INITIATION À LA CRYPTOLOGIE: LES MESSAGES CHIFFRÉS À TRAVERS LES ÂGES

Jonathan CERTES

1 Environnement de travail

1.1 Travail dans une machine virtuelle

Télécharger Oracle VirtualBox :
`https://www.virtualbox.org/wiki/Downloads`
Installer VirtualBox sur sa machine personnelle :
`https://www.virtualbox.org/manual/UserManual.html#installation`
Télécharger la machine virtuelle fournie par l'enseignant.
Importer la machine virtuelle dans VirtualBox :
`https://www.virtualbox.org/manual/UserManual.html#ovf-import-appliance`
Démarrer la machine virtuelle :
`https://www.virtualbox.org/manual/UserManual.html#intro-starting`
Tout le TP sera réalisé dans la machine virtuelle.

1.2 Environnement

La machine virtuelle contient un interpréteur `Python3` et un ensemble de bibliothèques utiles à la découverte de la cryptologie. Des scripts `Python` permettent d'instrumenter ces bibliothèques pour réaliser le chiffrement et le déchiffrement de messages. Ces scripts peuvent être modifiés à l'aide d'environnements de développement ou d'éditeurs de texte tels que `nano`, `vim`, `geany`, etc.

L'ensemble de ces scripts est présent dans la machine virtuelle. Pour procéder à leur lancement, ouvrir un terminal et se placer dans le dossier `tp-crypto` :

```
debian@myhostname:~$ cd tp-crypto/
debian@myhostname:~/tp-crypto/$ ls
```

Le mot de passe pour le compte utilisateur est : `debian`.

1.3 Exécuter un script Python

Lorsque vous écrivez un script Python `exemple.py`, vous pouvez l'exécuter en le passant comme argument à l'interpréteur `python3`. Par exemple :

```
debian@myhostname:~/tp-crypto/$ python3 exemple.py
```

Une autre possibilité est d'ajouter un *shebang* [1] à votre script. Le *shebang*, représenté par `#!`, est un en-tête de fichier texte qui indique au système d'exploitation (de type Unix) que ce fichier n'est pas un fichier binaire mais un script (ensemble de commandes) ; sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.

Ensuite, vous pourrez rendre votre script exécutable et l'exécuter comme un programme à part entière. Par exemple :

```
debian@myhostname:~/tp-crypto$ cat exemple.py
#!/usr/bin/python3
print("Hello world")

debian@myhostname:~/tp-crypto$ chmod +x exemple.py
debian@myhostname:~/tp-crypto$ ./exemple.py
```

1. `https://fr.wikipedia.org/wiki/Shebang`

2 Objectifs

L'objectif de ces séances est de se familiariser avec les primitives cryptographiques simples et de découvrir les différentes classes d'attaques utilisées pour les mettre en défaut. Les algorithmes choisis sont des algorithmes simples, ne nécessitant souvent pas l'usage d'un ordinateur pour être mis en défaut : ici nous nous sensibilisons aux propriétés de sécurité couvertes par la cryptographie. Différents algorithmes de chiffrement vulnérables (utilisés à travers les âges) sont énumérés ; une implémentation est réalisée puis accompagnée d'une attaque permettant de la mettre en défaut.

Attention ! Ce TP n'est pas un cours de cryptographie ! Il s'agit seulement d'une introduction aux principes de base des attaques sur les systèmes cryptographiques.

C'est parti !

3 Introduction

Le **chiffrement** est un procédé de cryptographie grâce auquel on souhaite rendre la compréhension d'un document impossible à toute personne qui n'a pas la clef de (dé)chiffrement. Ce principe est généralement lié au principe d'accès conditionnel.

Bien que le chiffrement puisse rendre secret le sens d'un document, d'autres techniques cryptographiques sont nécessaires pour communiquer de façon sûre. Pour vérifier l'intégrité ou l'authenticité d'un document, on utilise respectivement un code d'authentification de message ou une signature numérique. [...] La sécurité d'un système de chiffrement doit reposer sur le secret de la clef de chiffrement et non sur celui de l'algorithme. Le principe de Kerckhoffs suppose en effet que l'ennemi (ou la personne qui veut déchiffrer le message codé) connaisse l'algorithme utilisé.

Source : `https://fr.wikipedia.org/wiki/Chiffrement`

La **cryptologie**, étymologiquement la "science du secret", n'est considérée comme une science que depuis le XXe siècle. Elle englobe la cryptographie (l'écriture secrète) et la cryptanalyse (l'analyse de cette dernière).

Le terme "crypto" provient du latin et du grec et signifie "ce qui est dissimulé ou caché".

À la fois art ancien et science nouvelle, la cryptologie est utilisée durant l'Antiquité par les Spartiates (la scytale) et elle devient thème de recherche scientifique académique universitaire depuis les années 1970. Cette discipline est liée à beaucoup d'autres, notamment l'arithmétique modulaire, l'algèbre, la théorie de la complexité, la théorie de l'information ou encore les codes correcteurs d'erreurs.

Source : `https://fr.wikipedia.org/wiki/Cryptologie`

3.1 Vocabulaire

- **message clair** : message écrit par le défenseur avant le chiffrement, c'est le message que l'attaquant souhaite obtenir.
- **message chiffré** : message obtenu par le défenseur lors du chiffrement du message clair.
- **chiffrement** : action de transformer un message clair en message chiffré, à partir d'un secret.
- **déchiffrement** : action de transformer un message chiffré en message clair, à partir d'un secret.
- **décryptage** : action de transformer un message chiffré en message clair, sans connaître le secret.
- **cryptage** : ce mot est un anglicisme qui désigne le chiffrement. Étymologiquement, il n'a pas de sens : "action de transformer un message clair en message chiffré, sans connaître le secret" (ce qui est impossible).
- **algorithme de chiffrement** : suite d'opérations effectuées sur un message clair pour le transformer en message chiffré. Cette suite d'opérations est dépendante d'une clef de chiffrement : le secret.
- **clef de chiffrement** : secret permettant la transformation du message clair en message chiffré et vice-versa. Ce secret est connu du défenseur et inconnu de l'attaquant.
- **alphabet** : ensemble des lettres qui composent un message, cet ensemble peut être différent entre le message clair et le message chiffré.

3.2 Modèle de menaces

Dans ce TP, nous attaquons les systèmes cryptographiques pour permettre le décryptage de messages chiffrés. Nous travaillons avec un modèle de menaces relativement faible mais qui respecte le principe de Kerckhoffs. Nous considérons que :

— l'adversaire possède un ordinateur avec une faible puissance de calcul : il ne peut pas réaliser plus de 100 itérations par seconde ;
— l'adversaire n'a pas connaissance du secret (de la clef) qui a été utilisé pour chiffrer le message ;
— l'adversaire a connaissance de l'algorithme qui a été utilisé pour chiffrer le message ;
— l'adversaire a connaissance du langage utilisé dans le message clair.

4 Ensemble des possibles

Une problématique de la cryptographie est de considérer des modèles de menaces où l'adversaire possède une forte puissance de calcul. Dans ce cas, il devient envisageable pour celui-ci de parcourir l'ensemble des possibles pour les clefs de chiffrement et ainsi "casser" le chiffrement. Dans cette section, nous explorons des algorithmes de chiffrement où considérer l'ensemble des possibles est une nécessité.

4.1 Chiffrement par décalage (ou Chiffre de César)

```
KDYSYPJCPBUHHQWUPSAQYGPFJUPCDJHPHDJXQYIDCHPGUCTGUPYCSDCCJPTJCPQTKUGHQYGU
```

En cryptographie, le chiffrement par décalage, aussi connu comme le chiffre de César ou le code de César, est une méthode de chiffrement très simple utilisée par Jules César dans ses correspondances secrètes (ce qui explique le nom "chiffre de César").

Le texte chiffré s'obtient en remplaçant chaque lettre du texte clair original par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. Pour les dernières lettres (dans le cas d'un décalage à droite), on reprend au début.

Postérieur et plus simple que la *scytale*, le chiffre de César doit son nom à Jules César qui, selon Suétone, l'utilisait avec l'alphabet grec (inintelligible pour la plupart des Gaulois mais langue maîtrisée par les élites dirigeantes romaines) et un décalage de trois lettres sur la droite pour certaines de ses correspondances secrètes, notamment militaires.

Source : `https://fr.wikipedia.org/wiki/Chiffrement_par_d%C3%A9calage`

Cette méthode de chiffrement par décalage date donc du premier siècle avant notre ère. Dans cette section, nous implémentons un algorithme de ce chiffrement et un programme permettant de l'attaquer. Rentrons dans le vif du sujet.

4.1.1 Implémentation

Question 1 Le script `tp-crypto/01/decalage.py` contient un squelette prêt pour implémenter l'algorithme de chiffrement et chiffrer un message. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/01
debian@myhostname:~/tp-crypto/01$ ./decalage.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre algorithme.

Question 2 Nous souhaitons implémenter un chiffrement par décalage. Nous devons donc d'abord définir l'alphabet qui sera utilisé pour le message clair. Il sera identique pour le message chiffré.
Dans ce TP, nous définissons l'alphabet comme une chaîne de caractères contenant une fois chaque lettre de l'alphabet. Nous n'utilisons que 27 lettres : les 26 majuscules de l'alphabet latin et l'espace.
Compléter le script pour définir l'alphabet :

```
theAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
```

Question 3 Nous devons ensuite définir une clef de chiffrement. Dans ce cas, c'est un nombre entier, compris entre 1 et 26, qui détermine le décalage.
Prenons par exemple le nombre 16. Compléter le script pour définir la clef de chiffrement :

```
theKey = 16
```

4.1.2 Chiffrement

Question 4 Dans le message clair, chaque lettre de l'alphabet doit être décalée un nombre de fois égal à la clef de chiffrement. Pour chaque lettre, il nous faut donc définir l'index de celle-ci dans l'alphabet.
Compléter le script pour obtenir l'index de la lettre `E` dans l'alphabet et l'afficher :

```
theLetter = "E"
theIndex  = theAlphabet.find(theLetter)
print("Index de la lettre " + theLetter + ": " + str(theIndex))
```

Question 5 Dans le message chiffré, l'image de la lettre est celle située à l'index décalé.
Compléter le script pour obtenir l'image de la lettre `E` dans le message chiffré et l'afficher :

```
theImageIndex = theIndex + theKey
theImage      = theAlphabet[theImageIndex]
print("Image de la lettre " + theLetter + ": " + theImage)
```

Question 6 Avec ce même code, remplacer la lettre `E` par la lettre `M` et l'exécuter de nouveau. Que se passe-t-il ?

Question 7 Le script s'arrête car l'index de l'image (12 + 16) est supérieur à la taille de l'alphabet. La valeur correcte pour l'index de l'image est la somme entre l'index de la lettre et celui de la clef, modulo la taille de l'alphabet.
Corriger le script pour que l'index de l'image soit égal au reste de la division euclidienne de la somme par la taille de l'alphabet :

```
theImageIndex = (theIndex + theKey) % len(theAlphabet)
theImage      = theAlphabet[theImageIndex]
print("Image de la lettre " + theLetter + ": " + theImage)
```

Question 8 Avec ce même code, remplacer la lettre `M` par la lettre `K` et l'exécuter de nouveau. Que se passe-t-il ?

L'espace fait partie de l'alphabet, la lettre `K` est donc substituée par un espace. De même, un espace dans le texte clair sera substitué par la lettre `P` dans le message chiffré.

Question 9 Nous pouvons appliquer ce décalage sur tous les caractères du message clair pour obtenir notre message chiffré.
Écrire une boucle pour obtenir l'image de tous les caractères dans le message clair à partir de la clef :

```
for theLetter in theClearText:
  theIndex      = theAlphabet.find(theLetter)
  theImageIndex = (theIndex + theKey) % len(theAlphabet)
  theImage      = theAlphabet[theImageIndex]
  print(theLetter + " -> " + theImage)
```

Question 10 Nous avons donc terminé notre implémentation. À partir du code précédent, compléter la fonction `tp_cipher` qui retourne le message chiffré à partir de l'alphabet, de la clef et du message clair.

```
def tp_cipher(
  theAlphabet,
  theKey,
  theClearText
):
  theCipherText = ""

  for theLetter in theClearText:
    theIndex      = theAlphabet.find(theLetter)
    theImageIndex = (theIndex + theKey) % len(theAlphabet)
    theImage      = theAlphabet[theImageIndex]
    #
    theCipherText += theImage

  return theCipherText
```

Tester la fonction en l'appelant avec les données précédentes :

```
theCipherText = tp_cipher(theAlphabet, theKey, theClearText)
print("CLEAR TEXT  = " + theClearText)
print("CIPHER TEXT = " + theCipherText)
```

Le résultat doit ressembler à ceci :

```
CLEAR TEXT  = VOICI UN MESSAGE CLAIR QUE NOUS SOUHAITONS RENDRE INCONNU DUN ADVERSAIRE
CIPHER TEXT = KDYSYPJCPBUHHQWUPSAQYGPFJUPCDJHPHDJXQYIDCHPGUCTGUPYCSDCCJPTJCPQTKUGHQYGU
```

4.1.3 Déchiffrement

Pour le chiffrement par décalage, nous avons remplacé chaque lettre du texte clair par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. La clef de chiffrement indique la distance effectuée lors du décalage. Pour réaliser le déchiffrement, il suffit donc d'effectuer le décalage inverse : remplacer chaque lettre chiffrée par une lettre à la même distance, de l'autre côté dans l'alphabet.

