous utiliserons les *StreamReaders*. Ces outils mettent à notre disposition des fonctions très utiles, qui nous permettront d'arriver très facilement à nos fins. Voici comment les utiliser :

```
0   static void Main(string[] args)
1   {
2       // Instanciate a new StreamReader:
3       StreamReader stream_reader =
4               new StreamReader("my_file_name.txt");
5
6       //Reader into the file via the StreamReader:
7       string str = stream_reader.ReadLine();
8
9       //See what we just read:
10      Console.WriteLine(str);
11
12      //Never forget to close your StreamReader:
13      stream_reader.Close();
14  }
```

Le code est ici bien commenté, donc vous pouvez facilement comprendre ce qu'il fait : il lit une ligne dans le fichier, l'écrit sur la sortie standard (la console), et quitte le programme sans oublier de fermer le *StreamReader*.

3.2.2 Ecrire dans un fichier

L'écriture dans les fichiers se fait de façon symétrique à la lecture, grâce aux *StreamWriters*. Voici un petit exemple d'utilisation :

```
0   static void Main(string[] args)
1   {
2       // Instanciate a new StreamWriter:
3       StreamWriter stream_writer
4           = new StreamWriter("my_file_name.txt");
5
6       //Set the string that we wiant to write:
7       string str = "This is a fabulous line!";
8
9       //Write it in the file as a line:
10      stream_writer.WriteLine(str);
11
12      //Never forget to close your StreamWriter:
13      stream_writer.Close();
14  }
```

Ici encore, le code est explicite : on veut simplement écrire dans le fichier puis quitter.
Il y a cependant quelque chose d'intéressant à remarquer ici : la ligne 13 Ne vous fait-elle pas penser à ce bon vieux *Console.WriteLine();* ? Si ? C'est normal ! On utilise ici le même principe, vous ne serez donc pas très dépaysés !

Notez que l'on vous donne ici des exemples très simples, nous vous encourageons donc à fouiller la MSDN à la recherche d'informations.

3.3 Fichiers binaire

Les fichiers ne contiennent pas tous du texte. Un fichier binaire (par exemple BMP, EXE, WAV, etc ...) ne sera pas lisible par un humain dans un éditeur de texte.

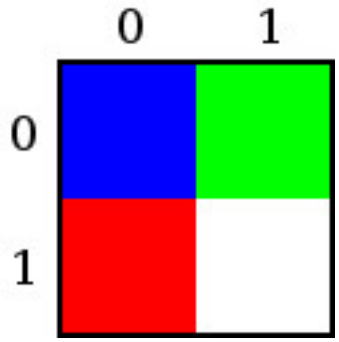
Ces fichiers comportent une structure particulière selon leur type. La plupart des fichiers binaires commencent par un champ appelé magic number. Ce champ est une signature de quelques octets permettant d'identifier le type du fichier. Par exemple, un fichier PDF débute par 4 octets au format ASCII : %PDF. Dans le cas d'un fichier BMP, le magic number est : BM

3.4 Le format BMP

Le bitmap est un format de fichier non compressé d'image. Il est composé de 4 zones :
— l'en-tête BMP ;
— l'en-tête DIB ;
— la palette de couleurs (ne nous intéressera pas pour ce TP) ;
— les données relatives à l'image.
Détails complets sur le format BMP `http://en.wikipedia.org/wiki/BMP_file_format`

Décalage	Taille	Valeur Hex	Déscription
0h	2	42 4D	Nombre magique
2h	4	46 00 00 00	Taille du fichier BMP
6h	2	00 00	Réservé
8h	2	00 00	Réservé
Ah	4	36 00 00 00	Décalage
		DIB Header	
Eh	4	28 00 00 00	Taille du DIB Header (en octet)
12h	4	02 00 00 00	Largeur
16h	4	02 00 00 00	Hauteur
1Ah	2	01 00	Nombre de plan de couleurs utilisé
1Ch	2	18 00	Nombre de bit par pixels
1Eh	4	00 00 00 00	Compression utilisé (aucune)
22h	4	10 00 00 00	Taille du tableau de pixel (padding inclus)
26h	4	13 0B 00 00	Réservé pour l'impression
2Ah	4	13 0B 00 00	Réservé pour l'impression
2Eh	4	00 00 00 00	Nombre de couleurs dans la palette
32Eh	4	00 00 00 00	Nombre de couleurs importantes
		Début du tableau de pixel	
36h	3	00 00 FF	Rouge
39h	3	FF FF FF	Blanc
3Ch	2	00 00	Padding
3Eh	3	FF 00 00	Bleu
41h	3	00 FF 00	Vert
44h	2	00 00	Padding

