CS-107 : Autoévaluation
Cryptographie & Stéganographie

Hamza REMMAL, Rafael PIRES & Jamila SAM

-

version 2.0.2

Contents

Ce document utilise des couleurs et contient des liens cliquables. Il est préférable de le visualiser en format numérique.

1 Présentation

Protéger les données et systèmes numériques est un enjeu crucial de notre époque. Le domaine de la cryptographie étudie des techniques pour sécuriser nos données et pour ce faire, fournit ce que l'on appelle des "cryptosystèmes" qui permettent de les coder et décoder.

La stéganographie quant à elle est un art de la dissimulation : il s'agit de cacher une information, par exemple une image ou un texte, dans un support comme une autre image ou un texte. Le but est que l'on ne puisse pas voir que le support cache le message. Contrairement à la cryptographie, qui rend l'information incompréhensible, mais toujours apparente[1], la stéganographie vise à ce que l'information elle-même ne soit même pas détectée. Elle est par exemple utilisée dans des situations où les messages que l'on transmet sont surveillés, comme sous une dictature, ou par des groupes terroristes[2]. Plus légalement, elle est utilisée pour le tatouage numérique (ou *digital watermarking*) qui permet de marquer de façon invisible (ou visible) un fichier, afin de pouvoir retracer sa provenance ; par exemple en cas de fuite de ce fichier.

Le but de cette auto-évaluation est de vous donner une introduction basique à certains cryptosystèmes classiques ainsi que de vous faire découvrir le domaine de la stéganographie. Vous combinerez en fait les deux en cachant dans des images des messages cryptés !

2 Structure et code fourni

2.1 Structure

Le projet est divisé en trois étapes :

- L'implémentation d'utilitaires : lors de cette première étape, vous implémenterez quelques méthodes dont le rôle sera de vous aider à implémenter le reste du projet plus aisément ;
- Les algorithmes cryptographiques : dans un second temps, vous implémenterez quelques algorithmes cryptographiques tels que l'algorithme de Caesar ou bien l'encodage CBC ;
- Les algorithmes stéganographiques : Pour finir votre projet, vous aurez à mettre en place deux algorithmes, un facile et autre un peu plus compliqué, qui vous permettra de cacher des données dans des images.

Les étapes 2 et 3 peuvent être codées de manière indépendante.

2.2 Code fourni

Pour accéder au code fourni, suivez avec attention les instructions données dans la description générale du projet.

[1] On sait que l'information existe et que l'on veut nous empêcher de la lire

[2] Ce qui intéresse d'ailleurs les sciences forensiques (voir les références en fin de document)

Vous trouverez ici la documentation javadoc du projet. Prenez connaissance du matériel fourni au travers de cette documentation (le code fourni lui-même n'est pas forcément à votre portée et il n'est pas attendu de vous que vous en compreniez les détails).

- Tout le code obligatoire devra être produit dans les fichiers :
  - `Bit.java`,
  - `Text.java`,
  - `Image.java`,
  - `Encrypt.java`,
  - `Decrypt.java`,
  - `ImageSteganography.java`,
  - `TextSteganography.java`
- Les entêtes des méthodes à implémenter sont fournies et **ne doivent pas être modifiées**.
- Le fichier fourni `SignatureChecks.java` donne l'ensemble des signatures à ne pas changer. Il sert notamment d'outil de contrôle lors des soumissions. Il permettra de faire appel à toutes les méthodes requises **sans en tester le fonctionnement**. Vérifiez que ce programme **compile** bien, avant de soumettre.
- La notation `UneClasse::uneMethode` se lit «la méthode `uneMethode` de la classe `UneClasse`»

2.3 Code avancé

Certaines fonctions nécessaires au développement du projet, telles que celles liées à la manipulation de fichiers ou d'images, sont trop avancées pour ce cours. Elles sont donc codées pour vous. À cet effet, un fichier utilitaire `Helper.java` vous est fourni contenant toutes fonctions dont l'utilité a été jugée nécessaire. Ce dernier contient :

- Méthodes pour la **manipulation d'images**
  - La méthode `public static int[][] readImage(String path)` qui prend comme paramètre le chemin d'accès vers l'image à lire. Cette méthode retourne un tableau bidimensionnel d'entier (`int[][]`) représentant une image au format *ARGB*.
  - La méthode `public static void writeImage(String path, int[][] image)` qui prend comme paramètres : le chemin d'accès où stocker l'image et un tableau bidimensionnel représentant l'image au format *ARGB*
- Méthodes pour la **manipulation de fichiers**
  - La méthode `public static byte[] read(String path)` permet de lire le contenu d'un fichier dont le chemin d'accès est donné en argument
  - La méthode `public static void write(String path, byte[] content)` qui permet d'écrire le contenu de *content* dans un fichier en suivant le chemin d'accès donné par *path*
- Méthodes pour l'**affichage**
  - La méthode `public static void show(int[][] image, String title)` qui crée une nouvelle fenêtre et affiche l'image en paramètre. Le titre de la fenêtre correspond à la valeur du deuxième paramètre.

  - La méthode `public static void dialog(String title, String message)`, avec un comportement similaire à `Helper::show`, permet d'afficher un texte dans une nouvelle fenêtre graphique.

- La méthode `public static byte[] generateRandomBytes(int length)` vous permettra de générer aléatoirement une séquence de `byte` dont la longueur dépend du paramètre *length*
- Méthode pour la **gestion d'erreur**.
  - La méthode `public static <T> T fail(String fmt, Object ... args)` permet de terminer l'exécution du programme. Elle prend comme paramètres le format du message `fmt`, ainsi que les *objets* nécessaires pour formatter le message final. (Le formatage fonctionne de manière tout à fait similaire à `String::format`)

L'utilisation de ces méthodes dans les contextes appropriés sera bien sûre décrite dans l'énoncé ou donnée en exemple dans le programme principal fourni. Pour le moment, il s'agit uniquement de prendre note de leur existence.

2.4 Tests

Important :

La vérification du comportement correct de votre programme vous incombe. Le fichier fourni `Main.java`, partiellement rédigé, vous servira à tester vos développements de façon simple. **Il vous revient la tâche de compléter `Main.java` en y invoquant vos méthodes de manière adéquate pour vérifier l'absence d'erreur dans votre programme.**

Pour vous fournir un feedback sur votre projet, nous utiliserons des tests automatisés, qui passeront des données d'entrée générées aléatoirement aux différentes fonctions de votre programme. **Il y aura donc aussi des tests vérifiant comment sont gérés les *cas particuliers*.** Ainsi, il est important que votre programme traite correctement toute donnée d'entrée *valide*.

Pour ce qui est de la gestion des cas d'erreurs, il est d'usage de tester les paramètres d'entrée des méthodes; par exemple, vérifier qu'un tableau n'est pas nul, et/ou est de la bonne dimension, etc. Ces tests facilitent généralement le débogage, et vous aident à raisonner quant au comportement d'une fonction.

Nous supposerons que les arguments des fonctions sont valides (sauf exceptions explicitement mentionnées). Pour garantir cette hypothèse, il faudra utiliser les assertions Java[3]. Une assertion s'écrit sous la forme : `assert expr;` avec `expr` une expression booléenne. Si l'expression est fausse, alors le programme lance une erreur et s'arrête, sinon il continue normalement. Par exemple, pour vérifier qu'un paramètre de méthode nommé `key` n'est pas `null`, vous pouvez écrire `assert key != null;` au début du corps de la méthode (les assertions n'ont pas forcément besoin de figurer au début du corps).

Les assertions doivent être activées pour fonctionner. Ceci se fait en lançant le programme avec l'option `-ea`:
- pour IntelliJ
- pour Eclipse.

Enfin, Nous vous recommandons vivement de vous familiariser d'emblée avec l'usage du dévermineur («debugger»). Essayez dès le test de votre première fonction de placer des points d'arrêt pour examiner les valeurs des variables impliquées.

