CS-107 Mini-projet 1 : Cryptographie

Gaultier Lonfat Barbara Jobstmann Jamila Sam

23 octobre 2020

Version 1.1

Table des matières

1 Présentation

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

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 ». Un des sous-domaines de la cryptographie est la « cryptanalyse », qui se focalise sur les moyens de briser ces cryptosystèmes, notamment dans l'objectif d'en vérifier la robustesse.

Le but de ce mini-projet est de vous donner une introduction basique à un certain nombre des cryptosystèmes classiques ainsi qu'à quelques techniques de cryptanalyse. Il s'agira notamment de vous présenter quelques uns des plus anciens cryptosystèmes que les humains aient utilisés (appelés César et Vigenère), puis de plus récents. Nous discuterons ensuite de la difficulté de briser ces systèmes à l'aide d'attaques naïves, dites par « force brute », c'est-à-dire, en essayant toutes les clés possibles. Nous vous présenterons enfin quelques techniques pour briser ces systèmes de manière plus efficace. Vous verrez également d'autres méthodes de chiffrement, telles que le « block cipher », difficiles à briser.

Notez que dans ce projet, l'alphabet utilisé pour les messages sera limité à une sous partie du jeu de caractères ISO-8859-1 (`https://fr.wikipedia.org/wiki/ISO/CEI_8859-1`). Ceci permettra notamment de vérifier de façon plus aisée les affichages produits par vos programmes. Nous nous restreindrons également à l'usage de l'anglais, à des fins de simplification[1].

2 Structure et code fourni

2.1 Structure

Le projet est divisé en trois étapes :

1. **Chiffrement** – Dans la première partie, nous allons découvrir différentes méthodes pour chiffrer un texte. Ces techniques se basent sur l'utilisation d'une *clé* de chiffrement.
2. **Cryptanalyse basique** – Dans la deuxième partie, nous allons développer des techniques simples pour récupérer le texte d'origine sans avoir connaissance de la clé. Nous allons aussi implémenter des techniques pour décoder un message chiffré avec la clé.
3. **Cryptanalyse basée sur les fréquences** – Dans la troisième partie, nous allons utiliser l'analyse des fréquences de caractères pour trouver la clé utilisée pour le chiffrement.

Vous disposez de deux fichiers à compléter :

- `Encrypt.java` pour la partie 1
- `Decrypt.java` pour la partie 2 et 3

1. Libre à vous de dépasser cette limitation dans les bonus ouverts du projet

- 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**[a]. Vérifiez que ce programme compile bien, avant de soumettre.
- Vous trouverez dans le dossier fourni `res/`, **quelques fichiers texte que vous pourrez utiliser** comme jeu de données **pour tester vos méthodes**.

a. Cela permet de vérifier que vos signatures sont correctes et que votre projet ne sera pas rejeté à la soumission.

2.2 Code fourni

Dans ce projet, nous vous proposons quelques méthodes qui vous permettront de tester vos résultats dans une méthode `main`. Étant donné que la manipulation de fichiers est trop avancée pour ce cours, vous retrouverez dans le fichier `Helper.java` ce dont vous avez besoin pour gérer cela, en plus de quelques autres méthodes pour vous faciliter les tests :

- La fonction `cleanString(String s)` prend une `String` en entrée et produit une autre `String` simplifiée de sorte à ce qu'il n'y reste que des lettres minuscules et des espaces : les majuscules deviennent des minuscules et les symboles, autres que les lettres, des espaces.
- Les méthodes `stringToBytes(String message)` et `bytesToString(byte[] numbers)` servent à convertir les données. La première transforme une `String` en un `byte[]` que vous pourrez traiter[2]. La deuxième fait l'inverse, ce qui peut être utile à des fins d'affichage dans la console par exemple.
- La fonction `writeStringToFile(String text, String name, boolean append)` vous permet d'écrire une `String` dans un fichier de nom `name` (par exemple `"text.txt"`). Ce fichier sera automatiquement placé dans le dossier `res/` à la racine du projet. Le `boolean append`, s'il vaut `false`, signifie que vous réécrivez le contenu du fichier depuis le début, tandis que s'il vaut `true` vous continuez d'ajouter du texte à la suite de ce qu'il contient déjà. La variante d'en-tête `writeStringToFile(String text, String name)` correspond à une surcharge de la méthode précédente où `boolean append` est par défaut `false`.
- La dernière méthode est `readStringFromFile(String name)`. Elle vous permet de lire un fichier texte de nom `name` pouvant contenir plusieurs lignes et retournera l'intégralité du texte dans une seule `String`.

2.3 Tests

Important : il vous incombe de **vérifier à chaque étape que vous produisez bel et bien des données correctes** avant de passer à l'étape suivante qui va utiliser ces données.

Pour ce qui est de la gestion des cas d'erreurs, il est d'usage de tester les paramètres d'entrée des fonctions ; e.g., vérifier qu'un tableau n'est pas nul, et/ou est de la bonne dimension, etc. Ces tests facilitent généralement le débuggage, et vous aident à raisonner quant au comportement d'une fonction. Nous supposerons que les arguments des fonctions sont valides.

2. Manipuler les données à l'échelle de l'octet (byte) présente des avantages qui vous seront exposés

Pour garantir cette hypothèse, nous vous invitons à 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 `key` n'est pas `null`, vous pouvez écrire `assert key != null;` au début de la méthode. Un exemple d'utilisation d'assertion est donné dans la coquille de la méthode `caesar` dans le fichier `Encrypt.java`.