Question 11 Copier et modifier le code de la fonction `tp_cipher` pour obtenir celui de la fonction `tp_decipher`, qui retourne le message clair à partir de l'alphabet, de la clef et du message chiffré.
L'obtention de l'index de l'image du caractère ne s'obtient donc plus par addition de la clef, mais par soustraction.

```
def tp_decipher(
  theAlphabet,
  theKey,
  theCipherText
):
  theClearText = ""

  for theLetter in theCipherText:
    theIndex      = theAlphabet.find(theLetter)
    theImageIndex = (theIndex - theKey) % len(theAlphabet)
    theImage      = theAlphabet[theImageIndex]
    #
    theClearText += theImage

  return theClearText
```

Tester la fonction en l'appelant avec les données précédentes :

```
theCipherText   = tp_cipher(  theAlphabet, theKey, theClearText)
theDecipherText = tp_decipher(theAlphabet, theKey, theCipherText)
print("CIPHER TEXT   = " + theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

Le résultat doit ressembler à ceci :

```
CIPHER TEXT   = KDYSYPJCPBUHHQWUPSAQYGPFJUPCDJHPHDJXQYIDCHPGUCTGUPYCSDCCJPTJCPQTKUGHQYGU
DECIPHER TEXT = VOICI UN MESSAGE CLAIR QUE NOUS SOUHAITONS RENDRE INCONNU DUN ADVERSAIRE
```

4.2 Attaque par force brute

Le chiffre de César peut être cassé très facilement, même à l'aide du seul texte chiffré. Dans notre modèle de menaces, l'adversaire a connaissance de l'algorithme qui a été utilisé pour chiffrer le message mais n'a pas connaissance du secret (de la clef). En d'autres termes, il sait que le chiffre de César a été utilisé, mais ignore la valeur du décalage.

Comme il n'y a qu'un nombre limité de décalages (26), il suffit de tester toutes les clefs possibles jusqu'à trouver la bonne. C'est ce que l'on appelle **une attaque par force brute** : technique de test de toutes les combinaisons possibles. Une méthode simple pour mener l'attaque est de prendre un fragment du texte chiffré et d'écrire les messages obtenus par tous les décalages possibles.

Question 12 Admettons que nous soyons en possession d'un message chiffré à l'aide du chiffrement par décalage et que nous pouvons exécuter 1 itération par seconde. Combien de temps serait nécessaire pour parcourir l'ensemble des possibles si nous effectuons une attaque par force brute sur la clef ?

Question 13 Voici un message qui a été chiffré à l'aide du chiffrement par décalage, avec une clef inconnue :

```
LBTMNTULTKYNLLBTUTWULLYKTEYTWABZZKYFYGMTXYTWYLUKTMUIYTXUGLTMYLTFUBGL
```

Proposer un algorithme qui implémente une attaque par force brute et décrypte le message.
Quelle clef de chiffrement a été utilisée pour chiffrer ce message ?

Question 14 Si vous avez obtenu la clef de chiffrement, c'est que vous avez pu identifier quel était le message clair lors de l'attaque par force brute. Quel moyen avez vous employé pour déterminer le message clair ?

Solutions : Pour implémenter l'attaque par force brute, nous devons appeler la fonction `tp_decipher` avec les 26 valeurs possibles pour la clef de chiffrement, soit les valeurs comprises entre 1 et 26. Le code suivant permet de jouer l'attaque :

```
theCipherText = "LBTMNTULTKYNLLBTUTWULLYKTEYTWABZZKYFYGMTXYTWYLUKTMUIYTXUGLTMYLTFUBGL"
print("CIPHER TEXT = " + theCipherText)

for i in range(1, 27):
  print(format(i, "02d") + ": " + tp_decipher(theAlphabet, i, theCipherText))
```

Pour déterminer lequel des messages est le message clair, nous cherchons celui qui décrit du texte en français. En effet, dans notre modèle de menaces, l'adversaire a connaissance du langage utilisé dans le message clair.

4.3 Chiffrement par substitution monoalphabétique

```
(@4+[47)8)+@(+[8}[-])220}>}(7[_10[+4/+7)747)@([_@40[+}[_0@7}*}0[-@(70}[8}+[177154}+[_10[2@0-}[/047}
```

La faiblesse du chiffrement par décalage est donc que la clef de chiffrement possède un ensemble des possibles réduit. Pour pallier cette faiblesse, nous pouvons choisir d'attribuer arbitrairement un décalage différent à chaque lettre de l'alphabet. Cette méthode de chiffrement s'appelle le **chiffrement par substitution**.

Le chiffrement par substitution est une technique de chiffrement utilisée depuis bien longtemps puisque le chiffre de César en est un cas particulier. Sans autre précision, elle désigne en général un chiffrement par substitution *monoalphabétique*, qui consiste à substituer dans un message chacune des lettres de l'alphabet par une autre (du même alphabet ou éventuellement d'un autre alphabet).

Source : `https://fr.wikipedia.org/wiki/Chiffrement_par_substitution`

Nous pouvons citer plusieurs implémentations connues du chiffrement par substitution :

- le carré de Polybe [2], deuxième siècle avant notre ère. Il est utilisée par plusieurs civilisations de différentes manières tout au long de l'histoire.
- le chiffre des Templiers [3] (ou Chiffre de Cornelius Agrippa), 16$^{\text{ième}}$ siècle.
- le chiffre de PigPen [4] (ou Chiffre des francs-maçons), 16$^{\text{ième}}$ siècle. Il possède surtout une valeur pédagogique, artistique et, dans une certaine mesure, historique.
- le chiffre de Delastelle [5], 19$^{\text{ième}}$ siècle. D'après une étude réalisée en 2021, cette dernière aurait été utilisée dans les années 1970 par le tueur du Zodiaque pour certains de ses cryptogrammes.

On retrouve aussi le chiffrement par substitution dans la culture populaire, par exemple :

- le langage *Al Bhed* dans le jeu vidéo *Final Fantasy X*,
- une énigme de message à déchiffrer dans le jeu vidéo *BioShock Infinite*,
- l'alphabet *Minbari* dans la série télévisée *Babylon 5*,
- le langage Alien dans la série animée *Futurama*,
- etc.

Dans cette section, nous implémentons un algorithme de chiffrement par substitution et un programme permettant de l'attaquer.

2. `https://fr.wikipedia.org/wiki/Carr%C3%A9_de_Polybe`
3. `https://fr.wikipedia.org/wiki/Chiffre_de_Cornelius_Agrippa`
4. `https://fr.wikipedia.org/wiki/Chiffre_des_francs-ma%C3%A7ons`
5. `https://fr.wikipedia.org/wiki/Chiffre_de_Delastelle`

4.3.1 Attaque par force brute ?

Nous pouvons substituer chaque lettre de l'alphabet par une autre lettre de manière arbitraire. Par exemple, nous pouvons substituer la lettre "A" par un "E", puis la lettre "B" par un "Y", etc.
Une fois toutes les substitutions réalisées, un tableau de correspondance constitue notre clef de chiffrement.

Question 15 Nous utilisons toujours un alphabet de 27 lettres : les 26 majuscules de l'alphabet latin et l'espace. Combien de possibilités avons-nous pour choisir par quelle lettre nous allons substituer la lettre "A" ?

Question 16 Une fois que nous avons choisi comment substituer la lettre "A", combien de possibilités avons-nous pour choisir par quelle lettre nous allons substituer la lettre "B" ?

Question 17 En déduire le nombre de possibilités (l'ensemble des possibles) pour la clef de chiffrement.

Question 18 Admettons que nous soyons en possession d'un message chiffré à l'aide du chiffrement par substitution et que nous pouvons tester 100 clefs par seconde. Combien de temps serait nécessaire pour parcourir l'ensemble des possibles si nous effectuons une attaque par force brute sur la clef ?
Note : pour donner un ordre de grandeur, en 2020, une étude de l'Université de Cornell, publiée dans *Journal of Cosmology and Astroparticle Physics*, fournit une estimation de l'âge de l'Univers à environ 13,77 milliards d'années.

Solutions : Pour substituer la lettre "A", nous avons 26 possibilités (ou 27 si l'on considère qu'une lettre peut être substituée par elle-même). Pour substituer la lettre "B", nous avons 25 possibilités (ou 26). Etc.
Le nombre total de possibilités pour la clef est donc de : $26 \times 25 \times 24...$, soit !26.
Nous pouvons donc calculer le temps nécessaire avec le script `Python` suivant :

```
import math
theNum = math.factorial(26)
print("Il faudrait " + str(theNum / 100) + " secondes")
print("soit " + str(theNum / 100 / 3600) + " heures")
print("soit " + str(theNum / 100 / 3600 / 24) + " jours")
print("soit " + str(theNum / 100 / 3600 / 24 / 365.25) + " annees")
print("soit " + str(theNum / 100 / 3600 / 24 / 365.25 / 13.77e9) + " fois l'age de l'univers")
```

4.3.2 Implémentation

Question 19 Le script `tp-crypto/02/substitution.py` contient un squelette prêt pour implémenter l'algorithme de chiffrement et chiffrer un message. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/02
debian@myhostname:~/tp-crypto/02$ ./substitution.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre algorithme.

Question 20 Ici aussi, nous définissons l'alphabet comme une chaîne de caractères contenant une fois chaque lettre de l'alphabet. Nous n'utilisons que 27 lettres : les 26 majuscules de l'alphabet latin et l'espace.
Compléter le script pour définir l'alphabet :

```
theAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
```

Question 21 Nous devons ensuite définir une clef de chiffrement. Dans ce cas, c'est une correspondance entre les caractères de l'alphabet du message clair et ceux du message chiffré. Nous pouvons par exemple, mélanger l'alphabet de manière aléatoire :

```
theCipherAlphabet = "".join(random.sample(theAlphabet, k=len(theAlphabet)))
print(theAlphabet)
print(theCipherAlphabet)
```

Voici un exemple de résultat d'exécution de ce script :

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
UCR DLYJZGQSBIPATWKFMHNVOEX
```

Dans cet exemple, la lettre "A" sera substituée par un "U", la lettre "B" par un "C", etc., puis l'espace sera substitué par un "X".

Question 22 Afin de pimenter l'implémentation, nous allons modifier l'alphabet du message chiffré pour remplacer chaque lettre par un symbole. Ajouter au script la définition suivante pour l'alphabet du message chiffré :

```
theCipherAlphabet = "1/-%}2*])<68>(@_50+74{#=93["
```

Avec cette clef de chiffrement, la lettre "A" sera substituée par un "1", la lettre "B" par un "/", etc.

Question 23 Vérifier que, pour les deux alphabets, un caractère n'apparait pas en double dans ceux-ci. La fonction `tp.verifyAlphabet()` permet d'effectuer cette vérification.
Vérifier aussi que les deux alphabets ont la même longueur.

```
tp.verifyAlphabet(theAlphabet)
tp.verifyAlphabet(theCipherAlphabet)
if ( len(theCipherAlphabet) != len(theAlphabet) ):
  print("** Error ** the two alphabets do not have the same size.")
```

4.3.3 Chiffrement

Question 24 Dans le message clair, chaque lettre de l'alphabet doit être substituée par le symbole placé à la même position dans l'alphabet du message chiffré. Pour chaque lettre, il nous faut donc définir l'index de celle-ci dans l'alphabet.
Compléter le script pour obtenir l'index de la lettre `E` dans l'alphabet et l'afficher :

```
theLetter = "E"
theIndex  = theAlphabet.find(theLetter)
print("Index de la lettre " + theLetter + ": " + str(theIndex))
```

Question 25 Dans le message chiffré, l'image de la lettre est celle située au même index dans l'alphabet du message chiffré.
Compléter le script pour obtenir l'image de la lettre `E` dans le message chiffré et l'afficher :

```
theImage = theCipherAlphabet[theIndex]
print("Image de la lettre " + theLetter + ": " + theImage)
```

Question 26 Nous pouvons appliquer cette substitution sur tous les caractères du message clair pour obtenir notre message chiffré.
Écrire une boucle pour obtenir l'image de tous les caractères dans le message clair à partir de l'alphabet du message chiffré :

```
for theLetter in theClearText:
  theIndex = theAlphabet.find(theLetter)
  theImage = theCipherAlphabet[theIndex]
  print(theLetter + " -> " + theImage)
```

Question 27 Nous avons donc terminé notre implémentation. À partir du code précédent, compléter la fonction `tp_cipher` qui retourne le message chiffré à partir de l'alphabet, de la clef et du message clair.

```
def tp_cipher(
  theClearAlphabet,
  theCipherAlphabet,
  theClearText
):
  theCipherText = ""
```

```
  for theLetter in theClearText:
    theIndex = theClearAlphabet.find(theLetter)
    theImage = theCipherAlphabet[theIndex]
    #
    theCipherText += theImage

  return theCipherText
```

Tester la fonction en l'appelant avec les données précédentes :

```
theCipherText = tp_cipher(theAlphabet, theCipherAlphabet, theClearText)
print("CLEAR TEXT  = " + theClearText)
print("CIPHER TEXT = " + theCipherText)
```

Le résultat doit ressembler à ceci :

```
CLEAR TEXT  = ICI NOUS UTILISONS LE CHIFFREMENT PAR SUBSTITUTION POUR SE PROTEGER CONTRE LES ATTAQUES
 PAR FORCE BRUTE
CIPHER TEXT = )-)[(@4+[47)8)+@(+[8}[-])220}>}(7[_10[+4/+7)747)@([_@40[+}[_0@7}*}0[-@(70}[8}+[177154}+
[_10[2@0-}[/047}
```

4.3.4 Déchiffrement

Pour le chiffrement par substitution, nous avons remplacé chaque lettre du texte clair par un symbole positionné au même index dans l'alphabet du texte chiffré. La clef de chiffrement est simplement l'alphabet du texte chiffré, qui définit les correspondances via l'index des symboles. Pour effectuer le déchiffrement, il suffit donc d'effectuer la substitution inverse : remplacer chaque symbole par la lettre positionnée au même index dans l'alphabet.

Question 28 Copier et modifier le code de la fonction `tp_cipher` pour obtenir celui de la fonction `tp_decipher`, qui retourne le message clair à partir de l'alphabet, de la clef et du message chiffré.

```
def tp_decipher(
  theClearAlphabet,
  theCipherAlphabet,
  theCipherText
):
  theClearText = ""

  for theImage in theCipherText:
    theIndex  = theCipherAlphabet.find(theImage)
    theLetter = theClearAlphabet[theIndex]
    #
    theClearText += theLetter

  return theClearText
```

Tester la fonction en l'appelant avec les données précédentes :

```
theCipherText   = tp_cipher(theAlphabet, theCipherAlphabet, theClearText)
theDecipherText = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)
print("CIPHER TEXT   = " + theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

Le résultat doit ressembler à ceci :

```
CIPHER TEXT   = )-)[(@4+[47)8)+@(+[8}[-])220}>}(7[_10[+4/+7)747)@([_@40[+}[_0@7}*}0[-@(70}[8}+[177154
}+[_10[2@0-}[/047}
DECIPHER TEXT = ICI NOUS UTILISONS LE CHIFFREMENT PAR SUBSTITUTION POUR SE PROTEGER CONTRE LES ATTAQU
ES PAR FORCE BRUTE
```

4.4 Attaque par analyse de fréquences

Nous avons vu que le chiffrement par substitution n'est pas vulnérable aux attaques par force brute. Ceci est dû au fait que le nombre de clefs possibles est très important.

Cependant, le chiffrement par substitution est facile à casser par **analyse des fréquences** des lettres du texte chiffré. Notons qu'il demeure tout de même en tant que composant élémentaire des chiffrements modernes (ce sont les S-Boxes des réseaux de substitution-permutation [6]).

L'analyse fréquentielle, ou analyse de fréquences, est une méthode de cryptanalyse dont la description la plus ancienne est réalisée par Al-Kindi au 9ième siècle. Elle consiste à examiner la fréquence des lettres employées dans un message chiffré. [...]

L'analyse fréquentielle est basée sur le fait que, dans chaque langue, certaines lettres ou combinaisons de lettres apparaissent avec une certaine fréquence. Par exemple, en français, le "E" est la lettre la plus utilisée, suivie du "A" et du "S". Inversement, le "W" est peu utilisé.

Ces informations permettent aux cryptanalystes de faire des hypothèses sur le texte clair, à condition que l'algorithme de chiffrement conserve la répartition des fréquences, ce qui est le cas pour des substitutions *monoalphabétiques* et polyalphabétiques. Une deuxième condition, nécessaire pour appliquer cette technique, est la longueur du cryptogramme. En effet, un texte trop court ne reflète pas obligatoirement la répartition générale des fréquences des lettres.

Source : `https://fr.wikipedia.org/wiki/Analyse_fr%C3%A9quentielle`

Voici un message qui a été chiffré à l'aide du chiffrement par substitution, avec une clef inconnue :