[3]Nous aurons l'occasion d'y revenir en détail, mais leur utilisation est assez intuitive pour que nous puissions déjà y recourir

Voici un résumé des consignes/indications principales à respecter pour le codage du projet :

- Les paramètres des méthodes seront considérés comme exempts d'erreur, sauf mention explicite du contraire.
- Les entêtes des méthodes fournies doivent rester inchangés : le fichier `SignatureCheck.java` ne doit donc pas comporter de fautes de **compilation** pour garantir la compatibilité avec le correcteur.
- En dehors des méthodes imposées, libre à vous de définir toute méthode supplémentaire qui vous semble pertinente. **Modularisez et tentez de produire un code propre !**
- La vérification du comportement correct de votre programme vous incombe. Néanmoins, nous fournissons le fichier `Main.java`, illustrant comment invoquer vos méthodes pour les tester. Les exemples de tests ainsi fournis sont **non exhaustifs** et vous êtes autorisé.e.s/encouragé.e.s à modifier `Main.java` pour faire davantage de vérifications. Pensez à exécuter les tests fournis un par un pour "vérifier" le comportement de votre code.
- Votre code devra respecter les conventions usuelles de nommage.
- Vous éviterez l'utilisation de l'instruction `System.exit()`.
- Le projet sera codé sans le recours à des librairies externes. Si vous avez des doutes sur l'utilisation de telle ou telle librairie, posez-nous la question et surtout, faites attention aux alternatives que IntelliJ (ou Eclipse) vous propose d'importer sur votre machine.
- Votre projet **ne doit pas être stocké sur un dépôt public** (de type github public). Vous pouvez utiliser GitLab fourni par l'EPFL, comme montré en cours et il est vivement recommandé d'apprendre à collaborer en utilisant `git`.
- Utilisez le forum Ed. Avant de poser votre question, vérifiez qu'il n'y a pas de questions similaires. Cela permet de faciliter l'utilisation du forum et de ne pas se perdre dans les questions. Respectez aussi la structure du forum : **Utilisez la catégorie Mini-projet 1**.

3 Tâche 1 : Implementation de méthodes utilitaires

Pour vous faciliter l'implémentation des algorithmes cryptographiques et stéganographiques, nous dédierons cette première tâche du projet à l'implémentation de trois types (classes) d'utilitaires. Les premières méthodes, codées dans le fichier `Bit.java`, seront relatives à la manipulation de *bits*. Les méthodes de la deuxième classe, implémentées dans le fichier `Text.java`, permettront de manipuler les chaînes de caractères. Vous implémenterez les méthodes de `Image.java` qui vous permettront de manipuler des images plus aisément. Le rôle de ces divers utilitaires deviendra plus clair au fûr et à mesure que votre projet prend forme.

> Il est conseillé de tester au fur et à mesure ce que vous développez. Commencez par prendre connaissance des conseils de la section Tests (en section 3.4).

3.1 Manipulation de bits : fichier `Bit.java`

Que ce soit pour la partie *cryptographie* ou *stéganographie*, la manipulation de bits représentera une partie fondamentale de votre implémentation.

> Vous utiliserez ici les opérateurs binaires et coderez dans le fichier `Bit.java`.

Indication: pour vérifier plus facilement vos résultats, vous pouvez vous aider de convertisseurs décimal/binaire en complément à 2, tel que celui disponible ici.

3.1.1 Méthode `getXthBit`

Pour commencer, il vous est demandé d'implémenter la méthode `Bit::getXthBit`:

```
public static boolean getXthBit(int value, int pos)
```

Cette méthode doit permettre d'extraire le bit à la position `pos` dans la séquence de bits formant un entier `value`. Les positions sont numérotées de droite à gauche (le bit le moins significatif est à la position zéro). Si la valeur du bit vaut `1`, cette méthode devra retourner la valeur boolean `true`, sinon, ce sera la valeur `false`.

Voici un exemple où l'on extrait chacun des 32 bits du littéral `-12` :

```
int value = -12;
// Integer.SIZE donne la taille des entiers (en terme de nombre de bits)
boolean[] computed = new boolean[Integer.SIZE];
for (int i = 0; i < Integer.SIZE; ++i) {
    computed[i] = Bit.getXthBit(value, i); }
```

Le contenu du tableau `computed`, après l'exécution de ce code, sera le suivant :

```
{ false, false, true, false, true, true, true, true, true, true,
  true, true, true, true, true, true, true, true, true, true, true,
  true, true, true, true, true, true, true, true, true, true, true }
```

Rappelons que pour chaque méthode que vous aurez à implémenter, il vous est demandé de vérifier au moyen d'assertions que les données en entrées sont valides. Il faudra donc typiquement penser aux *cas limites* pour formuler ces assertions. Par exemple, pour formuler des assertions relatives à `Bit::getXthBit`, il faut se poser des questions telles que « Que doit-il se passer si `Bit::getXthBit` est invoquée avec les paramètres `(value = 0, bit = 100)` ?

3.1.2 Méthode `getLSB`

Maintenant que vous pouvez extraire tous les bits d'un entier, nous nous intéresserons à un bit particulier, le *LSB (least significant bit)*. L'accès au bit de poids le plus faible, *least significant bit* en anglais, sera souvent utile. Dès lors, il vous est demandé d'implémenter la méthode `Bit::getLSB` avec la signature :

```
public static boolean getLSB(int value)
```

Cette dernière vous permettra de savoir si le *LSB* d'un entier vaut *1* ou bien *0*. En utilisant la même convention citée plus haut, nous représenterons la valeur entière `1` par le booléen `true` ainsi que la valeur entière `0` par le booléen `false`.

Par exemple, les appels à `Bit::getLSB` sur les valeurs `-12`, `12`, `0` et 255 devraient retourner respectivement `false`, `false`, `false` et `true`.

3.1.3 Méthodes `embedInXthBit` et `embedInLSB`

Extraire des bits d'un entier n'est pas la seule opération nécessaire au développement du projet. L'opération réciproque, la "modification" de bits donnés, s'avérera tout aussi utile. Pour cette étape, il vous est demandé d'implémenter les méthodes `Bit::embedInXthBit` et `Bit::embedInLSB`, réciproques des méthodes `Bit::getXthBit` et `Bit::getLSB` respectivement.

La méthode `Bit::embedInXthBit` prendra comme entrée un entier, ainsi que la position du bit à "modifier" et la nouvelle valeur à donner à ce dernier. Comme valeur de retour, cette dernière devra retourner le même entier, mais avec le bit dans la position donnée modifié. Ceci peut être résumé à la signature suivante :

```
public static int embedInXthBit(int value, boolean m, int pos)
```

Comme expliqué plus haut, les paramètres de cette méthode sont :

- l'entier à modifier (`value`);
- la nouvelle valeur que prendra le bit sélectionné (`m`) dans l'entier retourné ; La valeur `true` pour que le bit prenne la valeur `1` et `false` pour `0`
- la position du bit à modifier (`pos`).

Par exemple :

```
embedInXthBit(-12, false, 2); // retourne l'entier -16
```

La seconde méthode, réciproque de la méthode `getLSB`, permettra de façon analogue de retourner un entier valant celui en paramètre où le bit de poids faible aura été changé :

```
public static int embedInLSB(int value, boolean m)
```

Par exemple:

```
embedInLSB(-12, true); // retourne l'entier -11
```

3.1.4 Décomposition d'un byte

Pour finir cette première étape de votre projet, il vous est demandé d'implémenter une première méthode (`Bit::toBitArray`) qui vous permettra de décomposer un entier de type `byte` en un tableau de `boolean` représentant les bits qui le forment. Vous implémenterez aussi la méthode réciproque de `Bit::toBitArray`, qui permettra de reformer un entier de type `byte` à partir d'un tableau de booléan (`boolean[]`).