> Les assertions doivent être activées pour fonctionner. Ceci se fait en lançant le programme avec l'option "-ea" [Lien cliquable ici].

Le fichier fourni `Main.java` vous servira à tester vos développements de façon simple. Un test possible y est fourni en guise d'exemple. À vous de compléter `Main.java` en y invoquant vos méthodes de façon adéquate pour vérifier l'absence de bugs dans votre programme.

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 inchangées : le fichier `SignatureCheck.java` ne doit donc pas comporter de fautes de compilation au moment du rendu.
- En dehors des méthodes imposées, libre à vous de définir toute méthode supplémentaire qui vous sembles 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 autorisés/encouragés à modifier `Main.java` pour faire d'avantage de vérifications.
- Votre code devra respecter les conventions usuelles de nommage.
- Le projet sera codé sans le recours à des librairie externes, notamment sans le recours à `javafx.crypto` (bonus exceptés). Si vous avez des doutes sur l'utilisation de telle ou telle librairie pour les bonus posez-nous la question et surtout faites attention aux alternatives que Eclipse vous propose d'importer sur votre machine.
- Votre projet **ne doit pas être stocké sur un dépôt publique** (de type github). Pour ceux d'entre vous qui sont familiers avec git, nous recommandons l'utilisation de GitLab : https ://gitlab.epfl.ch/.

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

3 Chiffrement

Dans cette première partie, nous nous concentrerons sur le fichier `Encrypt.java`. 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. C'est ce que vous verrez lors des parties suivantes.

La première question qui se pose en vue de l'implémentation des algorithmes qui nous intéressent est celle de la représentation des données. Une chaîne de caractères peut naturellement être représentée au moyen du type prédéfini `String`. Travailler directement avec ce type peut néanmoins nous poser souci. Par exemple, certains caractères sont non affichables et cela complique donc la vérification des résultats des algorithmes. Nous allons donc plutôt manipuler les symboles à chiffrer sous la forme de leur codification entière[4]. Comme le jeu de caractères que nous utilisons contient 256 symboles, le type entier `byte`, qui peut modéliser 256 valeurs différentes comprises entre -128 et 127 est le candidat naturel. Nos chaînes de caractères seront ainsi plutôt manipulées sous la forme de tableaux de `byte`. Vous verrez que cela facilite aussi grandement le codage de certains algorithmes de chiffrement.

Prenez connaissance de l'Annexe A avant de commencer le codage des fonctions requises.

Par ailleurs, vous devrez faire attention à la gestion des espaces. Pour les méthodes de chiffrement de César, Vigenère et Xor, nous faisons le choix délibéré de ne pas les chiffrer. Ceci permet de préserver la structure des phrases et d'observer facilement les décalages opérés par les algorithmes[5] . Pour les algorithmes « *One-time Pad* » et « *CBC* » par contre, les espaces seront systématiquement chiffrés. En effet, les espaces sont d'ordinaire aussi encryptés pour ne donner aucune information aux attaquants potentiels. Il est donc important que vous voyiez cet aspect aussi.

3.1 Gestion des cas limites

Il vous sera demandé dans ce qui suit de coder un certain nombre de méthodes. Vous pouvez considérer que les données d'entrée sont valides.

Ceci étant dit, il est temps de rentrer dans le vif du sujet et d'aborder votre premier algorithme de chiffrement.

3.2 Chiffrement par décalage – Code de César

Le décalage est 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 suffit 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

4. par exemple le symbole « espace » est codifié au moyen de l'entier 32 dans le jeu de caractères ISO-8859-1 et tout entier est toujours affichable en clair !

5. ceci ne vous empêchera pas par le suite de chiffrer aussi les espaces si vous le souhaitez, pour rester plus général

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 a 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 (mod 4)	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 l'Annexe A sur les bytes pour pouvoir résoudre cette partie aisément.

Il vous est demandé ici de compléter la méthode `caesar(byte[] plainText, byte key)` pour qu'elle se comporte comme expliqué précédemment. Considérez le contenu des paramètres `plainText` et `key`, comme déjà représentés par leurs valeurs d'encodage, il vous suffit alors de faire le décalage et de retourner le tableau en résultat. Les espaces (de valeur 32 dans l'encodage) ne seront pas chiffrés et resteront tels quels dans le résultat.

Voici un exemple d'exécution du programme pour deux clés différentes, une positive et une négative, le paramètre `plaintext` étant la représentation en `byte` du texte `"i want"`. Vous noterez que les espaces (pour rappel de valeur 32) ne sont pas touchés :

```
byte[] plainText = {105, 32, 119, 97, 110, 116};
byte key = 50;
byte[] cipherText = caesar(plainText, key);
//Valeur attendue pour le cipherText: {-101, 32, -87, -109, -96, -90}

plainText = {105, 32, 119, 97, 110, 116};
key = -120;
cipherText = caesar(plainText, key);
//Valeur attendue pour le cipherText: {-15, 32, -1, -23, -10, -4}
```

Lorsque vous testerez ce genre de choses (dans `Main.java` par exemple) vous prendrez soin de créer le tableau de `byte` avant de le passer en argument à la méthode `caesar` (la syntaxe `caesar({105, 32, 119, 97, 110, 116}, 50)` est en effet invalide[6]).

