

TP C#8 : Cipher, Compress & Conquer

Consignes de rendu

A la fin de ce TP, vous devrez rendre un dépôt git respectant l'architecture suivante :

```
csharp-tp8-firstname.lastname/  
|-- README  
|-- Conquer/  
    |-- Conquer.sln  
    |-- Conquer/  
        |-- Cipher.cs  
        |-- Compress.cs  
        |-- Program.cs  
        |-- Everything else except bin/ and obj/
```

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

- Remplacez `prenom.nom` par votre propre login
- Le fichier `README` est obligatoire.
- Pas de dossiers `bin` ou `obj` dans le projet.
- Respectez scrupuleusement les prototypes demandés.
- Retirez tous les tests de votre code.
- **Le code doit compiler !**

README

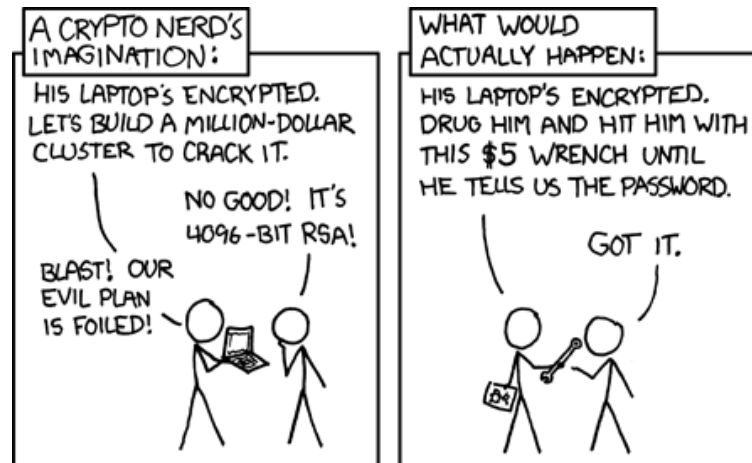
Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme un malus.

Table des matières

1	Introduction	3
1.1	Objectifs	3
2	Cours	3
2.1	Opérateurs bitwise	3
2.1.1	Opérateur ET	4
2.1.2	OR operator	4
2.1.3	Opérateur XOR	4
2.1.4	Opérateur Complément à un	5
2.1.5	Opérateur binaire de décalage à gauche	5
2.1.6	Opérateur binaire de décalage à droite	5
2.2	Manipulation d'images	5
2.2.1	Encodage des images	5
2.2.2	Format Bitmap (BMP) et modèle de coloration RVB	6
2.2.3	Le format Bitmap en C#	6
2.2.4	Structure Color	7
3	Exercices : Cipher	9
3.1	Exercice 1 : XOR-Cipher	9
3.2	Exercice 2 : ROR-Cipher	9
3.2.1	Exercice 2.1 : Rotate right	9
3.2.2	Exercice 2.2 : ROR-Cipher	10
3.3	Exercice 3 : ROTN	11
3.4	Exercice 4 : Analyse Fréquentielle	11
3.5	Exercice 5 : Vigenere Encode	12
3.6	Exercice 6 : Vigenere Decode	13
4	Exercices : Compression	13
4.1	Exercice 7 : String RLE	14
4.2	Image RLE	14
4.3	Exercice 8 : Chiffrement d'entier	14
4.3.1	Exercice 8.1 : Chiffrement Unaire	14
4.3.2	Exercice 8.2 : Chiffrement d'Elias Gamma	15
4.4	Exercice 9 : Image Compression	15
5	Bonus	16
5.1	Vrai binaire	16
5.2	Programme Shell	16
5.3	Plus de compression	17
5.4	Plus de chiffrement	17
5.5	Ressources additionnelles	17

1 Introduction

1.1 Objectifs



2 Cours

2.1 Opérateurs bitwise

Comme vous le savez en C# vous pouvez faire des opérations à l'aide d'opérateurs arithmétiques basiques ('+', '-', '*', '/', ...) Mais comme vous l'avez vu dans un précédent TP il existe également d'autres types d'opérateur, notamment les opérateurs Bitwise.

Les opérateurs bitwise fonctionnent de la même manière que les opérateurs arithmétiques, Ce sont aussi des opérateurs binaires (qui prennent 2 arguments [gauche et droite]) mais ce qui les caractérise est le fait qu'il permettent de manipuler les variables de manière plus précise en utilisant les bits.