Les signatures des méthodes seront les suivantes:

```
public static boolean[] toBitArray(byte value)
```

```
public static byte toByte(boolean[] bitArray)
```

Par exemple:

```
Bit.toBitArray(-4); // retourne [true, true, true, true, true, true, false,
    false]
```

N'oubliez pas de veiller à la validité des paramètres de vos méthodes (assertions). Le tableau de booléens est présumé exister (différent de `null`).

3.2 Manipulation de `String` : le fichier `Text.java`

Tout au long du projet, vous aurez à manipuler des chaînes de caractères. Nous représenterons ces dernières sous **trois** formats/types différents.

- Le type `String` : représentation naturelle des chaînes de caractères en Java ;
- Le type `byte[]` : représentation en mémoire des chaînes de caractères ;
- Le type `boolean[]` représentera une chaîne de caractères au format *bit map* ("noir et blanc", les détails sur le format suivront).

Une chaîne de caractères peut en effet naturellement être représentée au moyen du type prédéfini `String`. Travailler avec ce seul type peut néanmoins nous poser souci. Par exemple, certains caractères sont non affichables et cela complique donc la vérification des résultats des algorithmes. C'est la raison principale pour laquelle nous allons utiliser les deux autres formats alternatifs.

3.2.1 Méthodes implémentées pour vous

La représentation en mémoire d'une `String` dépend par défaut de l'encodage de la machine utilisée. Pour simplifier, nous allons adopter un encodage précis, l'encodage *UTF-8* et nous mettons à votre disposition les deux méthodes suivantes :

```
public static byte[] toBytes(String str)
```

```
public static String toString(byte[] bytes)
```

La première prend en entrée une chaîne de caractères et retourne sa représentation sous la forme d'un tableau de `byte` selon l'encodage UTF-8. La seconde fait l'opération inverse en interprétant une séquence de `byte` selon l'encodage UTF-8. Ceci permet d'obtenir des résultats indépendants de la machine et il vous sera suggéré dans l'énoncé quand ces méthodes devraient être utilisées.

3.2.2 À votre tour d'implémenter

Pour cette étape, vous aurez à implémenter deux méthodes. La première vous permettra de passer de la représentation au format `String` vers la représentation au format `boolean[]`. La seconde sera la réciproque de la première.

Pour commencer, nous vous demandons d'implémenter la méthode `Text::toBitArray`, dont la signature est :

```
public static boolean[] toBitArray(String str)
```

Cette méthode aura comme rôle de convertir une chaîne de caractères en tableau de booléens en interprétant son encodage en UTF-8. Il s'agira donc de convertir la chaîne en tableau de `bytes` (selon UTF-8), puis de traduire en séquence, chacun des bits de ces `bytes`, en booléen. Pensez à utiliser ce que vous avez codé dans le fichier `Bit.java`

Par exemple, l'appel à la méthode `Text::toBitArray` avec comme paramètre la chaîne de caractères *"ô$"* ("o circonflexe" et "symbole dollar") retournera le tableau suivant, représentant une chaîne de caractères au format *bit map*:

```
{ true, true, false, false, false, false, true, true,
true, false, true, true, false, true, false, false, false, false,
true, false, false, true, false, false }
```

La deuxième méthode, réciproque de la méthode que vous venez d'implémenter, prendra comme paramètre une chaîne de caractères au format *bit map* et retournera la représentation au format `String`.

```
public static String toString(boolean[] bitArray)
```

Cette méthode est une *surcharge* de la méthode `toString` déjà fournie. Le tableau de booléens est censé exister (différent de `null`, vérifiable par une assertion). Notez que la taille d'un octet peut s'obtenir par la formule `Byte.SIZE`. Si le nombre d'éléments du tableau `bitarray` n'est pas un multiple de cette taille, les données excedentaires à la fin de ce dernier peuvent être ignorées.

3.3 Manipulation d'images : fichier `Image.java`

Vous aurez dans ce projet à manipuler des images modélisées sous la forme de tableaux de *pixels*. Il vous est demandé maintenant de coder divers utilitaires nécessaires aux traitements requis.

3.3.1 À la découverte du format ARGB

Comme expliqué dans le complément théorique (voir la section 8), un pixel n'est rien d'autre qu'une couleur en représentation ARGB.

- Pour cette partie du projet, vous avez **l'obligation d'utiliser les opérateurs binaires** introduit en cours.
- Les compléments théoriques vous montrent que les composantes primaires d'une couleur ARGB sont compris entre 0 (inclus) et 255 (inclus). Ceci devrait vous aider à comprendre pourquoi le type `int` est suffisant pour représenter un pixel sans perte d'information. Il convient cependant de se préoccuper d'une question importante : est-ce que le fait que Java ne permet pas d'utiliser des nombres non signés change quelque chose ? C'est un point auquel il faudra porter attention dans ce qui suit.

Afin de vous familiariser avec la modélisation des couleurs en format ARGB, il vous est demandé d'implémenter:

- quatre méthodes de la classe `Image` permettant d'extraire les composantes primaires d'un pixel;
- ainsi qu'une méthode réciproque permettant de former un pixel à partir de ses composants primaires.

Voici la signature des méthodes à implémenter ainsi qu'une description du comportement attendu de chacune d'elle:

- La méthode `public static byte alpha(int pixel)` aura comme rôle d'extraire la composante *alpha* (transparence) d'un pixel donné.
- La méthode `public static byte red(int pixel)` devra extraire la composante rouge du pixel.
- La méthode `public static byte green(int pixel)` quant à elle, vous permettra d'extraire la composante verte du pixel.
- Finalement, vous devriez être capable d'extraire la composante bleue d'un pixel à l'aide de la méthode `public static byte blue(int pixel)`.

Comme exemple, un extrait de code vous est fourni ci-dessous pour vous aider dans votre travail.

```
int pixel = 0b11111111_11110000_00001111_01010101 ;// -1044651 (0xFFF00F55)
// On extrait les differentes valeurs:
byte alpha = Image.alpha(pixel); // -1 ( qui vaut 255, 0xFF, en non signé)
byte red   = Image.red(pixel) ;  // -16 (qui vaut 240, 0xF0, en non signé)
byte green = Image.green(pixel); // 15 (0x0F)
byte blue  = Image.blue(pixel);  // 85 (0x55)
```

Maintenant que vous êtes capable d'extraire les composantes primaires d'un pixel donné, il est temps pour vous d'essayer de former un pixel à l'aide de ses composants primaires. Implémentez pour cela la méthode `Image::argb` ayant pour signature :

```
public static int argb(byte alpha, byte red, byte green, byte blue)
```

L'exemple suivant devrait vous aider à comprendre ce qui est attendu:

```
int pixel = argb (-1, -16, 15, 85);// -1044651 (0xFF_F0_0F_55)
```

- Lors de l'implémentation de cette méthode, vous prendrez garde au fait qu'une couleur doit nécessairement être non signée (valeur comprise en 0 et 255). Par conséquent, les `byte` fournis en paramètre devront être convertis en valeurs non signées avant la reconstitution de l'entier.
- N'oubliez pas de tester votre implémentation grâce aux tests fournis dans le fichier `Main.java` que vous êtes encouragés à enrichir à votre convenance.

3.3.2 Un deuxième format d'image : Grayscale

Certains des traitements requis nécessiteront de travailler avec des formats d'images simplifiés. Plus précisément, il s'agira d'images en niveaux de gris ou en noir et blanc.

Pour commencer, il vous est demandé d'implémenter la méthode `Image::gray` qui permettra de calculer la nuance en gris (niveau de gris) d'un pixel donné. Cette dernière peut facilement être calculée comme étant la moyenne des composantes primaires (*transparence* exclue).

La méthode à implémenter aura la signature suivante :

```
public static int gray(int pixel)
```

Par construction, l'entier retourné aura toujours une valeur comprise entre 0 et 255 : en effet la valeur maximale correspond à la situation où chaque octet de l'entier construit vaut 255 (auquel cas la moyenne vaut 255) et la valeur minimale zéro (chaque octet vaut zéro, et donc leur moyenne aussi).