Le tableau précédent représente l'image suivante :



La matrice de pixels est un bloc qui décrit l'image pixel par pixel. Normalement les pixels sont stockés "à l'envers", en commençant dans le coin inférieur gauche, allant de gauche à droite, puis rangée par rangée du bas vers le haut de l'image.

De plus, des octets de 'padding' (dont la valeur n'est pas nécessairement 0) doivent être ajoutés à la fin des lignes pour aligner la longueur des lignes sur un multiple de quatre octets.

3.5 Stéganographie

La stéganographie est l'art de la dissimulation. Elle consiste à cacher un message au sein d'un autre message anodin, de sorte que l'on ignore l'existence même du secret.

Alors que la cryptographie repose sur le fait que le message ne soit pas compris, la stéganographie repose sur le fait que le message ne soit pas trouvé.

Pour cacher nos informations, nous allons exploiter le fait que 2 couleurs trop proches sont quasiment indiscernables par l'œil humain.
Nous allons donc utiliser le dernier bit de chaque octet pour cacher nos données. Il s'agit du bit de poids faible : le modifier revient à modifier l'octet de moins de 0,4%, tandis que modifier le premier bit, le bit de poids fort, aurait fait par exemple passer 50 à 178, soit de plus de 50%, ce qui est très voyant comme différence.

Cas pratique : Si vous voulez cacher 6 (110) dans le pixel suivant : p(120, 90, 212) ou p(0111 1000, 0101 1010, 1101 0100) vous allez modifier le bit de poid faible de chacune des composantes du pixel p et obtenir le pixel suivant : (0111 1000, 0101 1011, 1101 0101) soit (120, 91, 213) ce qui n'a que peut d'impact sur la couleur.

Pour modifier le bit de poids faible vous allez devoir utiliser les opérateurs binaire tel que le 'et' ainsi que le 'ou'.
Voici un exemple :

```
byte a = 0x78; // a = 120 = 0111 1000
a = a & F9;    // a = 121 = 0111 1001
```

3.6 Crypto : Vigenère

Le chiffre de Vigenère est un système de chiffrement par substitution, mais une même lettre du message clair peut, suivant sa position dans celui-ci, être remplacée par des lettres différentes, contrairement à un système de chiffrement monoalphabétique comme le chiffre de César (qu'il utilise cependant comme composant).

Cette méthode résiste ainsi à l'analyse de fréquences, ce qui est un avantage décisif sur les chiffrements monoalphabétiques. Cependant le chiffre de Vigenère a été cassé par le major prussien Friedrich Kasiski qui a publié sa méthode en 1863. Il n'offre plus depuis cette époque aucune sécurité.

Ce chiffrement introduit la notion de clé. Une clé se présente généralement sous la forme d'un mot ou d'une phrase. Pour pouvoir chiffrer notre texte, à chaque caractère nous utilisons une lettre de la clé pour effectuer la substitution.
Évidemment, plus la clé sera longue et variée et mieux le texte sera chiffré. Il faut savoir qu'il y a eu une période où des passages entiers d'œuvres littéraires étaient utilisés pour chiffrer les plus grands secrets. Les deux correspondants n'avaient plus qu'à avoir en leurs mains un exemplaire du même livre pour s'assurer de la bonne compréhension des messages.

Pour des cas pratique : `http://fr.wikipedia.org/wiki/Chiffre_de_Vigenère`

4 Exercices

Attention : Lors de ce Tp il vous est demandé de gérer toutes les erreurs susceptibles de faire crasher votre programme (sauf contre indication du sujet).