3.3 Chiffrement 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) :

6. cela devrait s'écrire `caesar(new byte[]{105, 32, 119, 97, 110, 116}, 50)`

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 (mod 4)	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.). Implémentez la méthode `vigenere(byte[] plainText, byte[] keyword)` en suivant l'algorithme décrit ci-dessus. Comme pour le César, vous ne chiffrerez pas les espaces. Imaginez comme exemple : *"A CD"* et la clé *"AB"* vous encoderez A avec A, C avec B et D avec A, vous ignorez donc l'espace pour la répétition de la clé, ce qui donne "A DD" en résultat. Aussi, comme précédemment, les caractères vont de -127 à 128.

Ci-dessous se trouve un exemple d'exécution du code, toujours avec le `plaintext` "i want" :

```
byte[] plainText = {105, 32, 119, 97, 110, 116};
byte[] key = {50, -10, 100};
byte[] cipherText = vigenere(plainText, key);
//Valeur attendue pour le cipherText: {-101, 32, 109, -59, -96, 106}
```

3.4 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 XOR 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 oeuvre 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 `byte`s peuvent être calculé par simple appel de l'opérateur binaire $\wedge$ (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. Comme précédemment vous laisserez les espaces sans les chiffrer.

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, 32, 69, 83, 92, 70}
```

3.5 Chiffrement de Vernam – masque jetable – One-Time Pad (OTP)

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. Le ième symbole du message codé s'obtient simplement par un « OU exclusif » entre le ième caractère du message à coder et le ième 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. A 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. Ici, vous chiffrerez aussi les espaces pour simplifier la méthode.

Vous vous assurez au moyen d'assertions que le « pad » est au moins aussi long que la message. S'il est plus long vous n'utiliserez que la partie du début s'appariant au message et ignorerez le reste.

3.6 Enchaînement des blocs – 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 par exemple 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. (Pour plus d'informations sur le CBC complet, veuillez vous référer à la partie 6 « Pour aller plus loin » du projet)

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 considérerez 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 le schéma qui illustre la méthode ci-dessous (Figure 1).

Complétez la méthode `cbc(byte[] plaintext, byte[] iv)` pour qu'elle chiffre le texte de la manière indiquée. Comme pour le « One-Time pad », vous chiffrerez aussi les espaces. Prenez en compte 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.

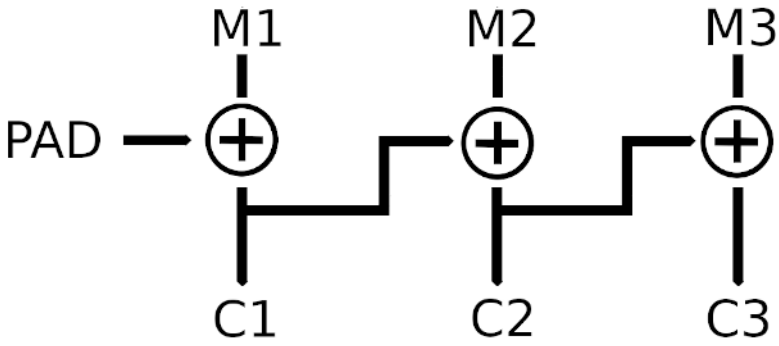

FIG. 1 : Processus de chiffrement CBC, partant de la gauche vers la droite : le bloc chiffré C1 de taille T est obtenu par XOR entre la clé PAD de taille T et le bloc M1 de taille T du message d'origine. Le bloc C1 est ensuite utilisé comme clé de chiffrement pour le bloc suivant M2, toujours de taille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

3.7 Génération de masque/pad

Pour générer un « pad » utilisable pour le « One-Time Pad » et le « CBC », il suffit de générer un tableau contenant des chiffres aléatoires. Dans la méthode `generatePad(int size)` vous initialiserez un tableau de la taille passée en argument. Pour le remplir, vous allez importer le paquet `java.util.Random` et utiliser un générateur aléatoire créé comme ceci :

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

Ce générateur est déjà fourni, aux côtés des variables se trouvant au sommet du fichier `Encrypt.java`. Vous pourrez utiliser `r.nextInt(256)` dans vos méthodes pour obtenir un `int` entre 0 et 255, qui converti en `byte`, sera entre -128 et 127 (voir l'Annexe A pour plus de détails sur la conversion et le test de code faisant appel à des générateurs aléatoires).

3.8 Tout en un

Complétez la méthode `encrypt(String plainText, String key, int type)` pour que vous puissiez chiffrer le texte en entrée `plainText` avec la clé `key` et en utilisant l'algorithme désigné par le paramètre `type`. Les valeurs possibles de ce dernier se trouvent en haut du fichier `Encrypt.java`, vous les utiliserez pour appeler la fonction correspondante. La méthode transformera le texte en tableau de bytes, le chiffrera, et reconvertira le résultat en `String`.

Dans le cas de méthodes de chiffrement n'utilisant qu'un seul symbole, celui choisi sera le premier caractère de `key`. Dans le cas de méthodes utilisant un « pad », considérez le paramètre `key` comme le « pad » (qui aura par ailleurs été généré aléatoirement ou par la méthode appelante).

3.9 Tests

Complétez le programme `Main.java`, selon ce qu'il vous semble judicieux de faire, pour tester l'ensemble des méthodes développées dans cette première partie du projet.

4 Cryptanalyse basique

Nous passons maintenant à la partie déchiffrage. A partir de maintenant, et pour les deux parties restantes du projet, nous ne nous occuperons que du fichier `Decrypt.java`. Nous nous intéresserons ici uniquement à des techniques simples pour récupérer le texte d'origine. Nous allons aussi implémenter la technique pour décoder le CBC avec la clé, car briser CBC sans connaître la clé (« pad » utilisé pour chiffrer le premier bloc) serait trop coûteux en ressources.

4.1 L'algorithme « Brute Force »

Comme son nom l'indique, le « Brute Force » utilise une méthode naïve pour récupérer le texte d'origine. Il va simplement tenter toutes les possibilités de clé, puis l'utilisateur (humain) peut chercher manuellement le texte compréhensible dans le résultat. Cette méthode est loin d'être optimale, comme vous pourrez le voir dans la partie 3 du projet, mais il est important de la connaître. Cette technique sera utilisée dans le projet sur le César et le XOR qui peuvent être brisés de cette façon.

4.1.1 César

Vous remplirez la méthode `byte[][] caesarBruteForce(byte[] cipher)` pour qu'elle itère sur toutes les valeurs de clé possible. Vous allez chaque fois, faire un chiffrement de César avec le texte chiffré et la potentielle clé. Vous stockerez alors le résultat dans la ligne correspondante d'un tableau de type `byte` en deux dimensions (`byte[][]`). (Pour rappel : pour décoder un message encodé avec un César, il suffit de réappliquer le César à nouveau avec la clé inverse)

4.1.2 XOR

La façon d'implémenter la fonction `xorBruteForce(byte[] cipher)` est presque identique à celle du César. La seule différence dans ce cas, est que l'on appelle la méthode XOR à la place.

4.1.3 Préparer le résultat pour l'écriture sur fichier

Complétez la méthode `arrayToString(byte[][] bruteForceResult)` de manière à ce qu'elle convertisse l'ensemble des lignes du tableau en une seule et unique `String`. Vous séparerez chaque ligne du tableau dans la `String` par le caractère `System.lineSeparator()` qui correspond au retour à la ligne de votre OS. Ainsi, il vous suffira d'appeler cette méthode dans votre fichier `Main.java` pour écrire le résultat dans un fichier et tester votre implémentation. Vous devriez voir dans le fichier chaque possibilité, avec les retours à la lignes, dont le message déchiffré.

Voici un exemple d'exécution :