Voici un exemple illustratif du fonctionnement de cette méthode:

```
gray(2146438997) // 2146438997 = 0xFFF00F55 = (240 + 15 + 85) / 3 = 113
```

des tests additionnels sont fournis dans le fichier `Main.java`.

3.3.3 Pixel noir ou pixel blanc ? Le format binaire

Le dernier format d'image que vous aurez à implémenter n'est pas un format usuellement utilisé. Il s'agit d'un format «fait maison» qui nous facilitera certaines tâches. Nous nommerons ce format le *format binaire* et allons le définir à partir du format *nuances en gris*.

Pour convertir un pixel au format binaire, il vous est demandé d'implémenter la méthode `Image::binary` qui prendra comme paramètres la *nuance en gris du pixel* (qui doit être une valeur comprise entre 0 et 255) ainsi qu'un seuil (`threshold`). Cette méthode devra retourner la valeur `true`, qui représentera un pixel blanc, si le niveau du gris du pixel est supérieur ou égal au seuil. De manière complémentaire, cette dernière devra retourner la valeur `false`, qui representra un pixel noir, si le niveau de gris est strictement inférieur au seuil donné.

La signature de la méthode `Image::binary` est la suivante :

```
public static boolean binary(int gray, int threshold)
```

En raison de la simplicité de `Image::binary`, nous ne fournissons pas d'exemple d'execution de la méthode. Néanmoins, un certain nombre de tests sont à votre disposition dans `Main.java`.

3.3.4 De la manipulation de pixel à la manipulation d'image

Nous disposons maintenant d'outils permettant de travailler avec des pixels isolés en différents formats. Il est temps de les utiliser pour travailler à l'échelle d'images complètes.

La première méthode que vous aurez à implémenter est la méthode `Image::toGray` qui aura comme rôle de convertir une image du format *ARGB* au format *nuances de gris*. La signature de la méthode prendra la tournure suivante :

```
public static int[][] toGray(int[][] image)
```

Le paramètre `image` sera considéré correct s'il est non nul.

Dans la foulée, implémentez aussi la méthode `Image::toBinary` qui vous permettra de convertir une image du format *nuances de gris* au format *binaire*. La signature de la méthode vous est fournie ci-dessous :

```
public static boolean[][] toBinary(int[][] image, int threshold)
```

> Pour l'instant, les méthodes implémentées permettent la conversion du format *ARGB* au format *nuances de gris* et *binaire*. À votre avis, serait-il possible d'implémenter les méthodes réciproques aux méthodes citées plus haut ? Si non, pourquoi ?

Intéressons nous maintenant aux fonctions réciproques de `Image::toGray` et `Image::toBinary`. Pour commencer, considérons la méthode `Image::fromGray` permettant de convertir une image du format *nuances de gris* au format *ARGB*.

> Les questions qui se posent sont les suivantes: «Est-ce qu'il est possible de reformer un pixel à partir de son niveau de gris ? Si non, quelle serait la meilleure approximation d'un pixel à partir de son niveau de gris ?»

Comme vous pouvez le deviner nous aurons évidemment recours à une approximation en raison de la perte d'information (un niveau de gris ne permet pas de reconstituer la couleur).

Implémentez la méthode `Image::fromGray`, ayant comme signature :

```
public static int[][] fromGray(int[][] image)
```

et dont l'exécution obéit à la spécification suivante:

> Soit un pixel de l'image en niveaux de gris, valant `gray`. Le pixel correspondant dans l'image ARGB à retourner est tel que :
>
> - il est complètement opaque (`0xFF` pour la composante `alpha`);
> - et chacune de ses composantes, rouge, verte et bleue vaut la valeur `gray`.

Intéressons nous maintenant à passer du format binaire à un format en niveau de gris. Rappelons-nous que le format *binaire* nous permet seulement de représenter 2 couleurs (noir et blanc). Le format *nuances de gris*, quant à lui, permet de représenter 256 couleurs au total.

Il est donc clair que la conversion du format *nuances de gris* vers le format *binaire* nous a fait aussi perdre de l'information et il est impossible de recalculer le niveau de gris à partir de la représentation *binaire*. Néanmoins, il est toujours possible d'approximer les calculs.

Dans cette optique, il vous est demandé d'implémenter la méthode `Image::fromBinary` qui permettra de convertir une image du format *binaire* au format *nuances de gris*.

La signature de la méthode se profilera de la manière suivante :

```
public static int[][] fromBinary(boolean[][] image)
```

Cette méthode devra faire la conversion en suivant la règle ci-dessous :

> Si un pixel est blanc dans `image` (`true`), la valeur `gray` de niveau de gris associée vaudra 255.
> Au contraire, si le pixel est noir, la valeur `gray` devra prendre la valeur 0.

Le pixel en niveau de gris à calculer dans ce cas est un entier dont l'octet le plus significatif vaut `0xFF` et dont les autres octets valent la valeur `gray`.

Attention à bien éviter les duplications de code

Nous vous rappelons que des tests vous sont fournis dans le fichier `Main.java`.

3.4 Tests

Pour tester le comportement de vos utilitaires, utilisez le programme `Main.java`. Vous pouvez y invoquer toutes méthodes utiles aux tests. Certains exemples de tests sont fournis au travers de méthodes que vous pouvez utiliser et dont vous pouvez vous inspirer. Ces exemples ne sont pas exhaustifs. complétez-les selon ce qu'il vous semble judicieux pour tester l'ensemble des méthodes implémentées dans cette première partie du projet.

Une petite astuce utile au débogage consiste à utiliser la facilité offerte par IntelliJ de faire afficher les valeurs en format hexadécimal ou binaire : dans le débogueur, faites un clic droit sur la une valeur de type entier puis *View as > Hex ou View as > binary*.

4 Tâche 2 : Algorithmes cryptographiques

Dans cette seconde partie du projet, vous allez découvrir le monde de la cryptographie. Il s'agit d'une introduction basique au domaine présentant quelques algorithmes fondamentaux qui ont pavé le chemin vers d'autres techniques beaucoup plus complexes.

Vous aurez à implémenter les méthodes des fichiers `Encrypt.java` et `Decrypt.java` sous le paquetage `ch.epfl.cs107.crypto`.

Le premier fichier regroupe des méthodes de *chiffrement*. Nous allons nous intéresser à différentes méthodes pour chiffrer un texte au moyen d'une clé. Le but est d'arriver à des résultats dont le sens n'est plus récupérable sans la clé une fois chiffré. Il est important de souligner que de nombreux algorithmes de cryptographie peuvent être brisés malgré cela.

Le second fichier regroupe quant à lui des méthodes de déchiffrement. Nous nous intéresserons uniquement à des techniques simples (ou simplifiées) pour récupérer le texte d'origine.

> Les méthodes de chiffrement/dechiffrement travailleront toutes avec des messages exprimés comme séquences de `byte` (représentation des "chaînes de caractères" sous un format plus universellement "lisible" que les `String`).

4.1 Votre Premier Algorithme : L'algorithme de Caesar

La première technique abordée est le *chiffrement par décalage*. Le décalage est en effet l'une des plus anciennes formes de cryptographie. Pour qu'un texte ne soit plus lisible, il suffit de décaler toutes les lettres d'un certain nombre de positions. Il suffira de faire le décalage contraire pour récupérer l'original. Le nombre de décalages à effectuer constitue la clé du chiffrement. Pour concrétiser l'idée, voici un exemple simple : imaginons un alphabet de 4 lettres A, B, C et D. Nous pouvons faire correspondre des chiffres à chacun des symboles, disons 0, 1, 2 et 3 respectivement. Nous obtenons ainsi l'encodage suivant :

A	B	C	D
0	1	2	3

Cet encodage peut être utilisé pour chiffrer un message exprimé dans cet alphabet. Prenons le message **BCDACAD** et *B* comme clé de chiffrement.