De plus lors de ce TP nous utiliserons l'objet Color, pour ce faire vous devrez charger la référence System.Drawing (via un click droit sur votre projet ajouter une Référence/add Reference).

Enfin l'utilisation de l'objet Bitmap est prohibé.

4.1 Exercice 0 : Fichiers texte

4.1.1 Lecture

Pour ce premier exercice, nous vous demandons de lire un fichier texte et de renvoyer son contenu dans une string.
En clair écrire la fonction suivante :

```
static String read_file(string filename);
```

4.1.2 Ecriture

Dans cette seconde partie il vous est demandé d'écrire dans un fichier le tableau de string que l'on vous donne en argument.

```
static void write_to_file(string filename, string[] lines);
```

4.2 Exercice 1 : Lecture de BMP && Traitement d'image

Lors de cet exercice nous allons vous demander de lire un fichier au format BMP. Notez que nous supposerons ici que les pixels du BMP n'ont subit aucune compressions et qu'ils font une taille de soit 24 bits (RGB) soit 32 bits (ARGB).

Vous implémenterez la classe BMPReader qui doit contenir au minimum les méthodes et attributs suivants :

```
public class BMPReader {
    public int height { get; private set; }
    public int width  { get; private set; }

    private int pixel_array_offset;
    private short bits_per_pixel;

    private byte[] header;
    private Color[, ] pixels;

    public BitmapReader(string filename);
    public void save(string filename);
    public void display_header(FileStream fs);

    public Color get_pixel(int x, int y);
    public void set_pixel(int x, int y, Color c);

    private void read_header(FileStream fs);
    private void read_pixels(FileStream fs);
}
```

Voici les fonctions et objets dont vous aurez besoin :
— FileStream : Seek, Read, ReadByte, Length ;
— Color : FromArgb ;
— BitConverter : ToInt32, ToInt16.
Conseils :
— Travaillez sur des petites images (3 * 3 ou 3 * 2) ;
— Regardez le BMP dans un éditeur hexadécimal (Hexedit).

4.2.1 Vérifier que le fichier est bien un BMP

Commencez par ouvrir le Filestream vers le fichier en vérifiant que celui-ci est accessible et existe bien dans le constructeur (c'est à dire **implémenter une première partie du constructeur** que vous compléterez au fur et à mesure de l'exercice) puis vous allez vérifier que le fichier que l'on vous donne est bien un fichier BMP.
Pour ce faire vous allez vérifier le magic number ainsi que la taille du fichier (l'entête du fichier possède une taille minimum que le fichier doit avoir).

Vous implémenterez donc la méthode suivante qui doit être appelée par le constructeur et qui doit lancer une exception si le fichier n'est pas un bmp :

```
private bool is_bitmap(FileStream fs);
```

4.2.2 Lire l'entête

Nous vous demandons de réaliser les méthode suivantes :

```
private void read_header(FileStream fs);
public void display_header();
```

Commencez par copier l'entête du BMP dans l'attribut header de votre classe (dans la méthode read_header), vous en aurez besoin.
La méthode read_header doit être appelé par le constructeur. La méthode display_header quand à elle doit afficher la sortie suivante (remplacez XX par les valeurs correspondant à celles de l'image) :

```
Width           : XX pixels
Height          : XX pixels
Offset          : XX bytes
Bits per pixels: XX bits
```

4.2.3 Lire le tableau de pixels

Maintenant que vous connaissez l'offset de l'endroit où sont écrits les pixels il ne vous reste plus qu'à construire votre tableau de pixel en lisant le fichier. La fonction demandée ici est :

```
private void read_pixels(FileStream fs);
```

La méthode read_pixels doit être appelé par le constructeur. N'oubliez pas de fermer votre stream. N'oubliez pas non plus qu'à la fin d'une ligne il existe des bits de padding et qu'il ne faut pas les lire.

Conseil : Affichez la première ligne de l'image pour vérifier que vous lisez correctement.