```
byte[][] decoded = {{65,66,67},{68, 69, 70}};
String result = arrayToString(decoded);

//Valeur attendue pour result : "ABC\r\nDEF\r\n"

//(\r\n correspond au retour à la ligne sous windows, si vous êtes sous MACOS
    ou Linux vous trouverez le caractère correspondant à votre OS à la place.)
```

Notez que la méthode `testCaesar` fournie dans `Main.java` vous donne un exemple de comment procéder pour écrire le résultat des tentatives de décryptage dans un fichier (décommentez la partie concernée lorsque vous aurez codé ce qu'il faut).

Pour voir si les méthodes par « force brute » parviennent à leur objectif, il vous « suffit » de **rechercher manuellement un message qui a du sens dans les fichiers produits**.

4.2 Déchiffrer le CBC

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 « 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. Nous allons donc simplement décoder le CBC normalement, comme si nous étions le destinataire en possession du « pad ».

La méthode est très similaire au chiffrement. Au lieu de faire les opération de XOR avec le texte en clair, nous le faisons avec le texte chiffré, ce qui retourne le message d'origine. Plus précisément, on commence par faire un XOR entre le « pad » et le premier bloc du message chiffré, puis on utilise le message chiffré précédent (et non pas le résultat) pour faire un XOR avec le second bloc chiffré et récupérer le second bloc d'origine et ainsi de suite. Cette méthode est illustrée par la figure 2 ci-dessous.

Implémentez la méthode `decryptCBC(byte[] cipher, byte[] iv)` pour pouvoir décoder le CBC comme discuté.

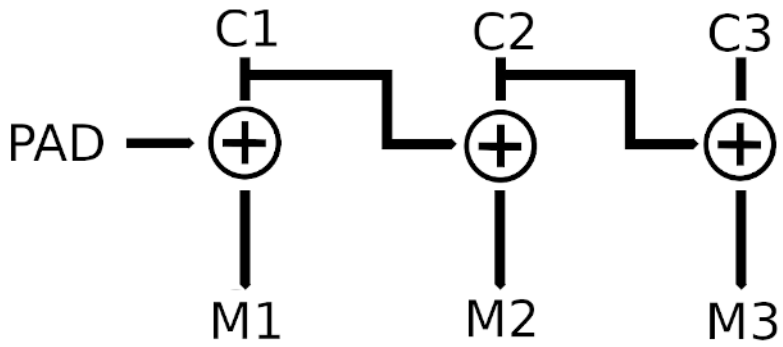

Fig. 2 : Processus de déchiffrage CBC, partant de la gauche vers la droite : Le bloc C1 du message chiffré subit un XOR avec PAD ce qui donne le bloc d'origine M1. Puis, le bloc C1 fait l'objet d'un XOR avec le bloc C2 de qui donne M2. Le bloc M_i est obtenu par XOR entre C_{i-1} et C_i.

Vous noterez une faiblesse plutôt évidente de l'implémentation actuelle du CBC. En effet, comme les textes chiffrés sont XOR entre eux pour décoder le message, si un opposant venait a intercepter le texte chiffré en cours de communication et à deviner la taille d'un bloc, il pourrait dès lors décoder l'intégralité de la suite du message. Il ne pourra par contre pas récupérer le début. C'est pour cela que normalement, le CBC est couplé avec une autre méthode de chiffrage pour palier à ce problème. Vous pouvez trouver plus d'informations à ce sujet dans la partie 6 « Pour aller plus loin » du projet, où vous aurez la possibilité de renforcer la sécurité de cette méthode.

5 Cryptanalyse basée sur les fréquences

Cette dernière partie du projet s'intéresse à détecter plus intelligemment la clé permettant le décryptage du message chiffré, par opposition aux méthodes testant toutes les possibilités. Nous aurons pour cela recours à l'analyse des fréquences de caractères ! En effet il est possible de trouver un rapport entre les fréquences d'apparition d'un symbole dans un texte chiffré et celles de la langue d'origine ; et ainsi en déduire le décalage.

5.1 Déchiffrage intelligent du César

5.1.1 Calcul des fréquences

Vous commencerez par remplir la méthode `float[] computeFrequencies(byte[] cipherText)` pour calculer les fréquences des 256 caractères possibles du texte chiffré. Vous remplirez un tableau de la bonne taille où vous compterez le nombre d'apparitions de chaque caractère dans l'argument `cipherText`. L'indice du tableau correspondant à un caractère est la valeur du caractère lui même.

Faites attention, le tableau commence à l'indice 0, tandis que les bytes peuvent être négatifs. Pensez donc à décaler pour que tout puisse être stocké dans le tableau et dans le même ordre. Vous ignorerez les espaces. Une fois cette étape terminée, vous diviserez chaque case du tableau par le total de caractères différents de l'espace, pour obtenir des valeurs normalisées entre 0 et 1.

5.1.2 Calcul de la clé

Complétez maintenant la méthode `caesarFindKey(float[] charFrequencies)`. Vous allez pour cela comparer le tableau des fréquences de la langue anglaise avec le tableau que vous avez obtenu lors de l'étape précédente. Le tableau des fréquences de la langue anglaise vous est fourni tout en haut du fichier `Decrypt.java`. Il dispose d'une entrée par lettre, par exemple, A a pour valeur 0.08497. A apparaît donc 8.497% du temps en moyenne dans un texte écrit en anglais. On peut comparer avec d'autres lettres, comme E, qui apparaît 11.162% du temps ce qui en fait la lettre la plus utilisée.

Pour analyser les fréquences vous calculerez **le produit scalaire** entre les fréquences anglaises et des parties de votre tableau. Selon l'algorithme expliqués ci-dessous.