L'algorithme de César consiste à additionner la valeur du code de la clé à chaque lettre du message. Cette addition se fait modulo la taille de l'alphabet (ici 4) pour toujours rester dans ce dernier. Pour rappel, x modulo y donne le reste de la division entière de x par y (par exemple : 3 mod 4 = 3, 7 mod 4 = 3).

Le tableau suivant donne le résultat du chiffrement ainsi opéré sur l'exemple précédent :

Texte	B (1)	C (2)	D (3)	A (0)	C (2)	A (0)	D (3)
Clé	B (1)	B (1)	B (1)	B (1)	B (1)	B (1)	B (1)
Résultat (mod4)	C (2)	D (3)	A (0)	B (1)	D (3)	B (1)	A (0)

Le message **BCDACAD** chiffré au moyen de la clé *B* est donc **CDABDBA**. Pour déchiffrer, il suffit de décaler de 1 vers l'arrière (ou de 3 = 4 - 1 vers l'avant). Ceci revient à utiliser la même démarche avec la clé inverse, ici D.

Attention ! Pour rappel, dans notre cas, les caractères ne sont pas que positifs et ont des valeurs entre -128 et 127 compris. Le résultat doit rester dans cette fourchette, en fonctionnant selon un principe analogue au calcul avec modulo, c'est-à-dire que 127+1 doit vous donner -128. N'oubliez pas de convertir le résultat en byte, étant donné que les opérations sur des bytes retournent un int (promotion entière). Un simple « cast » suffit.
Lisez bien les compléments sur les bytes (section 8.2) pour pouvoir résoudre cette partie aisément.

Dans le cadre de votre projet, vous aurez à implémenter les deux fonctions d'encodage et de décodage suivantes :

```
public static byte[] caesar(byte[] plainText, byte key)
```

```
public static byte[] caesar(byte[] cipher, byte key)
```

Même si la signature de la fonction est similaire, l'implémentation sera différente. La première, définie dans la classe `Encrypt`, devra encoder le paramètre `plainText` en utilisant l'algorithme de Caesar avec la clé `key`. La deuxième fonction devra faire l'opération inverse, décrypter le `cipher` en utilisant la clé `key`.

Avant de commencer le codage de ces méthodes, posez vous la question de quelle est la taille de l'alphabet si l'on travaille avec des `byte`.

4.2 Caesar revisité : L'algorithme de Vigenère

Le chiffrement de Vigenère est une version plus avancée du César. Il est très similaire, mais au lieu d'utiliser une clé d'un seul caractère, il opte pour un(e) mot (phrase) que l'on répète tout au long du message pour le crypter. On applique la même méthode que le chiffrement de César pour décaler le texte. Utilisons le même exemple que dans la partie Section 3.2, c'est-à-dire **BCDACAD** avec cette fois la clé **ABC** en gardant l'encodage strictement positif (de 0 à 3) :

Texte	B (1)	C (2)	D (3)	A (0)	C (2)	A (0)	D (3)
Clé	A (0)	B (1)	C (2)	A (0)	B (1)	C (2)	A (0)
Résultat (mod4)	B (1)	D (3)	B (1)	A (0)	D (3)	C (2)	D (3)

Cette fois, chaque caractère peut subir un décalage différent (le premier B est décalé de 0, le premier C de 1, le premier D de 2 etc.).

De manière similaire à l'implémentation de l'algorithme de Caesar, vous aurez à implémenter la fonction

```
public static byte[] vigenere(byte[] plainText, byte[] keyword)
```

de la classe `Encrypt` en plus de la fonction

```
public static byte[] vigenere(byte[] cipher, byte[] keyword)
```

de la classe `Decrypt`. Le nom et l'ordre des paramètres est semblable à celui des méthodes implémentées pour l'algorithme de Caesar, vous êtes donc capable d'inférer le rôle de chacun des paramètres des fonctions pour l'algorithme de Vigenère.

4.3 Chiffrement avec XOR

Le XOR (« OU exclusif ») est une fonction logique très connue en informatique lorsque l'on travaille en représentation binaire. Elle se comporte ainsi :

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

On peut l'utiliser pour chiffrer du texte. Comme nous travaillons avec une codification en byte des symboles, la mise en œuvre de cet algorithme de chiffrement sera grandement simplifiée. En effet, un byte n'est rien d'autre que 8 bits en binaire et le « ou exclusif » entre deux bytes peuvent être calculé par simple appel de l'opérateur binaire (https://www.baeldung.com/java-xor-operator). Dans la méthode

```
xor(byte[] plainText, byte key)
```

chaque symbole du message codé est obtenu en calculant le «OU exclusif» entre le caractère à coder et la clé.

Voici un exemple d'exécution du XOR, toujours avec le texte “i want” :

```
byte[] plainText = {105, 32, 119, 97, 110, 116};
byte key = 50;
byte[] cipherText = xor(plainText, key);
//Valeur attendue pour le cipherText: {91, 18, 69, 83, 92, 70}
```

> Indication: pour déchiffrer, il suffit de chiffrer le message chiffré avec la même clé car appliquer deux fois XOR à une valeur permet de retrouver cette même valeur.

4.4 Chiffrement de Vernam : One-Time-Pad

Le « One-Time Pad » est une technique simple et très efficace. Elle consiste à utiliser un masque jetable constitué d'une suite de bytes de la même taille que le texte à chiffrer. L'i-ème symbole du message codé s'obtient simplement par un « OU exclusif » entre l'i-ème caractère du message à coder et l'i-ème caractère du masque. Pour déchiffrer, on applique le même algorithme au message codé en utilisant le même masque. Le masque ne sera ensuite plus réutilisé pour éviter que l'on puisse le deviner. Le défaut majeur de cette technique est qu'elle est très gourmande en ressources. À chaque échange, nous devons générer un nouveau masque aléatoire, et de plus de la taille du texte, ce qui le rend peu pratique.

Complétez la méthode :

```
oneTimePad(byte[] plainText, byte[] pad)
```

Considérez que le `pad` vous est directement donné par le deuxième argument. Vous vous assurez au moyen d'assertions que le `pad` est au moins aussi long que le message.

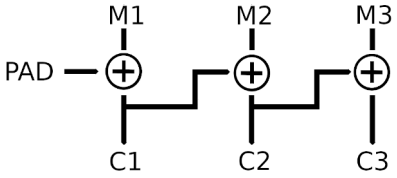

Figure 1: *Processus de chiffrement CBC, partant de la gauche vers la droite : le bloc chiffré C1 de taille T est obtenu par XOR entre la clé PAD de taille T et le bloc M1 de taille T du message d'origine. Le bloc C1 est ensuite utilisé comme clé de chiffrement pour le bloc suivant M2, toujours de taille T, du message d'origine. Le bloc chiffré Ci est obtenu selon le même principe en utilisant Ci−1 comme clé de chiffrement et qui fait l'objet d'un XOR avec le bloc Mi.*

Codez également la variante

```
oneTimePad(byte[] plainText, byte[] pad, byte[] result)
```

cryptant le message `plainText` selon le chiffrement de Vernam au moyen d'un `pad` généré aléatoirement. Lorsque la méthode aura fini son exécution, les deux paramères `pad` et `result` contiendront respectivement le `pad` aléatoire utilisé et le résulat de l'encodage. Le `pad` doit en effet nous être retourné pour le décodage.

Pourquoi le résultat de l'encodage ne peut être retourné par la fonction? Quelle critique peut-on formuler sur cette signature de fonction?

4.5 Un algorithme avancé: Cipher Block Chaining (CBC)

Le *Cipher Block Chaining* est une variante du *One-Time Pad* qui est en général utilisé pour chiffrer un flux de données continu, comme pour les chaînes TV ou streams. Il est combiné habituellement avec une autre méthode de chiffrement, au choix, pour plus de sécurité. Dans notre cas, nous allons nous intéresser au cas simplifié sans combinaison avec une autre méthode de chiffrement.