```
0>{4>)6>{8(-+{#(7#-_{>_>{1->){=>{<\%))#-_7>{<>__>{6_7#_>}->{=>{=><@-227>[>)_{8(#)={\%){#9#-_{6>4_{#)6
{4\%(7{4\%(9\%-7{_>7[-)>7{+>6{0>(*{6(7{+>6{4#8(>_6{=>{<>7>#+>6{>_{2#-7>{+>{[#+-){=>9#)_{+>6{<\%4#-)6{
=#)6{+#{<\%(7{=>{+><\%+>
```

Dans cette section, nous allons attaquer ce message pour extraire des informations sur la clef de chiffrement. C'est ce que nous appelons une **attaque sur texte chiffré seul**.

Question 29 Le script `tp-crypto/03/analyseFreq.py` contient un squelette prêt pour implémenter une attaque par analyse des fréquences. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/03
debian@myhostname:~/tp-crypto/03$ ./analyseFreq.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre attaque.

Ce script tente de déchiffrer le message sans connaître la clef de chiffrement : le résultat est un message illisible. Notre travail consiste donc à "casser" le chiffrement en découvrant un sous-ensemble de la clef suffisant pour décrypter le message (pas besoin de connaitre toute la clef si certaines lettres de l'alphabet n'apparaissent pas dans le message).

Pour réaliser une analyse de fréquences, nous devons connaître la fréquence d'apparition de chaque lettre dans la langue française. Bien que cette information se trouve facilement sur le Web, nous allons procéder nous-même à une analyse statistique depuis un texte écrit en français.

Question 30 Le fichier `~/tp-crypto/jverne.txt` contient le texte du roman de Jules Verne : Vingt mille Lieues Sous Les Mers. Ce texte est disponible dans le domaine public et est distribué librement par le projet Gutenberg [7].

La fonction `tp.formatText()` permet de modifier une chaîne de caractères pour n'utiliser que l'alphabet précédent (contenant les 26 majuscules de l'alphabet latin et l'espace).
Modifier le script `Python` pour lire le contenu fichier et formater le texte pour utiliser cet alphabet.
Attention à bien se placer dans le dossier contenant le script avant de l'exécuter (utilisation d'un chemin relatif).

```
theFile   = "../jverne.txt"
theBuffer = open(theFile, "r")
theText   = theBuffer.read()
theBuffer.close()
theText   = tp.formatText(theText)
```

6. `https://fr.wikipedia.org/wiki/R%C3%A9seau_de_substitution-permutation`
7. `https://www.gutenberg.org`

Question 31 Créer une liste de nombre réels contenant autant d'éléments que la taille de l'alphabet (27). Pour chaque lettre de l'alphabet, compter le nombre d'apparition dans le texte et en déduire la fréquence d'apparition (nombre d'apparition divisé par le nombre total de caractères).

```
theFreqList = [0.0] * len(theAlphabet)
for i in range(0, len(theAlphabet)):
  theFreqList[i] = theText.count(theAlphabet[i]) / len(theText)
#
print(theFreqList)
```

Question 32 Tracer un diagramme représentant la fréquence d'apparition de chaque lettre de l'alphabet dans le texte.

```
x = range(0, len(theAlphabet))
y = theFreqList
fig, ax = matplotlib.pyplot.subplots()
ax.bar(x, y)
ax.set(xticks=range(0, len(theAlphabet)), xticklabels=list(theAlphabet))
ax.set_title("Frequency for each char")
matplotlib.pyplot.show()
```

Quel est le caractère qui apparaît le plus ?
Quel est le second caractère qui apparaît le plus ?
Quel est le troisième caractère qui apparaît le plus ?

Question 33 Nous allons maintenant réaliser la même étude dans le message chiffré pour déterminer quels sont les caractères qui apparaissent le plus souvent.
Pour chaque lettre de l'alphabet du message chiffré, compter le nombre d'apparition dans le message et en déduire la fréquence d'apparition.

```
theFreqList = [0.0] * len(theCipherAlphabet)
for i in range(0, len(theCipherAlphabet)):
  theFreqList[i] = theCipherText.count(theCipherAlphabet[i]) / len(theCipherText)
#
print(theFreqList)
```

Question 34 Tracer un diagramme représentant la fréquence d'apparition de chaque symbole dans le message chiffré.

```
x = range(0, len(theCipherAlphabet))
y = theFreqList
fig, ax = matplotlib.pyplot.subplots()
ax.bar(x, y)
ax.set(xticks=range(0, len(theCipherAlphabet)), xticklabels=list(theCipherAlphabet))
ax.set_title("Frequency for each symbol in the cipher text")
matplotlib.pyplot.show()
```

Quel est le symbole qui apparaît le plus ?
Quel est le second symbole qui apparaît le plus ?
Quel est le troisième symbole qui apparaît le plus ?

Question 35 Pour réaliser notre analyse de fréquence, nous allons remplacer les symboles qui apparaissent le plus souvent par des caractères qui apparaissent fréquemment dans le texte. Nous allons effectuer ceci pour les premiers caractères et tenter de décrypter le message en cherchant des mots probables dans le message.
Modifier l'alphabet du message chiffré, remplacer :

- le symbole "{" par un espace,
- le symbole ">" par la lettre "E",
- le symbole "#" par la lettre "A",
- tous les autres symboles par des points.

Ensuite, tenter de déchiffrer le message. Garder en tête que les symboles suivants qui apparaissent souvent sont : _,), -, 6 et 7.

```
theCipherAlphabet = "#%()*+-/<=>@[]_{}0123456789"
theAlphabet       = "A.........E.... ..........."
theDecipherText = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)
print("CIPHER TEXT   = " + theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

Le résultat doit ressembler à ceci :

```
CIPHER TEXT   = 0>{4>)6>{8(-+{#(7#-_{>_>{1->){=>{<%))#-_7>{<>__>{6_7#_>}->{=>{=><@-227>[>)_{8(#)={%){
#9#-_{6>4_{#)6{4%(7{4%(9%-7{_>7[-)>7{+>6{0>(*{6(7{+>6{4#8(>_6{=>{<>7>#+>6{>_{2#-7>{+>{[#+-){=>9#)_{+>
6{<%4#-)6{=#)6{+#{<%(7{=>{+><%+>
DECIPHER TEXT = .E .E..E .... A..A.. E.E ..E. .E ....A...E .E..E ...A.E..E .E .E......E.E.. ..A.. ..
A.A.. .E.. A.. .... ....... .E....E. .E. .E.. ... .E. .A..E.. .E .E.EA.E. E. .A..E .E .A... .E.A.. .
E. ...A... .A.. .A .... .E .E...E
```

Question 36 Certains mots peuvent être déduits de cette première itération. Par exemple, le mot "`E.E`" est probablement "`ETE`". Nous pouvons donc en déduire que le symbole "_" est la lettre "T".
Faire une nouvelle itération, modifier l'alphabet du message chiffré et remplacer le symbole "_" par la lettre "T". Garder en tête que les symboles `)`, `-`, `6` et `7` sont probablement des lettres comme "I", "N", "R", "S" et "U" (comme montré par l'analyse de fréquence du roman de Jules Verne).

```
theCipherAlphabet = "#%()*+-/<=>@[]_{}0123456789"
theAlphabet       = "A.........E...T ..........."
theDecipherText = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)
print("CIPHER TEXT   = " + theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

Question 37 Cette nouvelle itération fait apparaitre le groupe de mots "`A..A.T ETE`" qui contient les symboles `)`, `-` et `7`. Nous pouvons donc en déduire que le groupe de mots doit être "`AURAIT ETE`".
Faire une nouvelle itération, modifier l'alphabet du message chiffré en conséquence.

```
theCipherAlphabet = "#%()*+-/<=>@[]_{}0123456789"
theAlphabet       = "A.U...I...E...T ........R.."
theDecipherText = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)
print("CIPHER TEXT   = " + theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

Question 38 Nouvelle itération : "`.ETTE .TRATE.IE`" → "`CETTE STRATEGIE`".

```
theCipherAlphabet = "#%()*+-/<=>@[]_{}0123456789"
theAlphabet       = "A.U...I.C.E...T G......SR.."
```

Question 39 etc. etc. Nous pouvons jouer à déterminer les substitutions suivantes par déduction et par itérations successives.

".E C...AITRE CETTE STRATEGIE"	→	"DE CONNAITRE CETTE STRATEGIE"
"A.AIT SE.T ANS"	→	"AVAIT SEPT ANS"
"DEVANT .ES COPAINS"	→	"DEVANT LES COPAINS"

```
theCipherAlphabet = "#%()*+-/<=>@[]_{}0123456789"
theAlphabet       = "AOUNXLI.CDEHM.T GJBF.P.SRQV"
theDecipherText = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)
print("CIPHER TEXT   = " + theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

Certaines substitutions ne peuvent pas être déterminées car les symboles n'apparaissent pas dans le message chiffré.

Le chiffrement par substitution est donc vulnérable aux attaques par analyse de fréquences. Ces attaques demandent une connaissance du langage du texte clair et de l'intuition / de la déduction. Pour automatiser les attaques sur ces algorithmes de chiffrement, il est préférable de se tourner vers des attaques plus répétitives qui ne demandent pas d'intuition. C'est le cas, par exemple, des attaques Monte-Carlo par chaînes de Markov que nous verrons dans la section 5.

4.5 Chiffrement par substitution polyalphabétique (ou Chiffre de Vigenère)

```
XEMILGEIVWZSSLGMYQNRXIYQHW RHSLMDCMNNRBQDCHWZUEQBYWZCRY
```

Le **chiffre de Vigenère** est un système de chiffrement par substitution *polyalphabétique* dans lequel une même lettre du message clair peut, suivant sa position dans celui-ci, être remplacée par des lettres différentes, contrairement à un système de chiffrement monoalphabétique. Cette méthode résiste ainsi à l'analyse de fréquences, ce qui est un avantage décisif sur les chiffrements monoalphabétiques. [...]

Il est nommé ainsi au 19$^{\text{ième}}$ siècle en référence au diplomate du 16$^{\text{ième}}$ siècle **Blaise de Vigenère**, qui le décrit (intégré à un chiffrement plus complexe) dans son traité des chiffres paru en 1586. On trouve en fait déjà une méthode de chiffrement analogue dans un court traité de **Giovan Battista Bellaso** paru en 1553.

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

Source : `https://fr.wikipedia.org/wiki/Chiffre_de_Vigen%C3%A8re`

Nous pouvons citer plusieurs implémentations dans la culture populaire, par exemple :

- dans le film Benjamin Gates et le Trésor des Templiers, un code au dos de la déclaration d'indépendance et chiffré selon le code de Vigenère et fait partie de l'énigme pour trouver le trésor des Templiers.
- dans la série télévisée américaine Sleepy Hollow, un manuscrit permettant de combattre le Cavalier Sans Tête est codé selon le chiffre de Vigenère.
- dans la série télévisée américaine Souvenirs de Gravity Falls, des cryptogrammes dans le générique de fin, à partir de la saison 2, peuvent être décryptés grâce au chiffre de Vigenère.

4.5.1 Implémentation

Question 40 Le script `tp-crypto/04/vigenere.py` contient un squelette prêt pour implémenter l'algorithme de chiffrement et chiffrer un message. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/04
debian@myhostname:~/tp-crypto/04$ ./vigenere.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre algorithme.

Question 41 Ici aussi, nous définissons l'alphabet comme une chaîne de caractères contenant une fois chaque lettre de l'alphabet. Nous n'utilisons que 27 lettres : les 26 majuscules de l'alphabet latin et l'espace.
Compléter le script pour définir l'alphabet :

```
theAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
```

Question 42 Nous devons ensuite définir un secret partagé en guise de clef de chiffrement. Dans ce cas, c'est une chaîne de caractère, de longueur arbitraire, comportant des lettres du même alphabet que le texte clair. Ajouter au script la définition suivante pour le secret partagé :

```
theSecret = "MANGEZ DES POMMES"
```

4.5.2 Chiffrement

En parallèle du message clair, nous allons recopier en boucle notre secret partagé. Pour chaque position dans le message clair, nous allons observer la lettre du message et la lettre du secret. Pour chacune de ces deux lettres, nous allons trouver leurs indexes dans l'alphabet que nous allons additionner : le résultat donnera la position dans l'alphabet de la lettre dans le message chiffré.
Par exemple, si nous choisissons comme secret partagé le mot `SECRET`, nous pouvons substituer la première lettre du message (à la position 0) ainsi :

```
(C)ECI EST MON MESSAGE     # index de "C" = 2
(S)ECRETSECRETSECRETSE     # index de "S" = 18
 U                         # 18 + 2 = 20 -> "U"
```

L'avantage de cette approche est que la même lettre de l'alphabet n'est pas substituée par la même image selon sa position dans le message. Un adversaire ne peut donc pas réaliser une attaque par analyse de fréquences (comme nous l'avons fait pour le chiffrement monoalphabétique).
Par exemple, pour la lettre `C` située à la position 2 du message, nous la substituons ainsi :

```
CE(C)I EST MON MESSAGE     # index de "C" = 2
SE(C)RETSECRETSECRETSE     # index de "C" = 2
U  E                       # 2 + 2 = 4 -> "E"
```

Selon certaines conditions sur la composition du secret partagé (longueur du secret, entropie, etc.), il devient impossible de réaliser une attaque à chiffré seul.

Question 43 Dans le message clair, chaque lettre doit donc être substituée par une image qui dépend de son index dans l'alphabet. Pour chaque lettre du message clair, il nous faut donc définir l'index de celle-ci dans l'alphabet.
Compléter le script pour obtenir l'index de la lettre `E` dans l'alphabet et l'afficher :

```
theLetter = "E"
theIndex  = theAlphabet.find(theLetter)
print("Index de la lettre \"" + theLetter + "\": " + str(theIndex))
```

Question 44 L'image de la lettre dépend également de sa position dans le message clair. Nous devons donc définir quelle lettre du secret sera utilisée pour la substitution.
Compléter le script pour définir quelle lettre du secret sera utilisée si la lettre `E` est placée à la position 0 dans le message clair. En déduire l'index dans l'alphabet de cette lettre secrète.

```
thePosition     = 0
theSecretLetter = theSecret[thePosition % len(theSecret)]
theSecretIndex  = theAlphabet.find(theSecretLetter)
print("Lettre du secret a la position " + str(thePosition) + ": " + theSecretLetter)
print("Index de la lettre secrete \"" + theSecretLetter + "\": " + str(theSecretIndex))
```

Question 45 Connaissant l'index de la lettre `E` du message clair et l'index de la lettre secrète, effectuer une addition pour obtenir l'index de la lettre dans le message chiffré. En déduire l'image de la lettre `E` si celle-ci est placée à la position 0 du message clair.