Premièrement, il vous faut savoir comment assigner un nombre binaire ou hexadécimal à une variable en C#

Pour dire au compilateur que le nombre est binaire on utilise la syntaxe suivante

```
uint foo = 0b101010 // 42
// or
uint bar = 0B101010 // 42
```

De même en Hexadécimal, la syntaxe est la suivante :

```
uint foo = 0xDEADBEEF // 3735928559
// or
uint bar = 0XCAFEBABE // 3405691582
```

Maintenant que nous savons comment utiliser les binaires et hexadécimaux, affichons les : Pour ce faire on utilise `Convert.ToString()` comme suit :

```
int value = 0b101010;
string binary = Convert.ToString(value, 2);
Console.WriteLine(binary);
//101010
```

Le 2ème argument de `Convert.ToString()` est la base dans laquelle vous voulez afficher votre nombre.

En combinant cette méthode avec la méthode `Convert.ToInt32()` il devient possible d'exprimer n'importe quel nombre dans la base que l'on veut facilement.

```
String number = "100";  
int fromBase = 16;  
int toBase = 10;  
  
String result = Convert.ToString(Convert.ToInt32(number, fromBase), toBase);
```

Regardons maintenant comment marchent les opérateurs bitwise :

2.1.1 Opérateur ET

Pour faire un ET bitwise entre 2 nombres la syntaxe est la suivante :

```
int val = 0b11111111 & 0b00001111  
// 0b00001111
```

ET est utile pour mettre un bit à 0. Dans l'exemple précédent le nombre binaire 0b00001111 est appelé masque.

Attention

Ne confondez pas l'opérateur `&` (Bitwise ET) et l'opérateur `&&` (ET Logique)

2.1.2 OR operator

Pour faire un OU bitwise entre 2 nombres la syntaxe est la suivante :

```
int val = 0b00000000 | 0b00001000  
// 0b00001000
```

OU est utile pour mettre un certain bit à 1.

Attention

Ne confondez pas l'opérateur `|` (Bitwise OU) et l'opérateur `||` (OU Logique)

2.1.3 Opérateur XOR

Le OU Exclusif agit de manière similaire au OU Bitwise à la différence près qu'il renvoie faux quand les 2 opérandes sont à 1.

Pour faire un XOR entre 2 nombres la syntaxe est la suivante :

```
int val = 0b1100 ^ 0b1111  
// 0b0011
```

XOR est utile pour changer la valeur d'un certain bit.

Il existe 2 propriétés intéressantes du XOR :

- $A \wedge A = 0$
- $(A \wedge B \wedge A) = B$

2.1.4 Opérateur Complément à un

L'opérateur complément à 1 renverse tous les bits d'un nombre.

```
int val = ~0b1100
// 0b0011
```

2.1.5 Opérateur binaire de décalage à gauche

L'opérateur de décalage vers la gauche («) décale son premier opérande vers la gauche du nombre de bits spécifié par son deuxième opérande.

```
int val = 0b1100 << 2
// 0b110000
```

2.1.6 Opérateur binaire de décalage à droite

L'opérateur de décalage vers la droite (») décale son premier opérande vers la droite du nombre de bits spécifié par son deuxième opérande.

```
int val = 0b1100 >> 2
// 0b11
```

Enfin, faites attention à la priorité des opérateurs bitwise¹ entre eux mais également avec les autres opérateurs. De manière générale les opérateurs "classiques" ont la priorité sur les bitwise

Pour plus d'informations voici des liens vers la documentation Microsoft^{2 3}.

2.2 Manipulation d'images

Vous connaissez sûrement déjà quelques notions basiques de manipulation d'images. Si ce n'est pas le cas, vous en apprendrez durant cette semaine.