Le concept est simple. Il s'agit de chiffrer le texte bloc par bloc. Vous commencerez par utiliser le «pad» donné en argument comme première clé. La taille (appelons la T) du «pad» sera la taille d'un bloc. Vous considérez alors les T premières lettres de votre texte, et faites un XOR avec le «pad», comme dans le «One-Time Pad». Cela vous donne la première partie du résultat. Vous allez ensuite utiliser les T lettres du texte chiffré obtenu comme clé pour chiffrer les T lettres suivantes du message, et ainsi de suite. Vous trouverez dans la Figure 1, le schéma qui illustre ce fonctionnement.

Complétez la méthode

```
cbc(byte[] plaintext, byte[] iv)
```

pour qu'elle chiffre le texte de la manière indiquée. Prenez en compte le fait que la dernière partie du texte ne sera pas forcément de la taille du bloc, vous vous arrêterez donc de faire le XOR là où le texte s'arrête.

Quelle critique pourriez vous émettre sur l'algorithme simplifié proposé pour coder CBC?

Pour plus d'informations sur le CBC complet, veuillez vous référer à la partie « Extensions » du projet (section 7.2).

4.6 Tests

Pour tester le comportement de vos nouvelles méthodes, utilisez à nouveau le programme `Main.java`. Encore une fois, les exemples ne sont pas exhaustifs. Complétez-les selon ce qu'il vous semble judicieux pour tester l'ensemble des méthodes implémentées dans cette partie du projet.

5 Tâche 3 : Stéganographie

Cette dernière partie du projet s'intéresse à cacher une image ou un texte dans une autre image. Vous utiliserez pour cela une des méthodes les plus simples: le *Least Significant Bit Encoding* (Encodage sur le Bit de Poids Faible), ou LSB, qui consiste à séparer l'information que l'on veut cacher en *bits* individuels, et de les stocker dans le bit le moins important (et donc le moins visible) de chaque pixel du support.

Pour ce projet, il vous est demandé d'implémenter deux types d'encodage :

- Encodage direct d'image direct : cacher (encoder) et révéler une image en noir et blanc. Il s'agit de:
  - convertir une image fournie vers une image en noir et blanc ;
  - encoder une image noir et blanc fournie, le message ou la *charge*, dans une autre image, le *support*;
  - extraire les bits de poids faible d'un support pour révéler le message caché.
- Encodage de texte : encoder et révéler du texte. Il s'agit de :
  - préparer un message texte (c'est-à-dire, convertir une `String` vers un tableau de bits qui représente l'encodage UTF-8 de chaque caractère de la `String`) ;
  - encoder le tableau de bits dans une image *support* ;
  - extraire les bits de poids faible et les convertir vers une `String`.

Attention aucune des méthode d'incrustation ne doit modifier le support d'origine. Elle doit retourner une version modifiée de ce dernier où le message caché aura été incrusté.

5.1 Encodage direct d'image

Dans cette première partie, vous cacherez une image binaire, une charge composée uniquement de pixels noirs et blancs, dans une autre image, le support (qui peut être en couleur), en changeant le dernier bit de chaque pixel du support vers la valeur du pixel correspondant de la charge. Pour cela, vous convertirez d'abord une image couleur en une image binaire (charge), en passant par une version en nuances de gris; puis vous itèrerez sur chaque pixel d'une autre image (support) pour y modifier le dernier bit, de façon à ce que la valeur du pixel de la charge soit stockée dans ce bit.

Vous implémenterez aussi le dévoilement d'une image cachée de manière analogue : vous itèrerez sur chaque pixel du support et lirez la valeur du dernier bit pour reformer la charge en tant qu'image binaire. Ces points sont décrits plus en détail ci-dessous.

Pour coder cette partie, pensez à utiliser les méthodes utilitaires appropriées du fichier `Bit.java`. Il sera considéré que les images fournies en paramètres sont correctement dimensionnées (différentes de `null`, ne contenant pas de lignes valant `null` , ne sont pas vides et ne contiennent pas de lignes vides).

5.1.1 Incrustation

Dans le fichier `ImageSteganography.java`, implémentez la méthode

```
embedBW(int[][] cover, boolean[][] load)
```

qui dissimule une image binaire `message` dans une image RGB `cover` en changeant le dernier bit de chaque pixel de `cover` vers la valeur du pixel correspondant de `load`.

Vous itérerez donc sur chaque pixel de `load`, en commençant en haut à gauche[4], et remplacerez le dernier bit du pixel de `cover` situé à la même position par la valeur de celui de `load`.

Important: souvenez vous de la consigne; l'incrustation ne doit pas se faire sur le **support d'origine qui doit rester inchangé**.

Vous tiendrez compte aussi du fait que la charge doit être plus petite ou égale au support.

Implémentez ensuite les méthodes:

```
embedGray(int[][] cover, int[][] grayImage, int treshold)
```

qui dissimule dans une image `cover`, une image en niveau de gris `grayImage`. Cette dernière sera au préalable convertie en image binaire en fonction du seuil `treshold`.

```
embedARGB(int[][] cover, int[][] argbImage, int treshold))
```

qui dissimule dans une image `cover`, une image en couleur. Cette dernière sera au préalable convertie en image en niveau de gris, puis en image binaire en fonction du seuil `treshold`.

5.1.2 Dévoilement

Pour pouvoir décoder l'image cachée, vous implémenterez ensuite la méthode

```
revealBW(int[][] image)
```

qui forme une image binaire en lisant les valeurs du bit de poids faible (le plus à droite) de chaque pixel de l'image ARGB `cover`.

Exemple d'utilisation

```
int[][] cover = Helper.readImage("cover.jpg");
int[][] messageImage = Helper.readImage("message.jpg");
int[][] gray = Image.toGray(messageImage);
boolean[][] bw = Image.toBinary(gray, 240);
Helper.show(bw, "Black and white image to hide");
int[][] hidden = ImageSteganography.embedBW(cover, bw);
Helper.show(hidden, "Cover with hidden image");
//retourne le support contenant le message caché
boolean[][] decoded = ImageSteganography.revealBW(hidden);
Helper.show(decoded, "Black and white revealed image");
```

[4] le point (0,0) du référentiel pour les images est en haut à gauche

5.1.3 Tests

Référez vous au programme `Main.java` pour commencer à tester votre implémentation. N'hésitez pas à compléter les tests fournis.

5.2 Encodage texte

Il s'agit maintenant de traiter le cas où la charge n'est plus une image binaire, mais un texte sous forme de `String`. Il faudra d'abord convertir ce texte en tableau de booléens, via une représentation en tableau d'entiers, pour pouvoir le cacher dans le support (qui est toujours une image). Avec la méthode du bit de poids le plus faible utilisée dans ce projet, il n'est en effet possible d'incruster qu'une suite de bits. Pour cacher du texte (une `String`), il faut donc commencer par le convertir en un tableau de bits.

Pour cela, on convertira chaque caractère vers sa valeur numérique (`byte`), puis chaque entier ainsi obtenu vers sa représentation binaire (`boolean[]`). On obtiendra ainsi un tableau de booléens qui correspond à la représentation binaire de la `String` que l'on veut encoder.

De plus, lorsqu'on dévoile le message encodé, on obtient un tableau de booléens. Il faut donc convertir ce tableau vers la `String` à laquelle il correspond. Pour cela, on appliquera l'opération inverse de celle utilisée pour transformer la `String` en tableau binaire: on convertira le tableau de booléens en tableau d'entiers, puis on convertira ce tableau d'entiers en caractères, et donc en `String`.

> Pensez à nouveau à utiliser tous les utilitaires appropriés du fichier `Bit.java`

5.2.1 Incrustation et dévoilement

La figure 2 illustre comment incruster la String: chaque nouvelle couleur correspond à un nouveau caractère. Cette figure utilise 4 bits pour un caractère, mais vous en aurez 16.