Soient vos deux tableaux (ici ne sont montrés que les indices et non le contenu des cases des tableaux et `Cipher` est le tableau de fréquence du message à déchiffrer) :

Anglais :	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Cipher :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...	251	252	253	254	255

L'algorithme consiste à itérer sur votre tableau de fréquences, `Cipher`, en prenant à chaque fois un sous-tableau de la même taille que celui des fréquences anglaises et commençant à l'indice i (i allant de 0 à 255). Il s'agira de calculer leur produit scalaire, puis de décaler i de 1 et de recommencer :

- **Itération 0 :** produit scalaire $= A \cdot 0 + B \cdot 1 + C \cdot 2 + D \cdot 3 + ... + Y \cdot 24 + Z \cdot 25$ (cette notation indique les "indices" des cases utilisées pour calculer le produit scalaire)
- **Itération 1 :** produit scalaire $= A \cdot 1 + B \cdot 2 + C \cdot 3 + D \cdot 4 + ... + Y \cdot 25 + Z \cdot 26$
- ...
- **Itération 255 :** produit scalaire $= A \cdot 255 + B \cdot 0 + C \cdot 1 + D \cdot 2 + ... + Y \cdot 23 + Z \cdot 24$

Attention : les A, B, C, D, ..., 0, 1 ,2, 3, ... utilisés ci-dessus sont les indices des tableaux, vous devez donc les remplacer dans votre programme par les valeurs à ces indices et non pas mettre les indices eux-mêmes.

Gestion cyclique des indices : au bout d'un certain nombre d'itérations, les indices vont commencer à dépasser ceux du tableau `Cipher`. Il s'agit cependant de continuer jusqu'à 255 comme valeur de départ pour l'indice i. Pour parer à ce problèmes, vous gérerez les indices de façon cyclique (« wrap around ») : les indices dépassant 255 repartent depuis 0. Voici un exemple :

Itération 240 : produit scalaire $= A \cdot 240 + B \cdot 241 + C \cdot 242 + ... + P \cdot 255 + Q \cdot 0 + R \cdot 1 + ... + Y \cdot 8 + Z \cdot 9$

Votre but est de trouver pour quelle itération le produit scalaire est maximisé. Plus le produit est grand, plus la ressemblance entre les deux tableaux est grande. L'indice d'itération correspondant au plus grand produit scalaire sera alors l'indice de la fréquence correspondante à celle de la lettre 'a', telle que chiffrée dans votre tableau `Cipher`. En clair, si c'est l'itération i qui donne le plus grand produit scalaire, alors la lettre à la position i dans `Cipher` est devinée comme étant la lettre 'a' chiffrée. La **clé de déchiffrage** est alors simplement donnée par la **la distance** entre `Cipher[i]` et l'encodage de 'a' dans le jeu de caractères que nous utilisons, à savoir 97.

5.1.3 Combiner

Complétez la méthode `caesarWithFrequencies(byte[] string)` en appelant simplement les méthodes codées précédemment dans le bon ordre. Vous pourrez alors essayer, dans `Main.java`, de déchiffrer le texte chiffré à l'aide du César grâce à cette méthode. Vous observerez que cette méthode est nettement plus rapide que celle « Brute Force », et de plus, elle retourne souvent le résultat correct directement.

Il est important de noter que la méthode n'est pas parfaite, si le texte est trop court, nous pouvons nous retrouver avec des fréquences qui ne sont pas suffisamment représentatives et l'algorithme ne sera alors pas capable de récupérer le texte d'origine.

5.2 Déchiffrage de Vigenère

Voici la dernière méthode à coder pour ce projet. Bien que le Vigenère soit très proche du César, la façon de le décrypter est bien plus complexe. En effet, on ne connaît pas la taille de la clé pour cette méthode, ce qui est un problème majeur. On pourrait utiliser le « Brute Force » pour tester toutes les tailles de clés, mais cela générerait des quantités de données gigantesques (un stockage simple exigera facilement de l'ordre 10'000 GB de mémoire pour un petit texte où l'on teste toutes les clés de taille plus petite que 6 !).

Vous allez donc programmer une méthode qui vous permettra de trouver la taille de clé rapidement, avec un taux de précision élevé, à condition que la clé ne soit pas trop petite, et que le texte ne soit pas trop court.

L'algorithme qu'il vous est demandé d'implémenter est tiré de la section 2.3 de [1] (voir aussi la vidéo : https://www.youtube.com/watch?v=LaWp_Kq0cKs).

5.2.1 Enlever les espaces

Dans toutes nos méthodes de chiffrement, nous avons conservé les espaces pour l'aspect visuel des résultats et faciliter la compréhension. Dans ce qui suit, nous devons cependant les enlever, car sinon les tâches à entreprendre sont sensiblement plus difficiles[7].

Complétez la méthode `removeSpaces(byte[] array)` pour qu'elle retourne un `ArrayList` contenant le texte chiffré sans espace. Notez que `List` est substituable à ArrayList comme nous l'étudierons plus tard. On écrit en effet habituellement :