2.2.1 Encodage des images

Pour commencer, vous devez savoir qu'une image est avant tout un fichier, rien de plus. Il s'agit d'une zone de votre mémoire, contenant des 0 et des 1. Dans la plupart des formats, le fichier commence par un en-tête contenant des informations générales sur l'image (taille, format, date de création, etc.). Après cet en-tête se trouve l'encodage réel de l'image. L'encodage dépend du format choisi pour enregistrer le fichier, il en existe un grand nombre (PNG, JPG, BMP, SVG, PPM, etc.), le format est déterminé d'après le besoin de compresser l'image ou non, gérer la transparence ou non, et d'autres facteurs. Durant ce TP nous ne vous présenterons que le format BMP, qui est un des plus simples (mais qui est compatible avec d'autres comme le PNG ou le JPG via l'API C#).

1. <https://docs.microsoft.com/en-us/cpp/c-language/precedence-and-order-of-evaluation?view=vs-2017>

2. <https://docs.microsoft.com/en-us/cpp/c-language/c-bitwise-operators?view=vs-2017>

3. <https://docs.microsoft.com/en-us/cpp/c-language/bitwise-shift-operators?view=vs-2017>

2.2.2 Format Bitmap (BMP) et modèle de coloration RVB

Nous allons traiter le format Bitmap 24-bit, qui ne gère pas la transparence. Dans ce format, une image n'est rien de plus qu'une matrice de pixels, la largeur et la hauteur de l'image sont données dans l'en-tête du fichier. Ce format utilise le modèle de coloration RVB pour encoder chaque pixel :

Rouge, vert, bleu, abrégé en RVB ou en RGB, de l'anglais « Red, Green, Blue » est, des systèmes de codage informatique des couleurs, le plus proche du matériel. Les écrans d'ordinateurs reconstituent une couleur par synthèse additive à partir de trois couleurs primaires, un rouge, un vert et un bleu, formant sur l'écran une mosaïque trop petite pour être aperçue. Le codage RVB indique une valeur pour chacune de ces couleurs primaires. – Wikipedia⁴

Dans le format BMP, chaque pixel est encodé sur 24 bits, 3 canaux de 8 bits (1 octet chacun). Chaque canal (octet) peut prendre une valeur dans l'intervalle [0, 255]. Chaque canal représente une certaine quantité de coloration (représenté par un `int` entre 0 et 255), rouge, vert ou bleu respectivement.

ATTENTION

Cela peut paraître contre-intuitif mais si tous les canaux ont une valeur de 0 alors le pixel résultant est noir. De même le pixel est blanc quand tous les canaux sont à 255. Finalement le pixel est gris si tous les canaux ont la même couleur.

2.2.3 Le format Bitmap en C#

Dans la bibliothèque .NET, une classe a été implémentée pour simplifier la création, modification et lecture d'images Bitmap. Le nom de cette classe est étonnamment `Bitmap`⁵, n'hésitez pas à visiter la page MSDN de cette classe pour en apprendre plus à son propos. Cette classe vous permet de créer des objets Bitmap depuis des fichiers (JPG, PNG, BMP, et d'autres), de les éditer, de créer de nouvelles images à partir de ses dimensions, facilement accéder aux propriétés du fichier (largeur, hauteur, résolution, etc.), et enregistrer des objets Bitmap dans des fichiers. Bien entendu, elle vous permet aussi d'accéder et de modifier des pixels en connaissant leur position dans la matrice. Les pixels sont stockés en tant qu'objets `Color`, cette structure sera expliquée après ce court exemple d'utilisation de la classe `Bitmap` :

4. https://fr.wikipedia.org/wiki/Rouge_vert_bleu

5. [https://msdn.microsoft.com/en-us/library/system.drawing.bitmap\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.110).aspx)

```
// Créer une image depuis un fichier
Bitmap img = new Bitmap("path/to/file.png");

// Créer un magnifique fond d'écran HD vide
Bitmap empty = new Bitmap(1920, 1080);

// Sauvegarder l'image dans un autre fichier, dans un autre format
img.Save("new/path/to/file.jpg");

// Accéder facilement à la largeur et la hauteur
if (img.Width == img.Height)
    Console.WriteLine("This picture is a square");

// Visiter tous les pixels, en remplaçant les blancs par des noirs
for (int i = 0; i < img.Width; ++i)
{
    for (int j = 0; j < img.Height; ++j)
    {
        if (img.GetPixel(i, j) == Color.White)
            img.SetPixel(i, j, Color.Black);
    }
}
```

Note : Pour utiliser la classe `Bitmap` et la structure `Color`, vous devrez ajouter une référence à `System.Drawing.dll` avant de compiler.