Cas d'un tableau de bits

Implémentez

```
embedBitArray(int[][] cover, boolean[] message)
```

qui incruste le tableau `message` dans les bits de poids faible des `X` premiers[5] pixels de l'image `cover`, où `X` est la longueur du tableau `message`. L'image `cover` doit rester inchangée. Si le message est trop long pour être entièrement caché dans l'image, il sera simplement tronqué (le reste du message qu'il n'est pas possible de cacher est ignoré).

Puis implémentez

```
revealBitArray(int[][] cover)
```

qui retourne le tableau booléen unidimensionnel correspondant aux bits de poids faible de `cover`.

Cas d'une `String`

Implémentez maintenant

```
embedText(int[][] cover, byte[] message)
```

et

[5] En commençant au pixel {0,0}, et en traversant de gauche à droite puis de haut en bas.

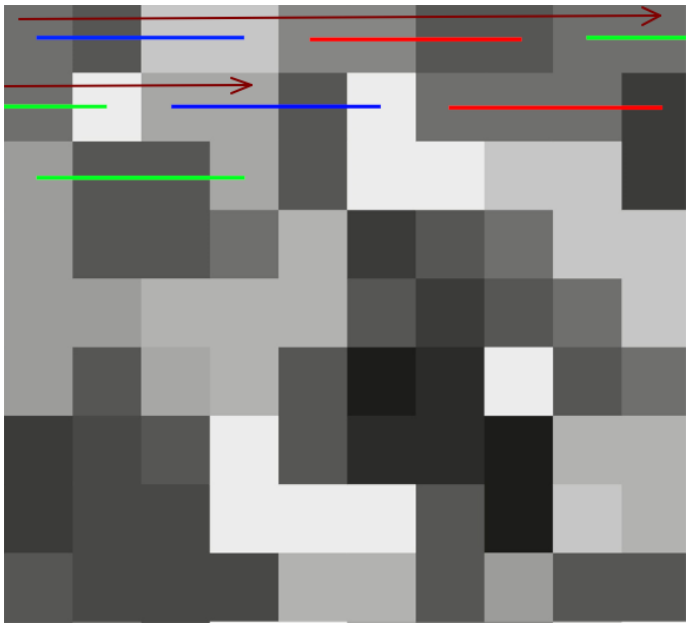

Figure 2: Les 4 bits du premier caractère sont incrustés dans les bits de points le plus faible des 4 premiers pixels, ceux du second caractère sont incrustés dans les bits de points le plus faible des 4 pixels suivants et ainsi de suite (dans cet exemple on veut cacher 6 caractères encodés sur 4 bits chacun au lieu de 16). Les flèches indiquent le sens du parcours.

```
revealText(int[][] cover)
```

qui, en utilisant les méthodes que vous avez codées jusqu'ici, dissimule ou révèle un `byte[]` dans/depuis une image `cover`.

5.3 Tests

Pour tester cette partie du projet, utilisez à nouveau le programme `Main.java`.

5.4 Combiner cryptographie et stéganographie

Vous avez à ce stade codé tous les outils nécessaires à permettre l'usage combiné des outils de cryptographie et stéganographie.

Vous pouvez si vous le souhaitez coder des méthodes permettant explicitement de procéder à de telles combinaisons.

Par exemple:

```
  int[][] embedText(int[][] cover, byte[] message, String
     encryptionMethodName, byte[] key)
```

```
byte[] revealText(int[][] image, String encryptionMethodName, byte[] key)
```

6 Challenge Capture the Flag : Trouver le Drapeau

6.1 Introduction

Bienvenue dans ce challenge de capture the flag ! Vous allez devoir résoudre une série d'énigmes pour découvrir un message caché dans des images. Vous aurez à utiliser les indices fournis pour trouver le drapeau final.

6.2 Instructions

1. **Les fichiers suivants sont fournis:**

   - `hint1.txt`
   - `hint2.txt`
   - `hint3.txt`
   - `hint4.txt`
   - `image1.jpg`
   - `image2.jpg`

2. **Utilisez les indices fournis dans les fichiers `hint*.txt` dans l'ordre pour trouver les messages cachés dans les images `image1.jpg` et `image2.jpg`.**

3. **Résolvez les indices pour découvrir les messages cachés dans les images. Les images doivent être traitées dans l'ordre suivant :**

   - `image1.jpg`
   - `image2.jpg`

4. **Implémentez votre solution dans la fonction `challenge()` de la classe `Challenge`**

5. **Retournez le flag final sous la forme `FLAG{*********}`.**

6. **Jetez un œil à la classe `Helper.java`. Elle pourrait s'avérer très utile ;-).**

6.2.1 Jouez le jeu!

Ne partagez pas le flag entre vous. Cela contribue à ce que le challenge soit amusant pour tout le monde :-)
Bonne chance !

7 Extensions

Si vous avez envie, à titre personnel, de poursuivre le projet plus loin, cette section vous propose des suggestions d'extensions.

Si vous souhaitez rendre du code avec des extensions, prenez soin de les documenter dans un fichier README et à préserver la qualité du code (il serait dommage que l'ajout d'extensions péjore la qualité de votre code et donc impacte négativement la note sur le style).

La mise en oeuvre d'extensions implique que vous fassiez preuve de plus d'indépendance et les assistants vous aideront en principe moins.

7.1 Éléments de cryptanalyse

Ce projet vous donné une introduction basique à un certain nombre des cryptosystèmes classiques. La cryptanalyse est, quant à elle, la discipline qui se focalise sur les moyens de briser ces cryptosystèmes, notamment dans l'objectif d'en vérifier la robustesse.

Vous pouvez donc étendre le projet en y ajoutant des méthodes qui tentent de décrypter les messages chiffrés sans en connaître la clé. Les méthodes naive «brute force» qui essaient toutes les clés possibles, sont applicables aisément à l'encodage de César ou l'encodage XOR.

Notez que briser un message encodé à l'aide d'un CBC est possible à l'aide du «Brute Force», mais extrêmement coûteux. Il faudrait tester toutes les tailles de bloc possible avec 256 possibilités par caractère du «$_{\text{pad}}$» originel. Imaginez une taille de bloc de 5, si l'on teste les possibilités de taille 1 à 5, cela ferait $256 + 256^2 + 256^3 + 256^4 + 256^5$; ce qui fait plus d'un milliard de possibilités.

Il est bien sûr possible de faire mieux que les approches «brute force» en utilisant des techniques plus élaborées telles que l'analyse fréquentielles des caractères. Si cela vous intéresse des compléments à ce sujet peuvent être fournis à la demande.

7.2 Extension de CBC

Vous pouvez pousser plus loin le concept du CBC en ajoutant, après le XOR, une méthode de chiffrement de votre choix pour solidifier l'implémentation, et ensuite déchiffrer le tout. Pour plus d'informations sur le CBC, vous pouvez regarder les figures sur wikipedia à l'adresse : `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#CBC`.

7.3 Interpréteur de commande

Une autre idée de bonus consiste à implémenter un shell pour chiffrer et déchiffrer des messages. Un shell est un programme basique en ligne de commande, que vous programmez de sorte qu'il comprenne des commandes rentrées dans la console. Voici l'exemple d'un shell basique :

```
...
boolean isFinished = false;
Scanner scan = new Scanner(System.in);

while(isFinished){
    System.out.print("Voulez vous terminer votre programme? [Oui/Non]");
    String s = scan.nextLine();

    if(s.equals("Oui")){
        isFinished = true;
    }else{
        System.out.println("Le programme ne se termine pas.");
    }
}
...
```

Ce code correspond à un programme qui tourne en boucle tant que vous ne lui dites pas oui en réponse à la question. Vous pouvez donc faire un shell qui possède des fonctions et des commandes, de manière à ce que vous puissiez lui donner du texte à encrypter et le lui faire décoder. Bien entendu vous devez le faire le plus "user friendly" possible, c'est-à-dire facile d'accès pour n'importe quel utilisateur. Une commande "help" ou "aide" listant l'intégralité des commandes et fonctions de votre programme est requise.