```
theImageIndex = (theIndex + theSecretIndex) % len(theAlphabet)
theImage      = theAlphabet[theImageIndex]
print("Image de la lettre " + theLetter + " a la position " + str(thePosition) + ": " + theImage)
```

Question 46 Modifier la position 0 par 1, puis 2, et exécuter le même code pour obtenir l'image de la même lettre `E` si celle-ci est positionnée ailleurs dans le message clair. Observer que son image change. Que remarquez-vous de particulier ?
Résultat attendu :

```
Image de la lettre E a la position 0: Q
Image de la lettre E a la position 1: E
Image de la lettre E a la position 2: R
```

Question 47 Nous avons donc terminé notre implémentation. À partir du code précédent, compléter la fonction `tp_cipher()` qui retourne le message chiffré à partir de l'alphabet, du secret et du message clair.

```
def tp_cipher(
  theAlphabet,
  theSecret,
  theClearText
):
  theCipherText = ""

  for thePosition in range(0, len(theClearText)):
    theLetter = theClearText[thePosition]
    theIndex  = theAlphabet.find(theLetter)
    #
    theSecretLetter = theSecret[thePosition % len(theSecret)]
    theSecretIndex  = theAlphabet.find(theSecretLetter)
    #
    theImageIndex = (theIndex + theSecretIndex) % len(theAlphabet)
    theImage      = theAlphabet[theImageIndex]
    #
    theCipherText += theImage

  return theCipherText
```

Tester la fonction en l'appelant avec les données précédentes :

```
theCipherText = tp_cipher(theAlphabet, theSecret, theClearText)
print("CLEAR TEXT  = " + theClearText)
print("CIPHER TEXT = " + theCipherText)
```

Le résultat doit ressembler à ceci :

```
CLEAR TEXT  = LE CHIFFRE DE VIGENERE RESISTE A LANALYSE DE FREQUENCES
CIPHER TEXT = XEMILGEIVWZSSLGMYQNRXIYQHW RHSLMDCMNNRBQDCHWZUEQBYWZCRY
```

Observer les différentes substitutions de la lettre `E` :

```
CLEAR TEXT  = L(E) CHIFFR(E) D(E) VIG(E)N(E)R(E) R(E)SIST(E) A LANALYS(E) D(E) FR(E)QU(E)NC(E)S
CIPHER TEXT = X(E)MILGEIV(W)ZS(S)LGMY(Q)N(R)X(I)YQ(H)W RH(S)LMDCMNNRBQ(D)CH(W)ZUE(Q)BY(W)ZC(R)Y
```

Conclure sur une potentielle attaque par analyse de fréquences.

4.5.3 Déchiffrement

Pour le chiffre de Vigenère, nous avons obtenu l'image de chaque lettre du message clair par une addition entre les indexes dans l'alphabet : l'index pour la dite lettre et l'index de la lettre située à la même position dans le secret partagé. Pour effectuer le déchiffrement, il suffit donc d'effectuer la substitution inverse : une soustraction entre ces deux indexes.

Question 48 Copier et modifier le code de la fonction `tp_cipher` pour obtenir celui de la fonction `tp_decipher`, qui retourne le message clair à partir de l'alphabet, du secret partagé et du message chiffré.

```
def tp_decipher(
  theAlphabet,
  theSecret,
  theCipherText
):
  theClearText = ""

  for thePosition in range(0, len(theCipherText)):
    theImage      = theCipherText[thePosition]
    theImageIndex = theAlphabet.find(theImage)
    #
    theSecretLetter = theSecret[thePosition % len(theSecret)]
    theSecretIndex  = theAlphabet.find(theSecretLetter)
    #
    theIndex  = (theImageIndex - theSecretIndex) % len(theAlphabet)
    theLetter = theAlphabet[theIndex]
    #
```

```
    theClearText += theLetter

  return theClearText
```

Tester la fonction en l'appelant avec les données précédentes :

```
theCipherText   = tp_cipher(  theAlphabet, theSecret, theClearText)
theDecipherText = tp_decipher(theAlphabet, theSecret, theCipherText)
print("CIPHER TEXT   = " + theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

Le résultat doit ressembler à ceci :

```
CIPHER TEXT   = XEMILGEIVWZSSLGMYQNRXIYQHW RHSLMDCMNNRBQDCHWZUEQBYWZCRY
DECIPHER TEXT = LE CHIFFRE DE VIGENERE RESISTE A LANALYSE DE FREQUENCES
```

5 Attaques Monte-Carlo par chaînes de Markov

Les attaques **Monte-Carlo par chaînes de Markov** (ou attaques MCMC, pour *Markov Chains Monte-Carlo*) sont des attaques itératives sur les algorithmes de chiffrement par substitution, où un score est attribué à chaque itération en fonction d'une vraisemblance avec le langage du message clair.

Dans ces classes d'attaque, l'adversaire va choisir une proposition de clef de chiffrement et itérer sur des valeurs possibles en effectuant de très petites modifications (modifications par étapes / par marches). À chaque itération, la proposition de clef est utilisée pour déchiffrer le message. Sur cette proposition de déchiffrement, un score est calculé pour déterminer si le résultat ressemble à un message dans la langue du message clair. Le score obtenu est comparé avec le score de l'itération précédente :

— si le score est inférieur, la proposition de clef est rejetée ;
— si le score est supérieur, alors proposition de clef est sauvegardée en tant que clef probable : la prochaine itération effectuera une petite modification à partir de cette clef probable.

L'idée est qu'après un grand nombre d'itérations, le résultat du déchiffrement obtenu aura une forte vraisemblance avec le langage du message clair. Donc, plus on itère, plus le résultat ressemble à un message intelligible.

Ces attaques ont été introduites en 2001 par Persi Diaconis, de l'université de Stanford (en Californie). En 2003, Stephen Connor, de l'université de York (en Angleterre), a formalisé cette famille d'algorithmes et introduit le concept de marche aléatoire pour ces algorithmes.

Dans cette section, nous allons implémenter une attaque à l'aide de cette méthode. L'algorithme proposé est largement inspiré des travaux de Eitan Zimmerman et Yonatan Lourie, de l'université de Jérusalem, qui ont implémenté cette attaque sur un algorithme de chiffrement par substitution où le texte clair est écrit en Hébreu.

5.1 Les Chaînes de Markov

L'expression **chaîne de Markov** fait référence à un concept mathématique permettant de modéliser les transitions entre états indépendamment du passé. Sa définition mathématique est la suivante : *c'est un processus stochastique dont la prédiction future à partir du présent n'est pas rendu plus précise par le passé*.

Concrètement, cela signifie que les chaînes de Markov servent à modéliser des systèmes à états dont l'évolution résulte du fruit du hasard. Nous avons donc un concept de probabilités. Également, cela signifie que la prédiction de l'état suivant dans l'évolution est seulement définie depuis l'état dans lequel le système se trouve (elle ne tient pas compte des états passés).

Illustrons par un exemple. La figure 1 représente un modèle du comportement d'un chat à l'aide d'une chaîne de Markov. Dans ce modèle, un chat ne peut se trouver que dans trois états : en train de dormir, de manger ou de courir. Les changements d'états (les transitions) sont régies par une loi de probabilité.

Par exemple, sur ce modèle : lorsqu'un chat a fini de dormir, il y a 20% de probabilité qu'il se mette à courir, il y a 60% de probabilité qu'il se mette à manger et 20% de probabilité qu'il se remette à dormir. Comme l'évolution est seulement modélisée par l'état dans lequel le système se trouve, peu importe que le chat ait couru, mangé ou dormi avant : les probabilités restent les mêmes.

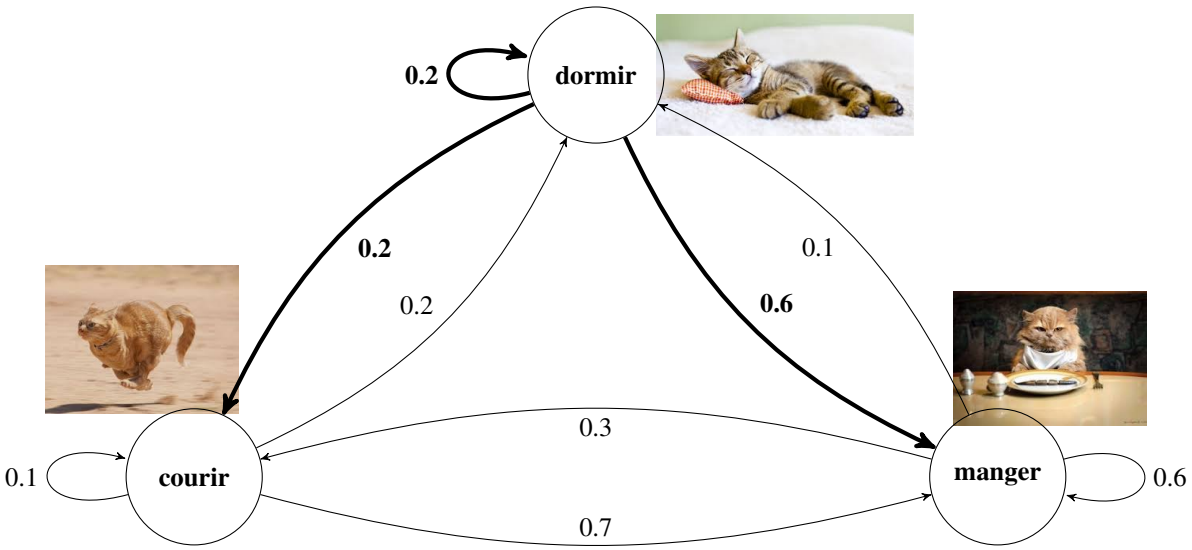

FIGURE 1 – Comportement d'un chat modélisé par une chaîne de Markov

Un tel modèle peut être construit en réalisant des statistiques : en observant une grande quantité de chats et en regardant quels changements d'états ceux-ci réalisent. Il suffit de compter le nombre de transitions durant cette phase d'observation. Une fois le modèle construit, nous pouvons observer le comportement d'un animal et le comparer à notre modèle : ceci permet de déterminer s'il est **vraisemblable** que cet animal soit un chat. Par exemple, si nous observons deux animaux, le premier se trouve dans ces états :

$$manger \to courir \to manger \to manger \to dormir$$

Le second se trouve dans ces états :

$$dormir \to dormir \to dormir \to dormir \to dormir$$

Nous pouvons attribuer un **score de vraisemblance** à ces deux animaux pour déterminer si ceux-ci sont susceptibles d'être des chats. Nous regardons donc toutes les transitions observées sur le comportement de ces deux animaux pour calculer le score : nous pouvons choisir de multiplier les probabilités, ou de les ajouter, ou de réaliser une fonction mathématique quelconque (selon l'application).

Dans cet exemple, le premier animal aura un score élevé et le second un score faible. Nous pouvons donc en conclure, seulement en regardant l'évolution des états que le premier animal a une plus forte vraisemblance d'être un chat.

Application à la cryptanalyse :

Dans les attaques MCMC, nous modélisons le comportement de la langue du message clair pour obtenir une chaîne de Markov qui représente les transitions des lettres. Nous pouvons choisir de regarder les lettres deux par deux (on parlera alors de **bigrammes**), ou trois par trois (on parlera alors de **tririammes**), etc. Ceci détermine le **n-gramme** qui encode notre modèle.

Prenons un exemple où nous observons les bigrammes : dans la langue française, le bigramme `ER` est très présent tandis que le bigramme `QV` est peu présent. En statistiques, si l'on prend un texte quelconque (mais long) écrit en français, le nombre de transitions de la lettre "E" vers la lettre "R" est élevé, tandis que le nombre de transitions de la lettre "Q" vers la lettre "V" est faible.

Si nous tentons de déchiffrer un message écrit en français et que celui-ci contient le bigramme `QV`, il est peu vraisemblable que la clef de chiffrement testée soit correcte.

Pour appuyer ce raisonnement, nous allons compter ces bigrammes dans le texte du roman de Jules Verne : Vingt mille Lieues Sous Les Mers. Nous allons réaliser des statistiques sur la présence des bigrammes dans le texte et construire un modèle probabiliste des changements de lettres dans la langue française : une chaîne de Markov.

Question 49 Le script `tp-crypto/05/mcmc.py` contient un squelette prêt pour implémenter l'attaque MCMC. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/05
debian@myhostname:~/tp-crypto/05$ ./mcmc.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre algorithme.

Question 50 Le fichier `~/tp-crypto/jverne.txt` contient le texte du roman de Jules Verne : Vingt mille Lieues Sous Les Mers. La fonction `tp.formatText()` permet de modifier une chaîne de caractères pour n'utiliser que l'alphabet précédent (contenant les 26 majuscules de l'alphabet latin et l'espace).
Modifier le script `Python` pour lire le contenu fichier et formater le texte pour utiliser cet alphabet.
Attention à bien se placer dans le dossier contenant le script avant de l'exécuter (utilisation d'un chemin relatif).

```
theFile   = "../jverne.txt"
theBuffer = open(theFile, "r")
theText   = theBuffer.read()
theBuffer.close()
theText   = tp.formatText(theText)
```

Question 51 Dans le texte formaté, compter le nombre d'apparitions du bigramme `ER` et du bigramme `QV`.

```
print("compteur de \"ER\": " + str(theText.count("ER")))
print("compteur de \"QV\": " + str(theText.count("QV")))
```

Question 52 Pour construire un modèle de la langue française à partir de bigrammes, nous devons mesurer la fréquence d'apparition de chaque bigramme.
À partir du texte formaté, construire une matrice de 27×27 éléments (la taille de l'alphabet au carré) donnant la fréquence d'apparition de chaque bigramme (nombre d'apparitions du bigramme divisé par le nombre total de bigrammes dans le texte).
Note : nous allons forcer le minimum de la fréquence à au moins une apparition dans le texte pour éviter d'avoir une valeur nulle (ce qui peut causer des problèmes avec certaines fonctions mathématiques ; par exemple, la division).

```
theMin    = 1.0 / (len(theText)-1)
theMatrix = [[theMin] * len(theAlphabet)] * len(theAlphabet)

for i in range(0, len(theText)-1):
  theChar     = theText[i]
  theNextChar = theText[i+1]
  # find the location of the bigram in the matrix:
  theCharIndex     = theAlphabet.find(theChar)
  theNextCharIndex = theAlphabet.find(theNextChar)
  # increment the probability by 1/(total number of bigrams):
  theRow = theMatrix[theCharIndex].copy()
  theRow[theNextCharIndex] += 1.0 / (len(theText)-1)
  theMatrix[theCharIndex] = theRow.copy()
```

Question 53 Vérifier que la fréquence d'apparition du bigramme `ER` est bien égale à la valeur observée précédemment ($+1$) divisée par le nombre de bigrammes dans le texte.

```
print(theMatrix[theAlphabet.find("E")][theAlphabet.find("R")])
print((theText.count("ER") + 1) / (len(theText)-1))
```

La valeur peut être différente à quelques chiffres après la virgule, ceci est dû à la précision de la machine lorsque l'on travaille avec des nombres réels.