2.2.4 Structure Color

La structure `Color`⁶ vous permet de manipuler des couleurs et d'en créer des nouvelles depuis des valeurs RVB. Chaque objet `Color` possède 3 propriétés, `R`, `G` (pour « Green ») et `B`, représentant les 3 canaux de couleurs du modèle de coloration RVB. Ces champs sont des entiers qui ne peuvent avoir une valeur que dans l'intervalle $[0, 255]$. Vous pouvez uniquement lire ces valeurs. Si vous souhaitez avoir une couleur avec des différentes valeurs, vous devez créer un nouvel objet, en utilisant la méthode statique `Color.FromArgb` (pas un constructeur, uniquement une méthode générant un `Color`). Cette structure possède aussi certains champs, représentant des couleurs spécifiques. Lisez l'exemple ci-dessous pour une petite utilisation de la structure, ou lisez la page MSDN correspondante :

6. [https://msdn.microsoft.com/en-us/library/system.drawing.color\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.drawing.color(v=vs.110).aspx)

```
// Créer une couleur depuis une existante
Color choco = Color.Chocolate;

// Récupérer les valeurs R, G et B et augmenter la luminosité
int r = choco.R + 50;
int g = choco.G + 50;
int b = choco.B + 50;

// Vérifier que nous n'avons pas excédé la valeur limite de 255
r = r > 255 ? 255 : r;
g = g > 255 ? 255 : g;
b = b > 255 ? 255 : b;

// Créer une nouvelle couleur depuis les nouvelles valeurs RVB
Color brightChoco = Color.FromArgb(r, g, b);
```

Dans cet exemple, vous pouvez voir comment créer une `Color` (vous pouvez aussi en obtenir une d'un appel à `Bitmap.GetPixel`), augmenter la luminosité, et sauvegarder le résultat dans un nouvel objet couleur.

Pour utiliser la classe `Bitmap`, il faut rajouter la référence à `System.Drawing` à votre projet. Sur Rider, il faut faire un click droit sur le fichier du projet ou le fichier "references" et choisir "add reference".

3 Exercices : Cipher

Note

Tous les exercices sont indépendants, ne restez pas bloqué sur un exercice. Si vous avez du mal sur un exercice, passez au suivant.

Attention

Les exercices 1 à 5 sont à faire dans le fichier Cipher.cs.

Cipher.cs

3.1 Exercice 1 : XOR-Cipher

Ce chiffrement est très simple : Pour chaque caractère de la chaîne d'entrée, appliquez l'opérateur XOR avec la clé également donnée en argument.

Pour rappel, voici la table de vérité de l'opérateur XOR :

Inputs		Outputs
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

```
public static string xor_cipher(string msg, char key);
```

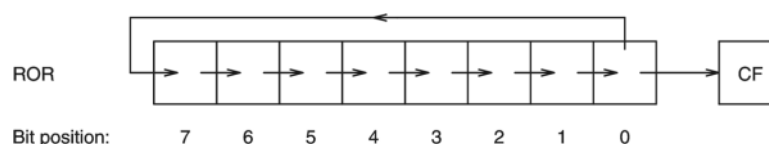
Exemple :

```
string message = "Hello world!";  
char key = (char)0b00011111;  
string output = Cipher.xor_cipher(message, key);  
Console.WriteLine("Ciphred message: " + output);  
//Ciphred message: Wzssp?hpms{>  
  
output = Cipher.xor_cipher(output, key);  
Console.WriteLine("Deciphred message: " + output);  
//Deciphred message: Hello world!
```

3.2 Exercise 2 : ROR-Cipher

3.2.1 Exercice 2.1 : Rotate right

La fonction **RO**ate **RI**ght (**ror**) prends un caractère et décale tous ses bits N bits vers la droite. Le dernier bit (LSB) revient toujours a la position du premier (MSB). Ci-dessous un schéma du fonctionnement de ror (Ne faites pas attention au bloc CF).



Tip

En C# le type `char` a une taille de 2 bytes (16 bits) car celui-ci supporte les représentations des caractères UTF-16 (Ce n'est pas le cas pour tous les langages, en C par exemple les `char` ont une taille d'un byte).