8 Complément théorique

8.1 Représentation ARGB

La représentation ARGB d'un pixel à l'aide d'un entier se fait de la manière suivante : La couleur est décomposée en 4 composants ayant une valeur comprise entre 0 et 255:

- **Alpha** est l'opacité du pixel. Si elle est à 0 alors le pixel est « invisible », peu importe ses autres composants. Si elle est entre 1 et 254 alors il est transparent et laisse passer les couleurs de l'image derrière celui-ci. Si sa valeur est 255, alors le pixel est parfaitement opaque.
- **Red** est la valeur d'intensité de la lumière rouge du pixel.
- **Green** est la valeur d'intensité de la lumière verte du pixel.
- **Blue** est la valeur d'intensité de la lumière bleue du pixel.

En considérant que le bit de poids le plus faible d'un entier ait pour indice 0 et que le bit de poids fort ait pour indice 31, alors ces 4 composants sont placés sur un même entier de telle sorte~: les bits 31 à 24 définissent la valeur alpha, les bits 23 à 16 la valeur rouge, les bits 15 à 8 la valeur verte et les bits 7 à 0 la valeur bleue. Par exemple, si l'on veut représenter un pixel rouge en RGBA on a besoin que le composant alpha et rouge soit au maximum 255. Ce qui nous donne:

Alpha	Rouge	Vert	Bleu
255	255	0	0
0xFF	0xFF	0x00	0x0
11111111	11111111	00000000	00000000
31 24	23 16	15 8	7 0

La manière la plus pratique pour représenter une couleur en Java est d'utiliser l'écriture hexadécimale (plus concise). Par exemple pour définir le pixel rouge nous pouvons écrire:

```
int red = 0xFF_FF_00_00;
```

Les opérateurs «bitwise» permettent d'extraire facilement des valeurs de canaux à partir de la représentation sous la forme d'un entier (voir exemple sur la page suivante).

```java
// Notre couleur en binaire
int x = 0b00000000_00100000_11000000_11111111;
// -> 2146559 (0x20c0ff)

// Décale de 8 bits vers la droite
int y = x >> 8; // -> 0b00000000_00100000_11000000

// ET binaire, ce qui a pour effet de ne garder que les 8
// premiers bits
int z = y & 0b11111111; // -> 0b11000000
// On a bien récupéré notre composante verte à 192
// on aurait aussi pu écrire: int z = y & 0xff;
```

8.2 Entiers signés et non signés

En mémoire, un entier n'est qu'une simple séquence de bits. Sa valeur effective dépend entièrement de la façon dont les bits qui le composent sont interprétés. Il existe plusieurs façons d'interpréter une séquence de bits, chacune d'elles ayant ces propres bénéfices et les plus connus étant: «2's complement» (Le complément à deux), «1's complement», «sign and magnitude» et «offset binary».

En java, les entiers sont tous encodés sous forme de *complément à deux*. C'est à dire que les entiers en java peuvent être négatifs et qu'en ajoutant 1 à la plus grande valeur représentable on obtient ls plus petite et qu'en retranchant 1 à la plus petite valeur possible ont obtient la plus grande valeur possible. Contrairement à d'autres langages de programmation, comme le C, il n'est pas possible de désactiver cette interprétation ce qui peut impliquer certaines difficultés.

Voici un résumé des points importants à retenir lors du codage du projet pour ce qui est de la représentation des entiers:

- Les canaux du format ARGB prennent comme valeurs des entiers non signés entre 0 et 255 (ce qui nécessite 1 octet).
- Le type entier `byte` est codé sur un octet, mais ses valeurs sont toujours considérées comme signées en Java et elles sont représentées en complément à deux.
- Dans un `byte` le bit de poids fort (celui tout à gauche) indique le signe. Lorsque vous convertissez un nombre plus grand que la limite maximum, des parties seront donc tronquées. Un `int` correspond à 32 bits. Imaginez le cas où vous avez un `int` qui vaut 255 (11111111). Vous additionnez 1 et obtenez 256 (100000000). Si vous le convertissez en `byte` vous perdrez tous les bits au delà du 8ème en partant de la droite, donc vous obtiendrez comme valeur 00000000 qui fait 0 en `byte`.
- En complément à deux l'addition et la soustraction sur les nombres binaires se font en arithmétique modulaire («wrap around»), c'est à dire qu'elles se font modulo le nombre maximal représentable et qu'il n'y a pas de débordement.

  Une astuce pour permettre d'obtenir les valeurs en arithmétique modulaire des additions et soustraction entre nombres binaires compris entre 0 et 255 est de les convertir en `byte`:

  ```java
  byte b = (byte)255;
  System.out.println(b); // - (2^7 - (255-128)) = -1
  ```

```
b = (byte)248;
System.out.println(b); // - (2^7 - (248-128)) = -8

// Arithmétique modulaire ("wrap around")
b = (byte)(255 + 1);
System.out.println(b);  // 0
b = (byte)-255;
System.out.println(b);  // +1
byte b1 = 127;
byte b2 = -128;
b = (byte)(b2-b1);
System.out.println(b);  // +1
```

- La méthode Java `int Byte.toUnsignedInt(byte x)` donne la valeur en représentation non signée du `byte x`:

```
byte b = -1;
System.out.println(Byte.toUnsignedInt(b)); // 255

b = 127;
System.out.println(Byte.toUnsignedInt(b)); // 127
```

Pour le représentation en complément à 2, vous pouvez regarder la vidéo «Représentation binaire des nombres entiers» par Olivier Lévêque (https://tube.switch.ch/videos/JWKOU3kEA1) ou la vidéo «Représentation des entiers en binaire» de Ronan Boulic (https://youtu.be/a5gLSc0tbjI).
En java pour convertir un entier en sa valeur non signée vous pouvez avoir recours à différents procédés simples (voir par exemple ceci).

8.3 Générateurs aléatoires de nombres

Il est nécessaire à plusieurs reprises dans ce projet d'avoir recours à un générateur de nombre aléatoires. C'est le cas typiquement pour des méthodes telles que le «One-time pad» et le «CBC». Un tel générateur peut être créé simplement en Java de la manière suivante:

```
import java.util.Random;

Random r = new Random();
```

Pour utiliser un générateur ainsi déclaré, il suffit ensuite d'invoquer la méthode `r.nextInt(int i)` qui permet d'obtenir un nombre aléatoire compris entre 0 et i, i non inclus.

Les nombres générés sont en réalité «pseudo-aléatoire» : les algorithmes utilisés par les générateurs ont recours à ce que l'on appelle une «graine» de départ («seed»). À partir d'une même graine, on obtient en fait toujours la même séquence de nombres (ce qui fait que ce n'est pas complètement aléatoire en réalité).

En cours de développement, les méthodes ayant recours à des nombres aléatoires sont difficiles tester du fait que leur comportement varie d'une exécution à l'autre. Pour contourner ce problème, il est possible de faire

en sorte que la séquence «aléatoire» soit toujours la même en fixant la graine («seed»). Ceci peut se faire en Java en déclarant le générateur comme suit:

```
import java.util.Random;

Random r = new Random(seed);
```

`seed` est une valeur de type `long`. Ce type modélise des entiers de plus grande taille que des `int` (permet de modéliser des valeurs entières bien plus grandes).

La valeur donnée à `seed` n'a pas d'importance dans notre contexte. L'appel pour une graine valant 2 se rédige ainsi:

```
Random r = new Random(2l); // le 'L' minuscule après le 2 indique un
   littéral de type long
```

Dans notre cas pas besoin de chiffres trop grands, vous pouvez utiliser un entier de votre choix (0, 1, 2 etc.).

References

[1] Trappe, Wade and Washington, Lawrence C. *Introduction to Cryptography with Coding Theory (2nd Edition)*. Prentice-Hall, Inc., 2005

[2]https://www.eiron.net/thesis

[3] https://www.ojp.gov/ncjrs/virtual-library/abstracts/overview-steganography-computer-forensics-examiner