```
List<Byte> list = new ArrayList<Byte>();
```

plutôt que :

```
ArrayList<Byte> list = new ArrayList<Byte>();
```

7. vous pourrez étudiez le comportement des algorithmes en préservant les espaces dans la partie bonus du projet si vous le souhaitez

5.2.2 Trouver la taille de la clé

Dans cette partie vous compléterez la méthode `vigenereFindKeyLength(List<Byte> cipher)` pour qu'elle se comporte selon les trois étapes indiquées ci-dessous.

Pour la lisibilité du code, pensez à modulariser le code de sorte à ce que chacune des étape fasse l'objet d'une méthode.

Les étapes de l'algorithme vous sont décrites dans ce qui suit. Une petite explication de pourquoi ces étapes aboutissent au résultat voulu sera donnée plus bas.

Etape 1

Pour trouver la taille de la clé, il faut commencer par compter les caractères qui coïncident dans le texte chiffré. Pour ce faire, il faut itérer sur la longueur de ce dernier. À chaque itération i, on décale le texte chiffré de i positions vers la droite et on compare cette version décalée avec l'originale en comptant les lettres identiques (que nous appellerons « coïncidences »).

Soit par exemple le texte chiffré AAFCAWWA. Il s'agira donc de décaler le texte d'origine de 1 vers la droite et comparer cette version avec le texte d'origine comme ceci :

Itération 1 :

A	A	F	C	A	W	W	A	
	A	A	F	C	A	W	W	A

Il s'agit de compter les lettres qui sont similaires dans la version d'origine (en haut) et celle décalée (en bas). Dans ce cas, il y a coïncidence aux indices 1 et 6 (lettre A et W respectivement). Cela fait 2 coïncidences. Il faut stocker ce nombre dans un tableau à l'indice 0, puis recommencer, toujours en gardant le texte d'origine intact en haut :

Itération 2 :

A	A	F	C	A	W	W	A		
		A	A	F	C	A	W	W	A

Ici, il n'y a pas de coïncidence, donc vous mettrez 0 à l'indice 1 du tableau de coïncidences. Vous répéterez ce processus jusqu'à ce que la ligne du bas ne soit plus du tout superposée à la ligne du haut.

Notez qu'il ne peut y avoir coïncidence que pour les cases non vides des séquences à comparer. Il suffit donc de comparer les séquences entre l'indice i et $taille\_sequence - i$.

Etape 2

Une fois le tableau de coïncidence calculé, l'étape suivante consiste à trouver les tailles potentielles de clé. Pour ce faire, il faut identifier les maximums locaux dans la **première moitié** de votre tableau de coïncidences. Une valeur sera considérée comme maximum local si et seulement si les deux valeurs à sa droite et les deux valeurs à sa gauche sont plus petites. Vous stockerez les indices (non pas les valeurs !) de ces maximums locaux, dans un `ArrayList` et **dans l'ordre**. L'élément d'indice 0 est un maximum local si les deux valeurs à sa droite sont plus petites. L'élément d'indice 1 est un maximum local si les deux valeurs à droite sont plus petites et celle d'indice 0 aussi.

Attention, vous ne chercherez les maximums locaux que sur la première moitié du tableau. Lors du calcul des coïncidences, la taille des séquences comparées devient en effet de plus en plus petite. Ceci fait que le nombre de coïncidences diminue drastiquement dans la seconde moitié du tableau. Les données deviennent ainsi moins significatives, baissant la précision générale. Si la taille du tableau est impaire, alors vous considérerez la moitié comme étant le premier entier supérieur à la valeur obtenue par la division par deux (utilisez

la fonction `Math.ceil`).

Etape 3

Dans cette dernière étape, il s'agit de récupérer la taille de la clé. Nous vous recommandons d'utiliser une table associative pour structurer les données calculées (voir l' Annexe C pour ses détails et son utilisation), bien qu'il soit possible de faire sans.
Il s'agit alors de parcourir la liste des indices des maximums locaux pour calculer les différences entre indices consécutifs, celui d'indice plus grand moins celui d'indice plus petit. Ceci donnera la distance entre les deux positions des maximums locaux. Le but est de trouver la distance apparaissant le plus de fois dans votre liste entre deux maximums conjoints. Cette distance sera la taille de la clé que vous devrez retourner. Vous pouvez stocker le résultat de ce traitement dans une table associative dont la clé est la distance et la valeur le nombre d'occurrences de cette distance.

5.2.3 Recouvrer la clé

Nous savons désormais quelle est la taille probable de la clé. Il s'agit maintenant de l'identifier au travers de la méthode
`vigenereFindKey(List<Byte> cipherNoSpace, int keyLength)`.

Il est important de noter que le message passé en paramètre a été purgé des espaces, ce qui facilitera les calculs.

Nous savons que le Vigenère s'applique à la manière du César sur différentes parties du texte avec différentes clés. Il est en fait possible d'utiliser la méthode `ceasarWithFrequencies(byte[] string)` pour récupérer chaque partie de clé selon le procédé décrit ci-dessous.

Soit par exemple le texte chiffré AFSXFSUWM dont on sait que la clé est de taille 3 avec valeurs inconnues xyz.

Il s'agit alors de regrouper les lettres du message chiffré en fonction des lettres de la clé. Prenons l'exemple ci-dessus, où nous voulons trouver les lettres qui ont été encryptées par la première lettre x de la clé. Nous savons que la clé est répétée chaque 3 caractères. Ceci signifie donc que la séquence AXU est chiffrée au moyen de x. Il suffit alors de décrypter ce sous-texte comme un César. Il faut ensuite répéter le processus pour chaque lettre de la clé, ce qui permettra de trouver les 3 symboles constituant la clé !

5.2.4 Fondements de l'algorithme

Voici maintenant quelques explications que vous pouvez lire si vous vous intéressez à pourquoi cet algorithme fonctionne.