En réalité `ror` est une [instruction assembleur](#).

```
public static char ror(char c, int n);
```

`ror` doit aussi pouvoir gérer les nombres négatifs.

Exemple :

```
char c = 'a';
Console.WriteLine(Convert.ToString(c, 2).PadLeft(16, '0'));
//00000000001100001
c = Cipher.ror(c, 0);
Console.WriteLine(Convert.ToString(c, 2).PadLeft(16, '0'));
//00000000001100001
c = Cipher.ror(c, 3);
Console.WriteLine(Convert.ToString(c, 2).PadLeft(16, '0'));
//00100000000001100
c = Cipher.ror(c, -3);
Console.WriteLine(Convert.ToString(c, 2).PadLeft(16, '0'));
//00000000001100001
c = Cipher.ror(c, -32);
Console.WriteLine(Convert.ToString(c, 2).PadLeft(16, '0'));
//00000000001100001
```

3.2.2 Exercice 2.2 : ROR-Cipher

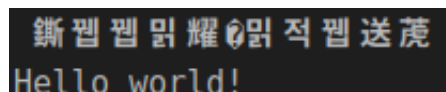
Maintenant que vous avez la fonction `ror`, nous pouvons à présent coder le chiffrement correspondant. Le but de cette fonction est d'appliquer la fonction `ror` précédente sur tous les caractères d'une chaîne de caractères.

```
public static string ror_cipher(string msg, int n);
```

```
string input = "Hello world!";
int nb = -42;
string res = Cipher.ror_cipher(input, nb);
Console.WriteLine(res);

res = Cipher.ror_cipher(res, -nb);
Console.WriteLine(res);
// See below
```

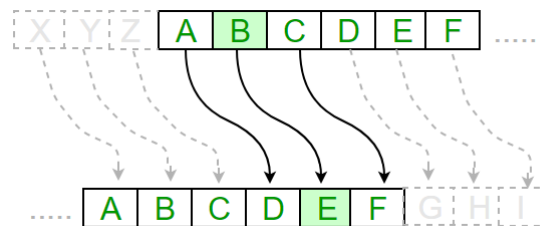
Voici le resultat que vous devriez obtenir :



Hello world!

3.3 Exercice 3 : ROTN

Le chiffrement ROTN (également appelé chiffre de César) est la fonction de chiffrement la plus basique qui vient à l'esprit quand on parle de chiffrement. Pour chaque lettre de votre message incrémentez la de N positions dans l'alphabet. Si la nouvelle position dépasse l'alphabet, on revient au début de l'alphabet. Pour une clé égale à 3 on aurait : A devient D, B devient E [...], W devient Z, X devient A ...



Pour plus d'informations sur le fonctionnement du chiffre de César voici l'article Wikipédia correspondant : [ROT₁₃](#)

```
public static string rotn(string msg, int n);
```

La fonction ne modifie que les lettres majuscules et minuscules. Tous les autres caractères doivent rester les mêmes et la chaîne de caractères de sortie doit avoir la même casse que celle d'entrée, c-à-d : ROT_1 de "ab;C\$D" devient "bc;D\$E".

Exemple :

```
string str = "Look at that! They call this a castle?";
int k = -42;
string m = Cipher.rotN(str, k);
Console.WriteLine(m);
//Vyyu kd drkd! Droi mkvv drsc k mkcdvo?

m = Cipher.rotN(m, -k);
Console.WriteLine(m);
//Look at that! They call this a castle?
```

3.4 Exercice 4 : Analyse Fréquentielle

Nous savons chiffrer un message à l'aide du chiffre de César nous voulons maintenant casser un message chiffré en ROT_N , pour ce faire nous allons faire de l'analyse fréquentielle.

L'analyse fréquentielle consiste à examiner la fréquence des lettres employées dans un message chiffré. Cette méthode est fréquemment utilisée pour décoder des messages chiffrés par substitution, comme le chiffre de César (Pourvu que le message soit assez long).

L'analyse fréquentielle est basée sur le fait que, dans chaque langue, certaines lettres ou combinaisons de lettres apparaissent avec une certaine fréquence. Par exemple, en français, le e est la lettre la plus utilisée, suivie du a et du s. Donc si dans un message chiffré c'est la lettre 'a' qui apparaît le plus souvent nous pouvons (dans une certaine mesure) déduire qu'il faut remplacer tous les 'a' par des 'e' pour commencer à obtenir le message original.