Question 54 Tracer la matrice sous forme d'une grille où la teinte d'une case reflète la fréquence d'apparition du bigramme.
L'ordonnée représente le premier caractère du bigramme et l'abscisse, le second caractère.

```
x = range(0, len(theAlphabet))
y = range(0, len(theAlphabet))
z = theMatrix
fig, ax = matplotlib.pyplot.subplots()
ax.set(xticks=range(0, len(theAlphabet)), xticklabels=list(theAlphabet))
ax.set(yticks=range(0, len(theAlphabet)), yticklabels=list(theAlphabet))
ax.set_title("Bigram frequency in the text")
theMesh = ax.pcolormesh(x, y, z, shading="auto")
fig.colorbar(theMesh)
matplotlib.pyplot.show()
```

Commenter la dernière ligne (`matplotlib.pyplot.show()`) permet d'empêcher le graphique de s'afficher.

À partir de cette matrice statistique (de toutes les fréquences d'apparition de chaque bigramme), nous pouvons construire un modèle probabiliste de la langue française sous forme d'une chaîne de Markov.
Concrètement, ce modèle possède 27 états (un par lettre de l'alphabet pour la première lettre du bigramme) où chaque état possède 27 transitions (une par lettre de l'alphabet pour la seconde lettre du bigramme).
La probabilité à attribuer à chaque transition n'est pas égale à la fréquence d'apparition, mais nous n'avons pas besoin de la calculer pour réaliser notre attaque [8].
Ce qui est important est de pouvoir attribuer un **score de vraisemblance** à tout message pour voir si celui-ci représente du texte en français. À chaque itération, nous allons observer si nous nous approchons d'un message clair correct ou si nous nous en éloignons.

5.2 La méthode Monte-Carlo

Le terme méthode de Monte-Carlo, ou **méthode Monte-Carlo**, désigne une famille de méthodes algorithmiques visant à calculer une valeur numérique approchée en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes. Le nom de ces méthodes fait allusion aux jeux de hasard pratiqués au casino de Monte-Carlo.

Source : `https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Monte-Carlo`

Dans ce TP, nous allons utiliser **l'algorithme de Metropolis-Hastings** pour notre attaque.

En statistique, **l'algorithme de Metropolis-Hastings** est une méthode MCMC dont le but est d'obtenir un échantillonnage aléatoire d'une distribution de probabilité quand l'échantillonnage direct en est difficile.

L'algorithme de Metropolis-Hastings génère des valeurs d'échantillon x de manière que plus on produit de valeurs, plus la distribution de ces valeurs se rapproche de la distribution recherchée. Ces valeurs sont produites de manière itérative, la probabilité de l'échantillon à l'étape suivante ne dépend que de l'échantillon à l'étape considérée, la suite d'échantillons est donc une chaîne de Markov.

Plus précisément, à chaque itération, l'algorithme choisit un candidat pour le prochain échantillon x_{n+1} qui n'est basé que sur l'échantillon courant x_n. Ensuite, le candidat x_{n+1} est soit accepté avec une certaine probabilité (et dans ce cas il sera utilisé à l'étape suivante), soit rejeté avec la probabilité complémentaire (et dans ce cas l'échantillon actuel x_n sera réutilisé à l'étape suivante).

La probabilité d'acceptation est déterminée en comparant les valeurs de la fonction $f(x_n)$, calculant le score pour l'échantillon actuel, à la valeur de $f(x_{n+1})$ pour l'échantillon candidat. Après un "grand nombre" d'itérations, la valeur des échantillons "perdent la mémoire" de l'état initial et suivent la distribution recherchée.

Source : `https://fr.wikipedia.org/wiki/Algorithme_de_Metropolis-Hastings`

5.2.1 Score de vraisemblance

Afin de déterminer la probabilité d'acceptation d'un échantillon, il est donc nécessaire d'implémenter une fonction $f(x)$ qui calcule un **score de vraisemblance**. Ce score sera calculé à partir de l'échantillon x (notre texte déchiffré avec une clef candidate) et notre matrice donnant la fréquence d'apparition des bigrammes dans la langue française.

Concrètement, nous allons observer les bigrammes présents dans l'échantillon x et incrémenter le score proportionnellement à la fréquence d'apparition de chaque bigramme. Pour deux textes de même longueur (contenant le même nombre de bigrammes), nous pourrons donc comparer les scores et en déduire une probabilité d'acceptation.

8. Pour les curieux, il est possible de calculer cette probabilité en additionnant toutes les fréquences d'apparition de chaque bigramme où la première lettre est identique, puis diviser chaque fréquence d'apparition par ce résultat.

Question 55 Nous allons, dans un premier temps, tenter de déchiffrer notre texte chiffré avec une clef de chiffrement fausse. La première étape pour calculer son score est de lister tous les bigrammes contenus dans le résultat et trouver leur fréquence d'apparition dans la langue française depuis notre matrice.
Appeler la fonction pour déchiffrer le message, puis écrire une boucle et afficher la fréquence d'apparition de chaque bigramme présent.

```
theText = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)

for i in range(0, len(theText)-1):
  theChar     = theText[i]
  theNextChar = theText[i+1]
  # find the location of the bigram in the matrix:
  theCharIndex     = theAlphabet.find(theChar)
  theNextCharIndex = theAlphabet.find(theNextChar)
  # print the frequency
  theFreq = theMatrix[theCharIndex][theNextCharIndex]
  print("bigramme \"" + theChar + theNextChar + "\": " + str(theFreq))
```

Ensuite, nous devons définir comment le score sera incrémenté en fonction de la fréquence d'apparition. Afin d'obtenir de meilleurs résultats pour la convergence de l'algorithme de Metropolis-Hastings, il est préférable de ne pas traiter les écarts de fréquence d'apparition de manière linéaire.

En effet, un bigramme qui apparait avec 14% de probabilité doit augmenter le score de manière presque identique à un bigramme qui apparait avec 15% de probabilité. Par contre, un bigramme qui apparait avec 0.01% de probabilité doit augmenter le score de manière beaucoup plus forte qu'un bigramme qui apparait avec 0.001% de probabilité.

Pour cette raison, nous allons utiliser la fonction logarithme sur la fréquence d'apparition, normalisée par le nombre de bigrammes dans le message.
Note : en conséquence, les scores obtenus seront négatifs. Dans cette section, l'ordre de grandeur du score sera de -3000 pour un message peu vraisemblable d'être du français, et -500 pour un message écrit en français.

Question 56 Ré-écrire la boucle qui permet d'obtenir la fréquence d'apparition de chaque bigramme présent, incrémenter un score par le logarithme de cette fréquence divisée par le nombre de bigrammes. En déduire le score pour le message.

```
theText  = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)
theScore = 0.0
for i in range(0, len(theText)-1):
  theChar     = theText[i]
  theNextChar = theText[i+1]
  # find the location of the bigram in the matrix:
  theCharIndex     = theAlphabet.find(theChar)
  theNextCharIndex = theAlphabet.find(theNextChar)
  # increment the score:
  theFreq   = theMatrix[theCharIndex][theNextCharIndex]
  theScore += math.log(theFreq / (len(theText)-1))
#
print(theScore)
```

Question 57 Nous avons donc terminé notre implémentation. À partir du code précédent, compléter la fonction `tp_f_score()` qui retourne le score d'un message à partir de la matrice contenant les fréquences d'apparition des bigrammes et l'alphabet.

```
def tp_f_score(
  theText,
  theMatrix,
  theAlphabet
):
  theScore = 0.0

  for i in range(0, len(theText)-1):
    theChar     = theText[i]
    theNextChar = theText[i+1]
    # find the location of the bigram in the matrix:
```

```
    theCharIndex     = theAlphabet.find(theChar)
    theNextCharIndex = theAlphabet.find(theNextChar)
    # increase the score:
    theIncr   = theMatrix[theCharIndex][theNextCharIndex] / (len(theText)-1)
    theScore += math.log(theIncr)

  return theScore
```

Question 58 Tester cette fonction sur le message que nous tentons de déchiffrer et vérifier que nous obtenons le même score que précédemment.

```
print(tp_f_score(theText, theMatrix, theAlphabet))
```

Question 59 Nous allons maintenant calculer les scores de deux messages proches que nous considérons comme candidats. Nous allons observer la différence afin de déterminer quel candidat sera accepté et quel candidat sera rejeté. Écrire un message en français et appeler la fonction `tp.formatText()` pour s'assurer que celui-ci n'utilise que des caractères de l'alphabet.

```
theText = "VOICI UN MESSAGE ECRIT DANS LA LANGUE DE MOLIERE"
theText = tp.formatText(theText)
```

Question 60 Créer un second message à partir d'une copie du premier où l'on a échangé les lettres "E" et "L". Calculer ensuite les scores des deux messages.

```
theText2 = theText.replace("E", "_")
theText2 = theText2.replace("L", "E")
theText2 = theText2.replace("_", "L")
#
print(tp_f_score(theText,  theMatrix, theAlphabet))
print(tp_f_score(theText2, theMatrix, theAlphabet))
```

L'algorithme de Metropolis-Hastings utilise des calculs de scores pour comparer deux échantillons. C'est donc la différence entre les deux scores qui est utilisée pour décider si un échantillon candidat est accepté ou rejeté. Cette différence constitue donc notre **distingueur** (ou **oracle**) qui nous indique si nous nous rapprochons ou nous éloignons d'un texte français vraisemblable.

5.2.2 Consultons l'oracle : définition des itérations

Pour décrypter le message, notre algorithme doit réaliser de petits changements sur la clef de manière itérative. Ces modifications nous permettent d'obtenir une clef candidate.

Le message candidat, déchiffré avec la clef candidate, a donc une petite différence avec le message déchiffré avec la clef précédente. La convergence de l'algorithme (le message déchiffré devient peu à peu du français vraisemblable) n'est possible que si la différence entre les deux messages est elle aussi petite.

Pour notre attaque sur l'algorithme de chiffrement par substitution monoalphabétique, nous allons donc, à chaque itération, inverser deux symboles dans les correspondances avec l'alphabet. Ensuite, nous allons tenter de déchiffrer le message, calculer son score et le comparer avec le score du message précédent.

Question 61 Comme nous allons travailler avec un processus pseudo-aléatoire, forcer la graine (*seed*) pour obtenir un résultat déterministe.

```
random.seed(2)
```

Question 62 Définir un nombre d'itérations à 10000 et créer une boucle dans laquelle le message est déchiffré avec la clef et un score est calculé. Appeler la fonction `tp_f_score()` pour calculer le score.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  theText  = tp_decipher(theAlphabet, theCipherAlphabet, theCipherText)
  theScore = tp_f_score(theText, theMatrix, theAlphabet)
```

Question 63 À chaque itération, nous allons sauvegarder le score obtenu dans une liste pour pouvoir le tracer par la suite et observer son évolution.
Compléter la boucle de manière à ajouter le score à une liste à chaque itération.

```
theScoreList = list()
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  theScoreList.append(theScore)
```

Question 64 Ensuite, nous créons une clef candidate avec un changement mineur et aléatoire sur la clef sauvegardée. Compléter la boucle avec la création d'une clef candidate où deux caractères sont inversés de manière aléatoire dans la clef. La fonction `tp.randomSwap()` réalise une inversion aléatoire sur deux caractères d'une chaîne de caractères.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  theCandidateAlphabet = tp.randomSwap(theCipherAlphabet)
```

Question 65 Ensuite, nous tentons de déchiffrer le message avec la clef candidate et calculons son score.
Compléter la boucle avec une tentative de déchiffrement du message avec la clef candidate. Calculer le score du résultat.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  theCandidateText  = tp_decipher(theAlphabet, theCandidateAlphabet, theCipherText)
  theCandidateScore = tp_f_score(theCandidateText, theMatrix, theAlphabet)
```

Question 66 Nous devons désormais accepter le petit changement sur la clef si le score obtenu est supérieur au score précédent.
Toujours dans la boucle, ajouter une condition pour écraser la clef sauvegardée par la clef candidate si le score obtenu avec celle-ci est supérieur.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  if ( theCandidateScore > theScore ):
    theCipherAlphabet = theCandidateAlphabet
```

Question 67 Après toutes ces itérations, afficher le dernier message déchiffré et tracer l'évolution du score.

```
print(theText)

x = range(0, theMaxIter)
y = theScoreList
fig, ax = matplotlib.pyplot.subplots()
ax.plot(x, y)
ax.set_title("Score for each iteration")
matplotlib.pyplot.show()
```

Le message obtenu est-il bien écrit en français ?
Que remarquez vous sur l'évolution du score ?

5.2.3 Algorithme de Metropolis-Hastings

Avec notre algorithme précédent, il est possible que le texte final obtenu se rapproche d'un message intelligible mais le résultat n'est pas convainquant. De plus, nous observons que le score obtenu augmente à chaque itération mais atteint rapidement un maximum qu'il ne peut pas dépasser.

Avec cette stratégie, plus de 7000 itérations sont inutiles car nous ne parvenons pas à obtenir un meilleur score, malgré la relative faiblesse de celui-ci. C'est à cet instant que l'algorithme de Metropolis-Hastings intervient.

Imaginons que nous marchons pour atteindre le sommet d'une montagne, sans savoir dans quelle direction se diriger. La stratégie revient à observer le sol et toujours se diriger là où celui-ci monte. C'est exactement ce que nous avons implémenté pour atteindre le score maximum dans notre algorithme précédent.

L'ennui est qu'avec cette stratégie, nous pouvons atteindre un **maximum local**, où quelque-soit le petit changement que nous effectuons, nous redescendons. Pourtant, ce maximum local peut être inférieur au **maximum global**. La figure 2 illustre cette problématique, où pour atteindre le sommet de la montagne, il est nécessaire de redescendre.

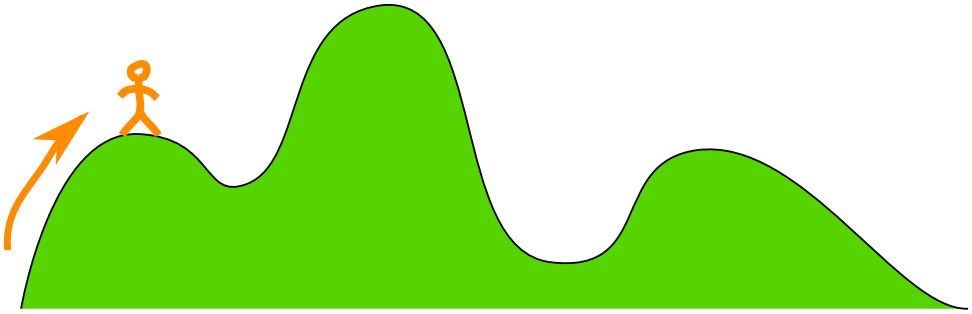

FIGURE 2 – Atteinte d'un maximum local

Dans le cas de notre algorithme précédent, nous avons atteint un maximum local où effectuer un petit changement sur la clef ne permet pas d'obtenir un meilleur score. Dans ce cas, inverser deux symboles fournit toujours un score plus faible, mais inverser quatre ou six symboles permettrait potentiellement d'obtenir un meilleur score.

L'algorithme de Metropolis-Hastings répond à cette problématique en suggérant d'accepter certains scores plus faibles avec une probabilité donnée. L'objectif est d'accepter toutes les clefs permettant d'atteindre un score plus élevé mais de ne pas rejeter certaines clefs donnant un score plus faible.