L'idée fondamentale est que dans les textes en langue anglaise, comme dans la plupart des autres langues, les différents lettres n'apparaissent pas à la même fréquence. Par conséquent, si l'on choisit au hasard deux lettres dans un texte en anglais, la probabilité qu'elles soient les mêmes sera plus grande que si on les avait choisies dans une séquence aléatoire de caractères. Comme nous l'avons vu précédemment, il y a 11.162% de chance qu'une lettre choisie au hasard dans un texte en anglais soit un 'E'. Dans le même texte chiffré, `cipherText` cette probabilité est « décalée ». Par exemple, si le chiffrement se fait par un décalage de 4 positions, alors il y a 8.497% de chance qu'une lettre choisie au hasard soit un 'E' (car dans ce cas le 'E' chiffre un 'A'). Si l'on choisit au hasard deux lettres dans un texte en anglais, la probabilité qu'elles soient toutes les deux un 'E' vaut $0.11162 \cdot 0.11162$, soit environ 1.25%. Plus précisément, la probabilité de tirer deux mêmes lettres est la somme des carrés p_i^2 pour toute lettre i, où p_i est la probabilité d'occurrence de la lettre i. Pour la langue anglaise cette probabilité est d'environ 6.6%. Notez que si l'on chiffre un texte avec une clé de longueur un, la probabilité de tirer deux mêmes lettres reste la même et vaut 6.6%. Par contre, dans une séquence quelconque de 26 lettres, la probabilité de tirer deux lettres identiques est de $26 \cdot \frac{1}{26^2} = 3.84\%$, donc bien plus faible. Soit maintenant un texte `cipherText` chiffré avec un Vigenère de taille m, si l'on compare

la lettre à la position k avec les lettres aux positions $k+1$, $k+2$, ..., $k+m$ alors la probabilité que les lettres aux positions k et $k+m$ soient les mêmes est plus élevée que pour les autres positions. Par conséquent, tout décalage qui est un multiple de la taille de la clé devrait donner le plus grand nombre de coïncidences. Comme nous travaillons avec des probabilités, il est sensé de s'intéresser à *toutes* les positions avec un grand nombre de coïncidences. La distance entre deux de ces positions est très probablement la taille de la clé.

5.2.5 Combiner

Il ne vous reste plus qu'à remplir la méthode `vigenereWithFrequencies(byte[] cipher)` en appelant les méthodes codées précédemment dans le bon ordre pour récupérer la clé.

5.3 Tout en un

Comme pour le chiffrement, vous implémenterez pour finir la méthode `breakCipher(String cipher, int type)` qui va regrouper plusieurs façons de briser un texte chiffré. Ici, la méthode tentera de décoder seulement 3 types, le César et le Vigenère en utilisant les versions codées lors de cette partie, et le XOR en utilisant la méthode « Brute Force » de la partie 2.

Comme pour la partie 1, cette méthode convertira le `cipher` en bytes, déchiffrera le contenu et retournera le texte décodé. Pour le XOR vous retournerez le résultat de la méthode `arrayToString(byte[][] bruteForceResult)` comme texte décodé.

5.4 Challenge

Le fichier fourni `res/challenge-encrypted.txt` contient un message chiffré à décrypter au moyen des techniques d'analyse des fréquences que vous avez implémentées. Si vous réussissez à le décrypter, suivez les instructions que cela vous donne :-)

Références

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

6 Pour aller plus loin

Dans cette section, **la partie optionnelle** du projet est décrite. L'implémentation de bonus vous permet de compenser des points éventuellement perdus dans la partie obligatoire et/ou de tester vos compétences plus librement sur des aspects qui vous intéressent. La note du projet reste plafonnée à 6 (une fois les points perdus compensés par les bonus, il n'est pas possible d'aller au delà du 6). Notez aussi que la mise en oeuvre de la partie bonus implique que **vous fassiez preuve de plus d'indépendance et les assistants vous aideront en principe moins**.

Vous trouverez ci-dessous deux suggestions de bonus. Si vous avez d'autres idées vous pouvez nous les soumettre via le forum Piazza si vous avez des doutes sur leur faisabilité.

Les bonus peuvent être implémentés dans les fichiers de votre choix. Cela peut notamment être fait dans des fichiers java additionnels à placer dans le dossier `crypto`.

6.1 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.

6.2 Interpréteur de commande

Une autre idée de bonus consiste à implémenter d'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.

Vous pouvez aussi soumettre une autre méthode de chiffrage de votre intérêt qui n'est pas couverte par ce projet.

Annexe

A Utilisation d'octets signés en java

En Java, les octets (`byte`) sont des nombres *signés* utilisant la représentation en complément à 2. Dans cette représentation, le premier bit détermine le signe du nombre. Un `byte` en java possède donc une valeur entre -128 et 127.

De plus, de par le fait de la promotion entière, le type de retour des opérateurs binaires est `int`. Si l'on combine donc deux variables de type `byte` à l'aide d'un opérateur binaire, le résultat sera de type `int` et non `byte`. Il conviendra donc dans ce projet de cast le résultat à nouveau en `byte` car nous allons uniquement nous concentrer sur ce type.

En effet, tout au long de ce projet nous utiliserons par simplification l'encodage ISO_8859_1 pour lequel les caractèressont encodés entre 0 et 255. Comme une `byte` prend une valeur entre -128 et 127, il y a exactement 256 valeurs possibles, ce qui en fait un choix idéal.

Une notion importante retenir : un `byte` en Java est un entier signé codé sur 8 bits, dont 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`.

De plus, les `byte` sont signés. Aussi, lorsque l'on convertit 128 (10000000) en `byte` nous obtenons -128. Donc, les nombres de 128 à 256 sont équivalents à -128 jusqu'à -1 en `byte`. Il est en fait impossible de sortir de cette fourchette de valeur.

Utilisez bien ces propriétés des `byte` pour vous faciliter l'implémentation de méthodes comme le chiffrement de César ou Vigenère.

Les exemples suivants vous donneront une idée de ce ce qu'il se passe si vous calculez avec des bytes en transtypant (casting) des `int` en `byte`. Si vous avez besoin de plus d'informations sur la conversion d'un nombre décimal en sa représentation binaire, penchez vous alors sur votre cours d'AICC :