Pour plus d'informations concernant l'analyse fréquentielle vous pouvez consulter ce lien : dCode.fr/analyse-frequence.

Votre but est de renvoyer les caractères du message d'entrée dans un tableau dans leur ordre croissant de fréquence. (La lettre la moins fréquente sera en premier). Les caractères imprimés ne doivent être que des lettres minuscules. (La chaîne de caractère "AAAa" affichera "a"). Notez que nous testerons votre fonction uniquement avec des caractères ASCII compris entre 0 et 255. Aussi, si 2 lettres ont la même fréquence, leur ordre n'importe pas.

```
public static char[] freq(string msg);
```

Exemple :

```
Cipher.freq("aaaaAAAA bbbbBb c");  
//cba  
  
Cipher.freq("c41ciiIfErrrrr123");  
//fecir  
  
Cipher.rotn("This is a pretty long string, the freq" +  
            "function should be able to recognize which" +  
            "characters have been permuted. The last" +  
            "character printed will most likely be an" +  
            " f which corresponds to e + 1", 1);  
  
Cipher.freq(test);  
//awrlznhguxqcemopbtdjisuf
```

Tip

Vous pouvez utiliser la fonction `Array.Sort(key, value)` pour trier un tableau en fonction des valeurs d'un autre.

Vous pouvez utiliser `Char.ToLower()` pour convertir un caractère en majuscule en minuscule.

3.5 Exercice 5 : Vigenere Encode

Le chiffre de Vigenère chiffre un message grâce à une clé (une chaîne de caractère). Voici un extrait de dCode.fr sur le fonctionnement du chiffre de Vigenère : "Le chiffrement consiste à additionner la clé au texte clair. Le calcul est effectué lettre par lettre (l'addition de lettre est en fait réalisée par des nombres, les valeurs des lettres sont ajoutées). Le résultat est donné modulo 26 : si le résultat est supérieur ou égal à 26, soustraire 26 au résultat (où 26 est la longueur de l'alphabet). Pour faire correspondre la longueur du texte à la clé, celle-ci est répétée à l'infini : CLECLECLEC..."

Exemple avec la phrase "Got it memorized?" et la clé "key" :

```
Plaintext:  GOT IT MEMORIZED ?  
Key:       KEY KE YKEYKEYKE  
          -----  
Ciphertext: QSR SX KOQMBMXOH ?
```

- $G + K = Q$
- $O + E = S$
- $T + Y = R$
- ...

Le message d'entrée peut contenir des lettres minuscules mais vous devrez les considérer comme des lettres majuscules : Convertissez les lettres minuscules en lettres majuscules avant de faire des calculs avec. Le résultat ne doit contenir que des lettres majuscules ; Les autres caractères doivent rester inchangés.

Vous pourrez trouver plus d'information sur le chiffrement de Vigenère sur la page Wikipedia⁷ correspondante.

```
public static string vigenere_encode(string msg, string key);
```

```
string message = "Here's another curse: may all your bacon burn.";
string key = "Calcifer";
string output = Cipher.vigenere_encode(message, key);
Console.WriteLine(output);
//JECG'A FRFVHPT KZVJG: MLA IQP PQUC DIHSE DUCP.
```

3.6 Exercice 6 : Vigenere Decode

Cet exercice est le même que le précédent mais le processus est inversé pour pouvoir retrouver le message d'origine.

```
public static string vigenere_decode(string msg, string key);
```

```
string message = "JecG'a FrFvhpT kZvJg: mla iQp PQUC DihSE DucP";
string key = "Calcifer";
string output = Cipher.vigenere_decode(message, key);
Console.WriteLine(output);
//HERE'S ANOTHER CURSE: MAY ALL YOUR BACON BURN
```

Ciphertext: ZJSBG GMBS HC NWIXVF QETOOMFB DW...

Key: OVER

Plaintext: ?

4 Exercices : Compression

Compress.cs

Attention

Les exercices à partir du 7ème doivent être codés dans la classe Compress.cs

Dans cette partie, nous allons nous intéresser à l'une des techniques de compression les plus connues, le Run Length Encoding ou RLE.