La solution proposée est d'appliquer une fonction mathématique exponentielle sur notre oracle (la différence des deux scores) et tirer un nombre aléatoire compris entre zéro et un. Le candidat est accepté si le résultat de l'exponentielle est supérieur au nombre aléatoire tiré :

— si le score du candidat est plus élevé que le précédent, alors l'exponentielle de la différence est forcément supérieure à un : le candidat est accepté quelque-soit le résultat du tirage aléatoire.
— si le score du candidat est plus faible que le précédent, alors nous avons une probabilité d'accepter néanmoins le candidat, qui est plus élevée si la différence est faible.

Question 68 Copier le script `tp-crypto/05/mcmc.py` vers `tp-crypto/05/mcmc2.py` pour y implémenter l'algorithme de Metropolis-Hastings.

```
debian@myhostname:~$ cp ~/tp-crypto/05/mcmc.py ~/tp-crypto/05/mcmc2.py
debian@myhostname:~$ cd ~/tp-crypto/05
```

Modifier cette copie pour supprimer le précédent algorithme, garder la construction de la matrice contenant les fréquences d'apparition des bigrammes.

Question 69 Nous avons précédemment travaillé avec deux clefs pour déchiffrer le message : la clef actuelle dans laquelle nous sauvegardons le meilleur des deux scores lors d'une itération, et la clef candidate où nous échangeons deux symboles de manière aléatoire.

Pour implémenter l'algorithme de Metropolis-Hastings, nous avons besoin d'une troisième clef : celle qui nous permet d'obtenir le meilleur score obtenu (là où la clef actuelle aura potentiellement un score plus faible). Nous baptisons cette clef la meilleure clef.
Modifier le script pour déclarer les trois clefs et la liste des scores à tracer.

```
theCurrentKey   = theCipherAlphabet
theCandidateKey = theCipherAlphabet
theBestKey      = theCipherAlphabet
theScoreList = list()
```

Question 70 Définir un nombre d'itérations à 10000 et créer une boucle dans laquelle le message est déchiffré avec la clef actuelle et un score est calculé.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  theText  = tp_decipher(theAlphabet, theCurrentKey, theCipherText)
  theScore = tp_f_score(theText, theMatrix, theAlphabet)
```

Question 71 Compléter la boucle de manière à ajouter le score à une liste à chaque itération.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  theScoreList.append(theScore)
```

Question 72 Compléter la boucle avec la création d'une clef candidate où deux caractères sont inversés de manière aléatoire dans la clef. Tenter de déchiffrer le message avec la clef candidate.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  theCandidateKey   = tp.randomSwap(theCurrentKey)
  theCandidateText  = tp_decipher(theAlphabet, theCandidateKey, theCipherText)
  theCandidateScore = tp_f_score(theCandidateText, theMatrix, theAlphabet)
```

Question 73 Comme défini par l'algorithme de Metropolis-Hastings, générer un nombre aléatoire compris entre 0 et 1 ; comparer le résultat à l'exponentielle de la différence des deux scores. Accepter la clef candidate selon le résultat de la comparaison.
La fonction `random.random()` permet de générer un nombre aléatoire compris entre 0 et 1. La fonction mathématique exponentielle peut être calculée avec la fonction `math.exp()`.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  u = random.random()
  if ( math.exp(theCandidateScore - theScore) > u ):
    theCurrentKey = theCandidateKey
```

Question 74 Toujours dans la boucle, sauvegarder la clef candidate si celle-ci permet d'obtenir le score maximum.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  if ( theCandidateScore > max(theScoreList) ):
    theBestKey = theCandidateKey
```

Question 75 Au final, déchiffrer le texte avec la meilleure clef obtenue puis l'afficher. Tracer également l'évolution du score au fil des itérations.

```
theText = tp_decipher(theAlphabet, theBestKey, theCipherText)
print(theText)

x = range(0, theMaxIter)
y = theScoreList
fig, ax = matplotlib.pyplot.subplots()
ax.plot(x, y)
ax.set_title("Score for each iteration")
matplotlib.pyplot.show()
```

Conformément à l'aspect aléatoire de la construction de la clef candidate, il est possible que certaines exécutions ne fournissent pas un résultat satisfaisant (sauf si la *seed* a été forcée). Néanmoins, l'algorithme de Metropolis-Hastings permet de produire de meilleurs résultats que notre algorithme précédent et de déchiffrer tout le texte.
Exécuter le script plusieurs fois, si besoin, pour obtenir un résultat correct. Que remarquez vous sur l'évolution du score ?

5.3 Attaque MCMC sur le Chiffre de Vigenère

Nous avons montré que l'attaque MCMC permet d'obtenir de bons résultats sur les algorithmes de chiffrement par substitution monoalphabétique. Dans cette section, nous allons évaluer ses capacités sur un algorithme de chiffrement par substitution polyalphabétique (ou Chiffre de Vigenère).

Question 76 Le script `tp-crypto/06/mcmc_vigenere.py` contient un squelette prêt pour implémenter l'attaque MCMC sur un algorithme de chiffrement polyalphabétique. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/06
debian@myhostname:~/tp-crypto/06$ ./mcmc_vigenere.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre attaque.

Comme dans la section précédente, ce script implémente les algorithmes de chiffrement et déchiffrement par substitution polyalphabétique (ou Chiffre de Vigenère). Également, ce script lit le roman de Jules Verne : Vingt mille Lieues Sous Les Mers. Comme précédemment, à partir de ce texte, il construit une matrice statistique contenant les fréquences d'apparition de chaque bigramme.

Un squelette de l'algorithme de Metropolis-Hastings est également présent, mais la définition des marches (petites modifications réalisées à chaque étape) n'est pas implémentée. De plus, le point de départ de l'algorithme n'est pas défini.

Question 77 Modifier le script pour définir les marches de l'algorithme de Metropolis-Hastings. À chaque itération, replacer aléatoirement une lettre de la clef par une autre lettre, prise de manière aléatoire dans l'alphabet.

```
theMaxIter = 10000
for theIter in range(0, theMaxIter):
  # [...]

  # TODO: define the steps here
  theIndex = random.randrange(0, len(theCandidateKey))
  theChar  = random.sample(list(theAlphabet), k=1)
  #
  theCandidateKeyList = list(theCurrentKey)
  theCandidateKeyList[theIndex] = theChar[0]
  theCandidateKey = "".join(theCandidateKeyList)
```

Tester et observer le résultat.

Question 78 Le point de départ de l'algorithme, c'est-à-dire la clef candidate utilisée lors de la première itération, a un impact plus important : il définit la taille de la clef candidate à chaque itération.
Modifier la taille de la clef candidate et essayer à nouveau.

```
theCurrentKey   = "AZERTY"
theCandidateKey = theCurrentKey
theBestKey      = theCurrentKey
```

Tester et observer le résultat. Conclure sur la taille de clef choisie.

Question 79 Admettons que nous connaissons la taille de la clef utilisée lors du chiffrement.
Replacer le point de départ de l'algorithme avec une clef de 26 caractères.

```
theCurrentKey   = "AZERTYUIOPQSDFGHJKLMWXCVBN"
theCandidateKey = theCurrentKey
theBestKey      = theCurrentKey
```

Tester et observer le résultat. Que dire de la vulnérabilité du chiffre de Vigenère aux attaques MCMC ?

Question 80 Le script `tp-crypto/06/final.py` propose de chiffrer un message par le chiffre de Vigenère, où le secret possède la même taille que le message : 237 caractères. La même attaque que précédemment est implémentée où le nombre d'itérations est augmenté à 30000.
Exécuter ce script et observer le résultat.

```
debian@myhostname:~$ cd ~/tp-crypto/06
debian@myhostname:~/tp-crypto/06$ ./final.py
```

Conclure sur la vulnérabilité du chiffre de Vigenère aux attaques MCMC.

6 Cryptanalyse des systèmes de chiffrement

Dans les sections précédentes, nous avons vu des algorithmes de chiffrement par substitution (monoalphabétique et polyalphabétique). Un ensemble des possibles conséquent pour la clef de chiffrement implique que nous ne pouvons pas décrypter les messages avec des attaques par force brute. Néanmoins, l'analyse de fréquences et les attaques Monte-Carlo par chaînes de Markov permettent de venir à bout de ces algorithmes. La seule restriction est, pour le chiffrement par substitution polyalphabétique, de connaître la taille de la clef et que celle-ci soit de longueur réduite.

Durant ces précédentes attaques, nous avons simplement considéré le texte chiffré. Celles-ci font donc partie de la catégorie des **attaques sur texte chiffré seul**. De ce fait, les vulnérabilités que nous avons établies sont intrinsèques à l'algorithme de chiffrement et au choix de la clef. Dans cette section, nous allons regarder l'impact de la mise en œuvre : les choix d'implémentation des précédents algorithmes, des choix de clefs, des choix de messages clair, etc. Ceci nous permettra de définir des attaques faisant partie d'autres catégories.

6.1 Test de Kasiski

Lors de la cryptanalyse du chiffre de Vigenère, si l'on connait le nombre de symboles que comporte la clef, il devient possible de procéder à une attaque par analyse de fréquences. Cette analyse de fréquences doit être réalisée sur chacun des sous-textes déterminés en sélectionnant des lettres du message clair à intervalle régulier, fixé par la longueur de la clef (autant de sous-textes que la longueur de la clef). Il est également possible de réaliser une attaque Monte-Carlo par chaînes de Markov, ici aussi en se basant sur la longueur de la clef.

On pense que Charles Babbage effectua la première véritable cryptanalyse du chiffre de Vigenère vers 1854. En parallèle, un officier prussien à la retraite, Friedrich Wilhelm Kasiski parvint au même résultat sans avoir eu vent des travaux de Babbage puisque ce dernier ne les avait pas publiés. Kasiski rédigea *Die Geheimschriften und die Dechiffrierkunst* en 1863 où il présentait le test qui allait porter son nom : **le test de Kasiski** qui permet d'estimer la taille de la clef.
Cette attaque exploite une faiblesse bien connue des utilisateurs : **l'utilisation d'un secret de taille réduite**.

Question 81 Le script `tp-crypto/07/kasiski.py` contient un squelette prêt pour implémenter le test de Kasiski.
Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/07
debian@myhostname:~/tp-crypto/07$ ./kasiski.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre attaque.

Le test de Kasiski consiste à analyser le texte chiffré est observer la répétition de schémas. Par exemple, si un texte contient beaucoup le bigramme `LE` et que la taille du secret utilisé pour le chiffre de Vigenère est faible, alors il est probable que ce même bigramme soit substitué plusieurs fois de la même manière.

Le test de Kasiski prévoit d'observer la répétition de tous les n-grammes, quelque-soit leur taille, pour permettre de déterminer la taille du secret plus simplement. Dans cette section, nous allons observer la répétition des trigrammes.

Question 82 Pour un message chiffré de longueur 3, combien y a-t-il de trigrammes ?
Idem pour un message chiffré de longueur 4 ? De longueur 5 ?
En déduire le nombre de trigrammes pour un message chiffré de longueur n.

Question 83 À partir du message chiffré fourni dans le script, construire une liste de tous les trigrammes qui le composent.

```
theTrigramList = list()
for i in range(0, len(theCipherText)-2):
  theTrigramList.append(theCipherText[i] + theCipherText[i+1] + theCipherText[i+2])
```

Question 84 Repérer les trigrammes qui apparaissent plus d'une fois dans la liste. Construire une liste des trigrammes répétés.

```
theDuplicateList = list()
for theTrigram in theTrigramList:
  if ( theTrigramList.count(theTrigram) > 1 ) and ( theTrigram not in theDuplicateList ):
    theDuplicateList.append(theTrigram)
```

Question 85 Afficher les positions de ces trigrammes dans le message chiffré.

```
# print the positions of the trigrams that appear more than once:
for theDuplicate in theDuplicateList:
  thePrint   = theCipherText.replace(theDuplicate, "___")
  thePattern = re.compile("[^_]")
  thePrint   = thePattern.sub(".", thePrint)
  print("               " + thePrint.replace("___", theDuplicate))
```

Question 86 Repérer les positions de ces trigrammes et en déduire les tailles de clefs possibles.
La fonction `tp.find_all()` retourne tous les indexes d'une sous-chaîne présents dans une chaîne de caractère donnée.

```
print("Tailles de clef possibles:")
theKeyLengthList = list()
for theDuplicate in theDuplicateList:
  theIndexes = tp.find_all(theCipherText, theDuplicate)
  if ( len(theIndexes) > 1 ):
    if ( (theIndexes[1] - theIndexes[0]) not in theKeyLengthList ):
      theKeyLengthList.append(theIndexes[1] - theIndexes[0])

for theKeyLength in theKeyLengthList:
  print(theKeyLength)
```

Question 87 À partir de ces tailles de clef possibles, en déduire une taille de clef probable.

Question 88 Le test de Kasiski permet d'obtenir la liste de toutes les tailles de clef possibles. Nous pouvons réduire cette liste en calculant le PGCD (Plus Grand Commun Diviseur) de toutes les tailles obtenues.
À partir d'une liste de nombres entiers, la fonction `tp.gcd()` calcule le PGCD de tous les n-uplets formés à partir de ces nombres.
Calculer la taille de clef probable.

```
print("Taille de clef probable: " + str(tp.gcd(theKeyLengthList)))
```

Question 89 La fonction `run()` exécute l'attaque MCMC sur un texte chiffré avec le chiffre de Vigenère pour une taille de clef donnée. Exécuter la fonction `run()` avec la taille de clef probable pour vérifier que le test de Kasiski a retourné le bon résultat.