4.2.4 Ecrire l'image lue

Pour vérifier que votre fonction read_pixel fonctionne correctement vous allez devoir implémenter la méthode

```
public void save(string filename);
```

Qui va écrire en premier le header que vous avez lu dans le fichier 'filename' puis écrire les pixels.

Vérifiez à ce point, que vous êtes capable de lire un fichier bmp et sauvegarder une copie.

4.2.5 get_pixel && set_pixel && ICloneable

Pour finir cet exercice de lecture et d'écriture vous allez devoir implémenter les fonctions suivantes :

```
public Color get_pixel(int x, int y);
public void set_pixel(int x, int y, Color c);
```

De plus afin de pouvoir travailler sur des copies de notre image nous vous demandons de faire en sorte que votre classe BMPReader implémente l'interface ICloneable et la méthode clone qui fera une 'deep copy' de votre objet.

```
public object Clone();
```

4.2.6 Binarize

Il est maintenant temps d'utiliser notre BMPReader pour des choses plus visuelles ! Vous allez maintenant devoir binariser une image. C'est à dire, si la somme des composantes d'un pixel (RGB) divisé par 3 dépasse le seuil donné alors la couleur de ce pixel est noir, sinon elle est blanche.

Ecrivez donc la fonction suivante :

```
public static BMPReader binarize(BMPReader image, int threshold);
```

Qui itère sur tout les pixels de l'image et décide de leur couleur (blanc ou noir) en fonction de la couleur du pixel.

N'oubliez pas de travailler sur une copie (ou un clone) du BMPReader !

4.2.7 Image2Grey

Pour ce second filtre basique vous allez passer une image en couleur vers une image en niveau de gris.

Notre conversion utilise la conversion classique basée sur de la colorimétrie. La version de base calcule la moyenne des composantes RGB de l'image puis fixe tout les pixels à cette valeur. Bien que censée être correcte, la perception humaine des couleurs est légèrement différente : le vert apparaît plus clair que le rouge qui est aussi plus lumineux que le bleu. Ainsi, nous calculerons la moyenne en utilisant le coefficient suivant : 0,3 pour le rouge, 0,59 pour le vert et 0,11 pour le bleu.

Ecrivez donc la fonction suivante :

```
public static BMPReader to_grey(BMPReader image);
```

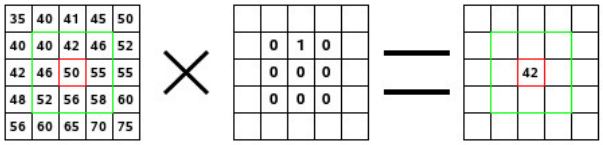
Qui itère sur tous les pixels de l'image et qui applique la formule donnée plus haut à chaque pixel (appliquer la formule et mettre chaque composante du pixel à l'entier trouvé).

N'oubliez pas de travailler sur une copie (ou un clone) du BMPReader !

4.2.8 Matrices de convolution

Ecrivez la fonction convolution qui applique une matrice de convolution carrée (même largeur et hauteur) à une image.

Pour ce faire il faut parcourir l'image et remplacer chaque composante de la couleur du pixel en cours par la somme des produits de la composante équivalente d'un pixel voisin avec son coefficient équivalent dans la matrice. Voici un schéma pour illustrer l'algorithme :



Vous pouvez considérer que les pixels en dehors de l'image sont noirs (tous à 0). Conseils :
- Vous pouvez faire une fonction IsValid(int x, int y, int height, int width) pour tester si des coordonnées sont bien dans l'image.
- Jetez un coup d'œil à la fonction clamp sur l'intervalle [0 ; 255].
- N'oubliez pas de travailler sur une copie (ou un clone) du BMPReader !

Ecrivez donc la fonction suivante :

```
public static BMPReader convolution(BMPReader image, float[,] mat);
```

4.3 Exercice 2 : Stéganographie avec une image

Dans cet exercice, vous allez devoir utiliser les notions du cours afin de cacher un message dans une image.