7. https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher

4.1 Exercise 7 : String RLE

Vous devez implémenter les deux fonctions suivantes :

```
public static string rle_encode(string msg);
```

```
public static string rle_decode(string msg);
```

L'implémentation du RLE pour les `strings` est très simple. Il suffit juste de compter le nombre de caractères égaux consécutifs et de les remplacer par cette valeur suivi du caractère. Les caractères seuls doivent quand même être précédés par un '1' et s'il y a plus de 9 caractères égaux consécutifs, il faut les séparer en 2 groupes (voire les exemples ci-dessous). Ces conditions permettent de supprimer toute ambiguïté dans le chiffrement, déchiffrement.

```
Compress.rle_encode("aaaabccccccccc"); // "4a1b9c1c"  
Compress.rle_decode("1a9b3b4c"); // "abbbbbbbbbbbcccc"
```

4.2 Image RLE

Comme vous l'avez peut-être remarqué, cette méthode n'est pas très efficace sur des `strings` usuels représentant du texte car il est peu probable que l'on ait beaucoup de caractères semblables à la suite. Ainsi, elle doit être combinée avec d'autres techniques (la transformation de Burrows-Wheeler par exemple) pour pouvoir être vraiment efficace. Dans cette partie, nous allons voir une application de RLE sur des images qui peut donner de bons résultats.

4.3 Exercise 8 : Chiffrement d'entier

Attention

Dans les parties suivantes, nous considérerons le binaire comme un `string` composé de 0 et de 1 pour simplifier les fonctions. La conversion vers du vrai binaire sera laissée comme un bonus mais est grandement recommandée comme entraînement.

Tout d'abord, nous devons trouver le moyen d'écrire et de lire des nombres avec une précision arbitraire dans un fichier. Pour cela nous allons voir deux méthodes :

Chiffrement d'entier

Cela peut paraître simple mais si nous sauvegardions le nombre directement sous sa notation binaire, il serait impossible de délimiter chaque nombre durant la lecture.

4.3.1 Exercise 8.1 : Chiffrement Unaire

Le code unaire est vraiment simple, pour chiffrer un nombre `n`, nous allons juste concaténer `n` fois des '0' et un '1' terminal. Comme précédemment, vous devez implémenter le chiffrement et le déchiffrement.

```
public static string unary_encode(int n);
```

```
public static int unary_decode(string msg);
```

```
Compress.unary_encode(5); // 000001  
Compress.unary_decode("01") // 1
```

4.3.2 Exercice 8.2 : Chiffrement d'Elias Gamma

La deuxième méthode est basée sur le code unaire mais est bien plus efficace car il n'y a plus besoin d'utiliser $n+1$ bits pour encoder n . Le but est de calculer :

$$U = \lfloor \log_2(n) \rfloor$$

U correspond au nombre de bits moins un de la représentation binaire du nombre. Le but du chiffrement d'Elias Gamma (ou juste Gamma) est de concaténer la représentation binaire du nombre au chiffrement unaire de U .

```
public static string gamma_encode(int n);
```

```
public static int gamma_decode(string msg);
```

Comme exemple, nous allons encoder 4. La représentation binaire de 4 est 100 ce qui requiert 3 bits ainsi $U = 2$. Le résultat est donc "001" (U en unaire) + "00" (les derniers U bits de 4).

Tip

Lors du déchiffrement, retrouver U permet de savoir combien de bits il faut lire pour récupérer la deuxième partie.

4.4 Exercice 9 : Image Compression

A première vue, une image peut sembler poser le même problème qu'un `string` (2 caractères consécutifs vont difficilement avoir la même valeur même si leur couleur sera sans doute proche). Ainsi, nous allons seulement utiliser des images en noir et blanc qui ont 2 valeurs possibles (noir = 0 et blanc = 1). Tout d'abord, vous devez implémenter la binarisation de l'image. Vous devez sauvegarder le résultat dans un fichier avec comme nom le deuxième argument.

```
public static void img_bin(string pathin, string pathout);
```

Tip

On considérera qu'un pixel doit être blanc si la moyenne des ses canaux est strictement supérieure à 127

```
Compress.img_bin("lena.jpg", "lena.bw"); //Ecrit dans lena.bw
```

Nous allons adapter un peu le RLE précédent pour compresser l'image. Le `string` commencera avec la largeur de l'image suivi de la hauteur tout les 2 chiffrés en Gamma. Le prochain bit représentera la valeur du pixel en haut à gauche et sera suivi d'entiers chiffrés en gamma représentant le nombre de pixels consécutifs (comme nous avons seulement deux valeurs, connaître seulement la première occurrence est suffisant).