```
run(theAlphabet, theCipherText, tp.gcd(theKeyLengthList))
```

Se protéger

La contremesure qui vise à se protéger du test de Kasiski est évidente : il suffit d'employer un secret dont la longueur est conséquente. L'idéal est bien évidemment de posséder une clef de chiffrement dont la longueur est supérieure ou égale à celle du message. De ce fait, aucune répétition dans le texte clair ne fait apparaître une répétition dans le texte chiffré.

Également, le contenu du secret doit être varié : utiliser deux fois le même mot mis bout à bout équivaut à utiliser un secret deux fois plus court. En effet, le chiffre de Vigenère consiste à répéter le secret en boucle jusqu'à atteindre la longueur du message clair avant d'appliquer les substitutions.

6.2 Attaques à clair connu

Dans le cas où le secret utilisé pour le chiffre de Vigenère est de longueur réduite, il devient possible de casser le chiffrement par de multiples analyses de fréquences ou une attaque MCMC. Si le secret utilisé est trop grand, il existe des moyens détournés d'obtenir une réduction de l'espace des possibles.

Une **attaque à texte clair connu** (en anglais *known-plaintext attack* ou *KPA*) est un modèle d'attaque en cryptanalyse où l'attaquant possède à la fois le texte chiffré (*cipher*) et un texte clair (*plaintext* en anglais), c'est-à-dire tout ou une partie du message déchiffré : le clair connu. Ces éléments peuvent être utilisés afin de révéler d'autres informations secrètes comme la clef de chiffrement ou la table de correspondance utilisée.

Les chiffrements dits "classiques" (en opposition aux chiffrements dits "forts" qui s'appuient sur l'informatique) sont généralement vulnérables à une attaque à texte clair connu. La clef d'un chiffrement par décalage par exemple, peut être révélée en utilisant une seule lettre d'un texte clair et du texte chiffré correspondant.

Source : `https://fr.wikipedia.org/wiki/Attaque_%C3%A0_texte_clair_connu`

Les algorithmes de chiffrement par substitution monoalphabétique sont vulnérables aux attaques à clair connu car celles-ci fournissent directement les correspondances utilisées lors de la substitution. Dans le cas du chiffrement par substitution polyalphabétique (chiffre de Vigenère), un sous-ensemble du secret peut être obtenu par cryptanalyse de la différence entre le message clair et le message chiffré.

Dans cette section, nous nous plaçons dans le cas où le message est chiffré à l'aide du chiffre de Vigenère et où une partie du texte clair est connue de l'adversaire. L'objectif est d'extraire un secret placé dans un message chiffré par la victime. Nous savons que cette victime termine tous ses messages par le mot `CORDIALEMENT`.
Cette attaque exploite une faiblesse des utilisateurs : **l'utilisation récurrente de mots spécifiques**.

Question 90 Le script `tp-crypto/08/kpa.py` contient un squelette prêt pour implémenter une attaque à clair connu. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/08
debian@myhostname:~/tp-crypto/08$ ./kpa.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre attaque.

La fonction `tp.verifySecret()` permet de vérifier si le secret extrait du message de la victime est correct. Si nous tentons de l'exécuter avec la chaîne de caractères `"TEST"`, alors celle-ci affiche le message `This is not our secret`. Cette chaîne de caractères n'est donc pas le secret.

Question 91 Nous connaissons l'alphabet utilisé par la victime. De plus, un test de Kasiski sur le message chiffré a révélé que la taille de clef était de 64 caractères.
Modifier le script pour tenir compte de ces informations.

```
theAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
theKeySize  = 64
```

Question 92 La fonction `run()` exécute l'attaque MCMC sur un texte chiffré avec le chiffre de Vigenère pour une taille de clef donnée. Exécuter la fonction `run()` avec la taille de clef donnée pour tenter de décrypter le message.
Que remarquez vous sur le résultat ? Exécuter le script plusieurs fois si besoin.

```
run(theAlphabet, theCipherText, theKeySize)
print()
```

L'attaque MCMC permet de déchiffrer une partie du message mais certains caractères n'ont pas de sens. C'est probablement à cet emplacement que se situe le secret que nous souhaitons extraire.

L'attaque MCMC ne permet pas de déchiffrer le secret car celui-ci est une suite de caractères aléatoires qui ne forme pas un mot de la langue française. Un score de vraisemblance avec la langue française n'est donc pas un distingueur adapté.

Question 93 Le reste du message, écrit en français, peut être décrypté par l'attaque MCMC. Il constitue donc notre clair connu.
Vérifier que ce texte se termine bien par le mot `CORDIALEMENT`.

Question 94 Sauvegarder le texte clair dans une chaîne de caractères, remplacer les caractères inconnus par des points "`.`" (il n'est pas nécessaire de sauvegarder tout le texte : une longueur égale à 1 ou 2 fois la clef de chiffrement suffit).

```
theKnown = """
BONJOUR TOUT LE MONDE VOICI UN SECRET A NE PAS PARTAGER ........ JE SUIS UN CRYPTOGRAPHE JE CONNAIS
LA TAILLE DES CLEFS ........
""".strip()
```

Question 95 À partir du message chiffré et de ce clair connu, nous allons maintenant exécuter notre attaque pour extraire des morceaux de clef.
Écrire une fonction qui exécute l'opération inverse du chiffrement depuis le clair connu. Ceci doit être réalisé sur tous les caractères qui ne sont pas des points. Cette fonction doit retourner la clef déduite, où tout symbole inconnu de la clef est un point.

```
def kpa(
  theAlphabet,
  theCipherText,
  theKnownText,
  theKeySize
):
  theKpaKey = "." * theKeySize
  for i in range(0, min(len(theKnownText), len(theCipherText))):
    if ( theKnownText[i] in theAlphabet ):
      theKeyList  = list(theKpaKey)
      theKeyIndex = i % theKeySize
      theKeyList[theKeyIndex] = tp_decipher(theAlphabet, theKnownText[i], theCipherText[i])
      theKpaKey = "".join(theKeyList)
  #
  return theKpaKey
```

Question 96 Afficher la clef déduite et tenter de déchiffrer le message avec celle-ci pour vérifier que l'attaque a fonctionné.

```
theKpaKey = kpa(theAlphabet, theCipherText, theKnown, theKeySize)
print("KPA KEY       = " + theKpaKey)
print("KPA MESSAGE   = " + tp_decipher(theAlphabet, theKpaKey, theCipherText))
print()
```

Si les morceaux de la clef déduite semblent corrects, nous ne pouvons pas conclure sur le secret car une partie de la clef est manquante.

Nous avons intercepté un second message chiffré de la victime. Celui-ci contient le texte suivant :

```
LWFGQXVJVNTYOJGQEB KJZVWHOPCAFRTHT DKXKVJLTRBNLXEYNCVZGAMJLHXZXNV
```

Question 97 Tenter une attaque MCMC sur ce nouveau message. Que pouvons nous en conclure ?

```
theNewCipherText = "LWFGQXVJVNTYOJGQEB KJZVWHOPCAFRTHT DKXKVJLTRBNLXEYNCVZGAMJLHXZXNV"
theNewCipherText = tp.formatText(theNewCipherText)
print("CIPHER TEXT    = " + theNewCipherText)

run(theAlphabet, theNewCipherText, theKeySize)
print()
```

Question 98 Ce nouveau message chiffré est trop court pour que l'attaque MCMC obtienne de bons résultats. Nous ne pouvons donc pas l'attaquer ainsi.
Tenter de le déchiffrer avec la précédente clef déduite pour voir si les deux messages ont été chiffrés avec la même clef.

```
print("KPA MESSAGE    = " + tp_decipher(theAlphabet, theKpaKey, theNewCipherText))
print()
```

Question 99 Le message déchiffré semble correct. Appliquer notre connaissance de la victime pour définir notre nouveau clair connu.
Également, exécuter une nouvelle attaque à clair connu pour déduire les morceaux de clef manquants. Vérifier en déchiffrant le second message chiffré avec la clef déduite.

```
theKnown  = "JE SUIS CONTENT DE MON BEL ALGORITHME DE CHIFFREMENT CORDIALEMENT"
theKpaKey = kpa(theAlphabet, theNewCipherText, theKnown, theKeySize)
print("KPA KEY        = " + theKpaKey)
print("KPA MESSAGE    = " + tp_decipher(theAlphabet, theKpaKey, theNewCipherText))
print()
```

Question 100 Une fois la clef obtenue, déchiffrer le message initial pour en extraire le secret.

```
print("CLEAR TEXT     = " + tp_decipher(theAlphabet, theKpaKey, theCipherText))
print()
```

Question 101 Vérifier que ce secret est correct en appelant la fonction `tp.verifySecret()`.

```
tp.verifySecret("QSDFGHJK")
```

Se protéger

La contremesure qui vise à se protéger des attaques à clair connu est évidemment de tenter de ne pas répéter les mêmes mots. Seulement, cette solution n'est pas toujours possible à mettre en œuvre. Une solution plus appropriée est d'effectuer régulièrement un changement du secret.

Les systèmes de chiffrement modernes utilisent des **clefs de session** pour protéger les communications, où la durée de validité d'une clef est très courte. L'idée est que les correspondants utilisent un algorithme d'échange de clef pour convenir d'un secret partagé temporaire. Les messages sont alors chiffrés à l'aide de ce secret. Ainsi, très peu de messages sont chiffrés avec la même clef : casser une clef de session ne permet pas de déchiffrer l'ensemble des communications des correspondants.

6.3 Attaques à clair choisi / chiffré choisi

Dans la section précédente, nous avons vu les attaques à clair connu, où l'adversaire possède à la fois le message chiffré et un morceau du message clair. Dans cette section, nous regardons les concepts d'**attaques à clair choisi** (*chosen plaintext attack*, en anglais) et d'**attaques à chiffré choisi** (*chosen ciphertext attack*, en anglais).

Ces attaques sont similaires, seule la possession d'un algorithme diffère. Les attaques à clair choisi sont possibles lorsque l'adversaire peut choisir différents textes clairs et qu'il possède les mécanismes pour les chiffrer (et observer le résultat). Les attaques à chiffré choisi sont possibles lorsque l'adversaire peut choisir différents textes chiffrés et qu'il possède les mécanismes pour les déchiffrer. Dans les deux cas, l'adversaire a en sa possession un dispositif qui ne peut être désassemblé et qui réalise le chiffrement/déchiffrement. Sa tâche est alors d'en extraire la clef.

Nous allons voir dans cette section des cas d'attaques à chiffré choisi sur des algorithmes de déchiffrement parmi ceux que nous avons vu précédemment. Avec la cryptographie moderne, une cryptanalyse linéaire de l'algorithme de chiffrement/déchiffrement est généralement nécessaire, ce qui constitue un travail plus fastidieux.

Le secret est nécessaire au chiffrement/déchiffrement des messages. Les attaques à clair choisi et à chiffré choisi exploitent une faiblesse de l'implémentation de l'algorithme : même si le secret est offusqué dans l'implémentation, **rendre le mécanisme de chiffrement/déchiffrement accessible à l'adversaire lui permet une cryptanalyse**. Dans cette section, nous allons donc procéder à cette cryptanalyse.

6.3.1 Premier exemple

Question 102 Le script `tp-crypto/09/cca1.py` contient un squelette prêt pour implémenter une attaque à chiffré choisi. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/09
debian@myhostname:~/tp-crypto/09$ ./cca1.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre attaque.

Nous avons accès à l'algorithme de déchiffrement à l'aide de la fonction `tp.cca1()`. Pour le bien de l'exercice, il est suggéré de ne pas étudier son code.

Question 103 Nous devons donc procéder à une cryptanalyse. La première étape est de définir la transformation appliquée sur le message chiffré.
Nous pouvons par exemple regarder caractère par caractère. Si le message chiffré commence par la lettre `A`, quel sera le premier caractère déchiffré ?

```
print("A -> " + tp.cca1("A"))
```

Question 104 Nous pouvons aussi observer l'impact de la position du caractère dans le message chiffré. Si le message chiffré est composé de plusieurs caractères `A`, comment varie le message clair ?

```
print("AAAAAAAAAAAAA -> " + tp.cca1("AAAAAAAAAAAAA"))
```

En déduire l'algorithme de chiffrement.

Question 105 L'algorithme semble être du chiffrement par substitution monoalphabétique. Proposer une méthode pour obtenir la chef de chiffrement (le tableau des substitutions).

Question 106 Pour connaître les substitutions de toutes les lettres de l'alphabet, il suffit de fournir tout l'alphabet à l'algorithme de déchiffrement.

```
theAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
print(theAlphabet + " -> " + tp.cca1(theAlphabet))
```

Question 107 Une fois la chef de chiffrement obtenue, nous devons vérifier que celle-ci est correcte. Pour cela, nous devons ré-implémenter l'algorithme de chiffrement et tester avec la clef obtenue.
Copier l'algorithme de déchiffrement par substitution depuis les exercices précédents.

```
def tp_decipher(
  theClearAlphabet,
  theCipherAlphabet,
  theCipherText
):
  theClearText = ""
  for theImage in theCipherText:
    theIndex  = theCipherAlphabet.find(theImage)
    theLetter = theClearAlphabet[theIndex]
    #
    theClearText += theLetter
  #
  return theClearText
```

Question 108 Avec cet algorithme fait maison et la clef obtenue, déchiffrer le même message que sur l'algorithme que nous attaquons.
Est-ce vraiment la clef de chiffrement que nous avons obtenue ?

```
print(tp_decipher(theAlphabet, "COKBHUVFEYTQZLNS JPWAGRDMIX", "BONJOUR"))
```

Question 109 En réalité, nous avons la clef de déchiffrement, ce qui signifie que ne devrions déchiffrer avec l'algorithme de chiffrement, et vice-versa.

```
print(tp_decipher("COKBHUVFEYTQZLNS JPWAGRDMIX", theAlphabet, "BONJOUR"))
```

Proposer un algorithme pour obtenir la véritable clef de chiffrement.

```
theCipherAlphabet = ""
for i in range(0, len(theAlphabet)):
  theIndex = "COKBHUVFEYTQZLNS JPWAGRDMIX".find(theAlphabet[i])
  theCipherAlphabet += theAlphabet[theIndex]
#
print(theCipherAlphabet)
```

Question 110 Vérifier que cette clef est correcte.

```
print(tp.cca1("BONJOUR"))
print(tp_decipher(theAlphabet, theCipherAlphabet, "BONJOUR"))
```

6.3.2 Deuxième exemple

Question 111 Le script `tp-crypto/09/cca2.py` contient un squelette prêt pour implémenter une autre attaque à chiffré choisi. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/09
debian@myhostname:~/tp-crypto/09$ ./cca2.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre attaque.

Nous avons accès à l'algorithme de déchiffrement à l'aide de la fonction `tp.cca2()`. Pour le bien de l'exercice, il est suggéré de ne pas étudier son code.
Ici aussi, nous devons donc procéder à une cryptanalyse. La première étape est de définir la transformation appliquée sur le message chiffré. Pour gagner du temps, tentons directement de fournir l'alphabet en entier à l'algorithme.

Question 112 Tenter de déchiffrer un texte chiffré composé de toutes les lettres de l'alphabet. Observer le résultat et déduire si l'algorithme utilisé est du chiffrement par substitution monoalphabétique. Pourquoi ?

```
theAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
print(tp.cca2(theAlphabet))
```

Question 113 Des lettres différentes sont substituées par le même symbole, l'algorithme utilisé semble donc être du chiffrement par substitution polyalphabétique.
Que peut-on faire pour confirmer cette intuition ?

Question 114 Le chiffrement par substitution polyalphabétique est un chiffrement par décalage dont la valeur diffère à chaque position.
Tenter de déchiffrer 27 messages longs de 1 caractère, un message par lettre de l'alphabet. Observer les indexes des lettres substituées.

```
for theLetter in theAlphabet:
  print(theLetter + " (" + str(theAlphabet.find(theLetter)) + ")", end="")
  print(" -> ", end="")
  print(tp.cca2(theLetter) + " (" + str(theAlphabet.find(tp.cca2(theLetter))) + ")")
#
print("")
```

Conclure sur l'algorithme de chiffrement utilisé.

Pour chaque caractère placé en première position du message chiffré, le décalage est constant. Nous avons donc confirmé que nous sommes face à un chiffrement par substitution polyalphabétique.

Question 115 De combien est ce décalage pour le caractère en première position ? En déduire la fonction mathématique appliquée sur l'index de la lettre que nous transmettons à l'algorithme de déchiffrement.
Quelle lettre est-il préférable de transmettre en continu pour simplifier les calculs ?

Pour déchiffrer le message, si le caractère transmis est en première position, un décalage de 10 modulo 27 est appliqué. Pour simplifier nos calculs, nous pouvons transmettre une suite de caractères `A`, dont l'index est 0, et nous observerons directement le décalage.

Question 116 Pour le premier caractère du message, nous connaissons le décalage pour déchiffrer.
Quelle est la valeur du décalage pour chiffrer ? En déduire le premier caractère de la clef.