Votre code se situera dans la classe suivante :

```
public class Stegano {
    public BMPReader img { get; private set; }

    public Stegano(BMPReader img) {
        this.img = img;
    }

    // @return: if the bit was hidden or not
    public bool hide_bit(bool bit, int position);
    public int read_bit(int position);

    public void stegano_encipher(string message);
    public string stegano_decipher();
}
```

4.3.1 get_byte && set_byte

Tout d'abord implémentez les fonctions suivantes dans la classe BMPReader :

```
public byte get_byte(int n);
public void set_byte(int n, byte b);
```

Sachant que vous avez la largeur de l'image (width) vous devriez être capable de retrouver les coordonnées en x et y du pixel.
Attention, il vous faudra tout d'abord diviser par 3 ou 4 en fonction du nombre de bits par pixel (24 ou 32). Puis renvoyer la bonne composante (A, R, G, ou B).

4.3.2 Cacher un bit dans un octet

Vous allez maintenant écrire la fonction suivante :

```
public bool hide_bit(bool bit, int position);
```

Cette fonction sera appelé sur chaque octet de l'image ou jusqu'à ce que vous ayez fini d'écrire le message.

Cette fonction renvoie un booléen indiquant si le bit à été caché. De cette façon vous pouvez cacher le message de façon plus intelligente que au début de l'image (par exemple cacher votre message dans les octets pairs ou uniquement sur la composante rouge de l'image).

Vous pouvez évidemment choisir la manière la plus facile qui est de cacher le message au début de l'image sur le bit de poid faible des X premiers octets.

Cependant les messages caché de manière astucieuse seront récompensés (commencez tout de même par la méthode facile si vous ne vous sentez pas sûr de vous).

4.3.3 Cacher un message dans une image

Pour cette partie il vous faut tout d'abord vérifier que l'image est de taille suffisante pour cacher votre message. Puis il vous suffit d'itérer sur tout les octets de l'image ou jusqu'à ce que votre message soit caché et d'appeler la fonction que vous venez de coder.
Tout ceci doit être fait dans la méthode suivante :

```
public void stegano_hide(string message);
```

Attention **ajoutez la taille de votre message au début de celui-ci** avant de commencer à cacher votre message. Cela vous sera utile.

4.3.4 Lire un message dans une image

Votre fonction respectera ce prototype :

```
public string stegano_find();
```

Qui s'occupera de trouver le message dans l'image et le renverra. Commencez par lire la taille du message puis le message en lui même (4 caractères).

4.3.5 Conseils

- Considérez que votre message ne fera pas plus de 9999 caractères.
- Stockez la taille de votre message avant le message lui-même.
- Stockez votre message bit par bit sur les bits de poids faibles des octets de l'image. Si vous faites autrement, merci de le spécifier dans un "README" à coté du "AUTHORS".

4.4 Crypto

4.4.1 Xor

Cet exercice vous demande de chiffrer et déchiffrer un message stocké sous forme de string à l'aide d'un XOR (appliquer un xor sur chaque caractères).
Voici le prototype à respecter :

```
public static string xor_encode(string message, string key);
public static string xor_decode(string message, string key);
```

"message" correspondra au message à crypter.
"key" sera la clé utilisée afin de crypter.

4.4.2 RotN

Cet exercice vous demande de chiffrer un message stocké sous forme de string à l'aide d'un rotN.
Voici le prototype à respecter :

```
public static string rotN(string message, int n);
```

4.4.3 Vigenère

Cet exercice vous demande de chiffrer et déchiffrer un message stocké sous forme de string à l'aide du chiffrement de Vigenère.
Voici le prototype à respecter :

```
public static string vignere_encode(string message, string key);
public static string vignere_decode(string message, string key);
```

4.4.4 Bonus

Ces méthodes de cryptographie étant relativement simples, voici quelques pistes :
— Machine Enigma ;
— Cryptage Monoalphabétique ;
— Cryptage Polyalphabétique ;
— Cryptage "Playfair" ;
— Cryptage à double transposition ;
— etc . . .

Notez que chaque fonction de cryptage doit être codée avec sa fonction de décryptage.
De plus, il va de soi que les cryptages plus "forts" seront plus récompensés.

The Bullshit is strong with this one...