Parcours

Vous devez utiliser un parcours ligne par ligne de gauche à droite (l'ordre usuel de parcours de matrices).

Comme exemple, nous allons encoder une image de 5x2 contenant 6 pixels blancs puis 4 pixels noirs. Cela nous donne "00101" (5 chiffré en Gamma) + "010" (2 chiffré en gamma) + "1" (valeur du premier pixel) + "00110" (6 chiffré en gamma) + "00100" (4 chiffré en gamma).

```
public static void img_compress(string pathin, string pathout);
```

```
public static void img_decompress(string path, string pathout);
```

5 Bonus

5.1 Vrai binaire

Implémenter les 2 fonctions suivantes :

```
public static void real_img_compress(string pathin, string pathout);
```

```
public static void real_img_decompress(string pathin, string pathout);
```

Elles devraient avoir le même comportement que précédemment mais au lieu d'écrire le **string** directement dans un fichier, elles doivent utiliser la représentation binaire. Globalement, cela devrait diviser par 8 la taille du résultat final. Il faudra cependant ajouter un certain nombre de bits pour avoir un résultat divisible par 8 (voir TIP ci-dessous). Ainsi vous devez rajouter le nombre de bits à ignorer à la fin du fichier devant la largeur de l'image. Comme cette valeur, sera forcément inférieure à 8, vous devez représenter le nombre en utilisant trois bits (0 = "000").

Tip

Il est impossible d'écrire moins d'un **byte** à la fois dans un fichier en C# donc il vous faudra bien utiliser les bitwise. Une façon est de construire des **Char** en regroupant le string 8 par 8 puis de les écrire dans le fichier.

Tip

Sur Linux, il est possible de voir la représentation binaire d'un fichier avec :

```
xxd -b [file]
```

5.2 Programme Shell

```
public static void shell(string[] args);
```

Pour tester votre Shell, vous devriez avoir un main semblable à :

```
public static void Main(string[] args)
{
    shell(args);
}
```



```
./main -h
> Options:
> -rotn n message
> -vigenere key message
> -rle message
> -binarize image
> -compress image

./main "Hello"
> Hello

./main -rotn
> Error: missing arguments
> Usage: ./main -rotn n message

./main -rotn 1 "Hello"
> Ifmmp

./main -vigenere "abc" "Hello world"
> [...]
```

5.3 Plus de compression

Vous pouvez essayer d'implémenter d'autres algorithmes de compression. Par exemple :

- Faire une transformation de Burrows-Wheeler sur le **string** avant le RLE
- Compresser des images en couleurs.

5.4 Plus de chiffrement

Vous pouvez aussi essayer d'implémenter une des méthodes de chiffrement suivantes : Code Morse⁸, Rail fence⁹, Two-square¹⁰, ...)

5.5 Ressources additionnelles

Le site web ctf.lse.epita.fr/ offre de bons exercices dont certains sont un peu reliés au contenu de ce tp. Nous vous conseillons de surtout regarder les sections suivantes :

- **CrackMe** : Analyse de fichiers binaires pour retrouver un mot de passe.
- **Forensics** : Recherche d'un message particulier dans des fichiers.
- **Misc (Subject 201X)** : De bons exercices pour apprendre les différents types de fichier.
- **Cryptography** : Pour ceux qui ont survécu à l'AFIT.

Tip

Si vous êtes intéressés, renseignez-vous sur la commande **file** (man file). Il s'agit du minimum si vous voulez commencer à résoudre les challenges les plus simples.

Voici un petit [quizz](#) non-obligatoire pour vous évaluer sur les notions vues dans ce TP

8. https://en.wikipedia.org/wiki/Morse_code
9. https://en.wikipedia.org/wiki/Rail_fence_cipher
10. https://en.wikipedia.org/wiki/Two-square_cipher

I see no point in coding if it can't be beautiful.