```
print(theAlphabet[(0 - 10) % 27])
```

Question 117 Nous allons appliquer cette stratégie pour déterminer tous les caractères de la clef de chiffrement. Mais nous ne connaissons pas sa taille.
Proposer une méthode pour déterminer la taille de la clef.

Le test de Kasiski permet de déterminer la taille de la clef de chiffrement dans le cas où nous sommes en possession d'un message chiffré dont le texte clair possède des répétitions.

Dans le cas du chiffrement par substitution polyalphabétique, la même clef est utilisée pour chiffrer et déchiffrer. Ici, nous pouvons donc faire l'inverse : effectuer un test de Kasiski sur le message clair dont le texte chiffré possède des répétitions.

Question 118 La fonction `kasiski()` permet d'effectuer un test de Kasiski sur un message chiffré.
Fournir à l'algorithme de déchiffrement un texte chiffré composé uniquement du même caractère répété en boucle et effectuer un test de Kasiski sur le résultat.

```
kasiski(tp.cca2("A" * 42))
kasiski(tp.cca2("A" * 1000))
```

Quelle est la taille de la clef ?

Question 119 Appliquer la méthode précédente sur tous les caractères d'un message aussi long que de la clef. Ceci permet d'extraire la clef de chiffrement.

```
theCipherText = "A" * 50
theClearText  = tp.cca2(theCipherText)
theKey = ""
for i in range(0, 50):
  theLetter = theClearText[i]
  theIndex  = theAlphabet.find(theLetter)
  theKey   += theAlphabet[(0 - theIndex) % 27]
#
print("KEY = " + theKey)
```

Question 120 La dernière étape est de vérifier que notre clef extraite est correcte.
Copier l'algorithme de déchiffrement par substitution polyalphabétique depuis les exercices précédents.
Tenter de déchiffrer le même message avec cet algorithme et celui que nous attaquons. Comparer les résultats.

```
def tp_decipher(
  theAlphabet,
  theSecret,
  theCipherText
):
  theClearText = ""

  for thePosition in range(0, len(theCipherText)):
    theImage      = theCipherText[thePosition]
    theImageIndex = theAlphabet.find(theImage)
    #
    theSecretLetter = theSecret[thePosition % len(theSecret)]
    theSecretIndex  = theAlphabet.find(theSecretLetter)
    #
    theIndex  = (theImageIndex - theSecretIndex) % len(theAlphabet)
    theLetter = theAlphabet[theIndex]
    #
    theClearText += theLetter

  return theClearText

print(tp.cca2("A" * 100))
print(tp_decipher(theAlphabet, "RAHGTSYCLAFNAFROFPVAVSJEZJCCWQVTO KOWQXPTBGHCGCHRJ", "A" * 100))
```

Se protéger

La contremesure qui vise à se protéger des attaques à clair choisi (ou à chiffré choisi) consiste à s'assurer d'une forte entropie lors de la transformation du message à partir d'un secret. L'objectif est que la recherche exhaustive pour un secret doit être autant que possible la meilleure attaque sur un schéma de chiffrement. Un algorithme de chiffrement est dit "cassé" si un attaquant qui ne possède pas le secret peut systématiquement retrouver le clair depuis son chiffré correspondant dans une limite de temps appropriée inférieure à la force brute.

La vérification de l'entropie d'une transformation doit être réalisée mathématiquement, une hypothèse lors de cette vérification est qu'un adversaire aura toujours connaissance du fonctionnement de l'algorithme. Ce principe a été énoncé par Auguste Kerckhoffs à la fin du 18$^{\text{ième}}$ siècle : "la sécurité d'un cryptosystème ne doit reposer que sur le secret de la clef. Autrement dit, tous les autres paramètres doivent être supposés publiquement connus".

Le principe de Kerckhoffs n'implique pas que le système de chiffrement soit public, mais seulement que sa sécurité ne repose pas sur le secret de celui-ci. Une tendance plus récente est de considérer que quand les systèmes de chiffrement sont publics, largement étudiés et qu'aucune attaque significative n'est connue, ils sont d'autant plus sûrs.

En conclusion, se protéger des attaques à clair choisi (ou à chiffré choisi), c'est s'assurer qu'aucun accès direct ou indirect au secret puisse être réalisé par l'adversaire. Les systèmes de chiffrement modernes allient chiffrement par blocs (où le chiffrement n'est pas réalisé caractère par caractère), tables de substitutions fixes et transformations à partir du secret (généralement avec des fonctions logiques *ou exclusif*). Ces opérations sont également répétées plusieurs fois (plusieurs étages de chiffrement).

6.4 Attaques par canaux auxiliaires

Dans le domaine de la sécurité informatique, une attaque par canal auxiliaire (en anglais : *side-channel attack*) est une attaque informatique qui, sans remettre en cause la robustesse théorique des méthodes et procédures de sécurité, recherche et exploite des failles dans leur implémentation, logicielle ou matérielle. En effet, une sécurité "mathématique" ne garantit pas forcément une sécurité lors de l'utilisation en "pratique".

Une attaque est considérée comme utile dès lors qu'elle présente des performances supérieures à une attaque par force brute. Les attaques par canal auxiliaire sont nombreuses et variées et portent sur différents paramètres. On en distingue deux grandes catégories :

— les attaques invasives, qui nécessitent une interaction avec le matériel (potentiellement destructive).
— les attaques non invasives, qui se contentent de procéder à une observation extérieure du système (passive).

Source : `https://fr.wikipedia.org/wiki/Attaque_par_canal_auxiliaire`

Dans cette section, nous allons procéder une attaque non invasive sur un algorithme de chiffrement en exploitant un canal temporel.

Question 121 Le script `tp-crypto/10/sidechannel.py` contient un squelette prêt pour implémenter une attaque par canal auxiliaire. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-crypto/10
debian@myhostname:~/tp-crypto/10$ ./sidechannel.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre attaque.

Ce script contient un algorithme permettant le chiffrement d'un message à l'aide du chiffrement par substitution polyalphabétique. Afin de protéger l'algorithme contre les attaques à clair connu, son concepteur a implémenté une vérification de la clef de chiffrement avant de procéder au chiffrement. Si la clef fournie ne correspond pas au secret prédéfini, alors le chiffrement n'a pas lieu.

Question 122 Après exécution du script, observer la taille attendue pour la clef.
Exécuter l'algorithme avec une clef de chiffrement de la bonne taille et observer le résultat.

```
print("BONJOUR -> " + sidechannel("BONJOUR", "A" * 20))
```

Question 123 Les caractères qui composent la clef de chiffrement sont tous issus de l'alphabet de 27 lettres. Connaissant la taille de la clef, combien de possibilités a-t-on pour le secret ?

```
math.pow(27, 20)
```

Question 124 Avec la possibilité de tester 100 clefs par seconde, si nous devions effectuer une attaque par force brute sur cette clef, combien de temps serait nécessaire ?

```
import math
theNum = math.pow(27, 20)
print("Il faudrait " + str(theNum / 100) + " secondes")
print("soit " + str(theNum / 100 / 3600) + " heures")
print("soit " + str(theNum / 100 / 3600 / 24) + " jours")
print("soit " + str(theNum / 100 / 3600 / 24 / 365.25) + " annees")
print("soit " + str(theNum / 100 / 3600 / 24 / 365.25 / 13.77e9) + " fois l'age de l'univers")
```

Question 125 Observer le code de la fonction `sidechannel()`, en particulier les instructions où la clef est vérifiée. La fonction `usleep()` met le programme en pause pendant un certain nombre de microsecondes.
Si aucun caractère de la clef n'est correct, combien de temps s'écoulera-t-il avant que la fonction ne retourne une erreur ?

Question 126 Si seulement le premier caractère de la clef est correct, combien de temps s'écoulera-t-il avant que la fonction ne retourne une erreur ?

Question 127 Même question avec les deux premiers caractères de la clef.
Déduire une manière d'attaquer cette fonction (plus efficace que la force brute).

Question 128 Plutôt que d'effectuer une attaque par force brute sur toute la clef, nous pouvons tester caractère par caractère et observer la durée d'exécution de la fonction `sidechannel()`.
En admettant que nous pouvons identifier chaque caractère après avoir testé toutes les lettres de l'alphabet, combien d'itérations sont nécessaires pour venir à bout de la clef ?

```
27 * 20
```

Notons que sur cette fonction, nous ne pouvons pas tester 100 clefs par secondes. En effet, l'attente que celle-ci réalise est supérieure ou égale à $10ms$.

Question 129 Effectuer une attaque par force brute sur le premier caractère. À chaque itération, exécuter 100 fois la fonction `sidechannel()` et mesurer le temps d'exécution. Sauvegarder les temps d'exécution dans une liste.

```
theTimeList = list()
for theLetter in theAlphabet:
  theTime = time.time()
  for i in range(0, 100):
    sidechannel("BONJOUR", theLetter + "A"*19)
  theTimeList.append(time.time() - theTime)
```

Question 130 Tracer les temps d'exécution de la fonction `sidechannel()` pour chaque valeur testée.

```
x = range(0, len(theAlphabet))
y = theTimeList
fig, ax = matplotlib.pyplot.subplots()
ax.set(xticks=range(0, len(theAlphabet)), xticklabels=list(theAlphabet))
ax.plot(x, y)
ax.set_title("Computation for letter at index 0")
matplotlib.pyplot.show()
```

Quelle est la valeur probable pour le premier caractère de la clef secrète ?

Question 131 Sauvegarder ce premier caractère et effectuer une nouvelle attaque par force brute, cette fois-ci sur le second caractère. Continuer de mesurer le temps d'exécution de 100 fois la fonction `sidechannel()` et le sauvegarder dans une liste.

```
theTimeList = list()
for theLetter in theAlphabet:
  theTime = time.time()
  for i in range(0, 100):
    sidechannel("BONJOUR", "B" + theLetter + "A"*18)
  theTimeList.append(time.time() - theTime)
```

Question 132 De nouveau, tracer le temps d'exécution de la fonction `sidechannel()` pour chaque valeur testée. En déduire la valeur probable pour le second caractère de la clef secrète.

```
x = range(0, len(theAlphabet))
y = theTimeList
fig, ax = matplotlib.pyplot.subplots()
ax.set(xticks=range(0, len(theAlphabet)), xticklabels=list(theAlphabet))
ax.plot(x, y)
ax.set_title("Computation for letter at index 1")
matplotlib.pyplot.show()
```

Question 133 Toujours en suivant cette stratégie, implémenter un algorithme qui réalise une attaque par force brute caractère par caractère. À chaque nouveau caractère, sauvegarder dans une clef candidate celui qui a provoqué l'exécution la plus longue.
Il est possible d'afficher la clef candidate pour connaître l'état d'avancement de l'algorithme.
Note : durant l'exécution de cette attaque, il est conseillé de ne pas interagir avec la machine pour ne pas modifier les temps d'exécution. En effet, exécuter un autre programme peut provoquer une mise en pause de l'interpréteur `Python` par le système d'exploitation et fausser les résultats.

```
theKey = ""
#
for theIndex in range(0, 20):
  theTimeList = list()
  for theLetter in theAlphabet:
    theCandidate = theKey + theLetter + "A"*(19 - len(theKey))
    print(theCandidate)
    # brute force the char and calculate the execution time:
    theTime = time.time()
    for i in range(0, 100):
      sidechannel("BONJOUR", theCandidate)
    theTimeList.append(time.time() - theTime)
  # the letter is the one that made it take the max time:
  theKey += theAlphabet[theTimeList.index(max(theTimeList))]
#
print("Found a key: " + theKey)
```

Question 134 Cet algorithme permet d'extraire une clef probable. Vérifier si celle-ci est correcte en tentant de chiffrer une chaîne de caractères.

```
print("BONJOUR -> " + sidechannel("BONJOUR", "BSDMSVMNALKXOULEWWKC"))
```

Question 135 Observer de nouveau le code de la fonction `sidechannel()`. Si l'appel à la fonction `usleep()` venait à être supprimé, une attaque par canal auxiliaire serait-elle toujours envisageable ? Pourquoi ?
En déduire la véritable vulnérabilité de la fonction `sidechannel()`.

Autres attaques par canaux auxiliaires

Dans cette section, nous avons réalisé une attaque par canal temporel. Il existe d'autres classes d'attaques par canaux auxiliaires où une information sur le secret n'est pas obtenue par mesure temporelle. Nous pouvons citer par exemple, comme attaques invasives :

- l'**attaque par sondage**, qui consiste à placer une sonde directement dans le circuit à étudier, afin d'observer son comportement.
- l'**attaque par injection de faute**, qui consiste en l'introduction volontaire d'erreurs dans le système pour provoquer certains comportements révélateurs (par exemple, à l'aide d'une impulsion laser).

Et comme attaques non invasives :

- l'**analyse d'émanations électromagnétiques**, qui consiste en l'étude du rayonnement électromagnétique généré par un ordinateur ou une machine qui chiffre/déchiffre.
- l'**analyse de consommation**, où une consommation accrue indique un calcul important et peut donner des renseignements sur la clef.
- l'**attaque par prédiction de branchement**, qui consiste à étudier les unités de prédiction de branchement des nouvelles architectures de processeurs.

Se protéger

Les constructeurs de systèmes de chiffrement visent à aplanir la courbe de consommation électrique pour dissimuler les opérations sous-jacentes. Des protections et des blindages des systèmes matériels permettent de limiter le rayonnement en dehors du circuit. Certains systèmes implémentent également des opérations inutiles afin d'offusquer leur trace d'exécution.

Il faut également tenir compte des états impossibles qui ne doivent pas se produire et doivent être traités correctement s'ils venaient à être détectés, dans ce cas on parlera de tolérance aux fautes ou d'élimination des vulnérabilités.

Rendre un système tolérant aux fautes nécessite la détection et le recouvrement lorsqu'une erreur se produit. Éliminer les vulnérabilités repose sur de la vérification formelle (preuve mathématique).

Limiter les messages d'erreur et la communication d'informations diverses avec l'extérieur est aussi une solution, mais elle pénalise les développeurs et les utilisateurs du système.

Dans le cas des attaques par canaux temporels comme nous l'avons vu dans cette section, une propriété mathématique à vérifier est l'**indépendance au secret**. Un algorithme qui vérifie cette propriété exécutera toujours les mêmes étapes, quelque soit le secret utilisé par le système et la clef transmise par l'utilisateur. L'indépendance au secret est vérifiée par un calcul du nombre de cycles nécessaires à l'exécution de l'algorithme. Elle doit être vérifiée tout au long de la conception du système : de sa définition mathématique à son implémentation finale.