```
byte b = 127;         // 127 (in decimal) = 0111_1111 (in binary)

//Example: shift by 1:
int i1 = b + 1;       // 127 + 1 = 128 = 1000_0000
byte b1 = (byte) i1; // 1000_0000 = -128

//Example: shift by 10:
int i2 = b + 10;      // 127 + 10 = 137 = 1000_1001
byte b2 = (byte) i2; // 1000_1010 = -119

//Example: shift by 256:
int i3 = b + 256;     // 127 + 256 = 383 = 1_0111_1111
byte b3 = (byte) i3; // 0111_1111 = 127
```

B Générateur aléatoire 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 :

```
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 :

```
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.).

C Les tables associatives

Les tables associatives (« map ») permettent de généraliser la notion d'indice à des types autres que des entiers. Elles permettent d'associer des *valeurs* à des *clés*.

Par exemple :

```
import java.util.Map;
import java.util.HashMap;
import java.util.Map.Entry;

//...
        // String est le type de la clé et Double le type de la valeur
        Map<String, Double> grades = new HashMap<>();
        grades.put("CS107", 6.0); // associe la clé "CS107" à la valeur (note
            ici) 6.0
        grades.put("CS119", 5.5);
         // … idem pour les autres cours auxquels on aimerait associer une
             note

        grades.replace("CS107", 3.0); // remplace la note associée à CS107
            par la nouvelle valeur 3.0 (6.0 devient 3.0)

        // Trois façon d'itérer sur le contenu de la map
```

```java
for (String key : grades.keySet()) {
    //itérer sur les clés
    System.out.println(key+ " " +grades.get(key));
}

for (Double value : grades.values()) {
    //itérer sur les valeurs
    System.out.println(value);
}

for (Entry<String,Double> pair : grades.entrySet()) {
    //itérer sur les paires clé-valeur
    System.out.println(pair.getKey() + " " + pair.getValue());
}
```

La clé d'une Map peut dont être vue comme la généralisation de la notion d'indice. L'interface Java qui décrit les fonctionnalités de base des tables associatives est `Map`, l'implémentation concrète que nous utiliserons est `HashMap`.