Partie 1 – Cryptographie : le chiffrement RSA

On cherche à démontrer φ(n) = (p-1)(q-1).

D'après l'énoncé "la fonction indicatrice d'Euler associe à tout entier naturel *n* non nul le cardinal, noté φ(*n*), de l'ensemble des nombres naturels non nuls inférieurs ou égaux à et premiers avec *n)*" soit le cardinal de l'ensemble suivant que l'on appelle E : $\{k \in \mathbb{N}^* \mid k \leq n \; et \; k \wedge n = 1\}$
On peut décomposer le cardinal de E en deux parties :
card(E) = card($\{k \in \mathbb{N}^* \mid k \leq n\}$) - card($\{k \in \mathbb{N}^* \mid k \wedge n \neq 1\}$)

L'ensemble $\{k \in \mathbb{N}^* \mid k \leq n\}$ est simplement la liste des chiffres de 1 à n inclus donc le cardinal est *n (*soit *pq* car pour générer la clé on calcule *n=pq)*.

$\{k \in \mathbb{N}^* \mid k \wedge n \neq 1\} = \{k \in \mathbb{N}^* \mid k \wedge pq \neq 1\}$
$k \wedge pq \neq 1 \Rightarrow \exists m \in \mathbb{N}^* \setminus \{1\} \; tel \; que \; k \wedge pq = m$ on peut donc en déduire que *m|k* ainsi que *m|pq*.
On peut en déduire que *m|p* ou *m|q* ou *m|pq* et donc que $q = \lambda m$ ou $p = \lambda m$ ou $pq = \lambda m$
On peut donc chercher les 3 ensembles regroupant les k qui divisent ou *q* ou *p* ou *pq* :

- $\{k \in \mathbb{N}^* \mid k \leq pq \; et \; k \wedge pq = q\}$
- $\{k \in \mathbb{N}^* \mid k \leq pq \; et \; k \wedge pq = p\}$
- $\{k \in \mathbb{N}^* \mid k \leq pq \; et \; k \wedge pq = pq\}$ -> donc k = pq

Pour chaque ensemble on peut voir que le pgcd (donc ou q ou p ou pq) divise k, on peut donc écrire:

- $k \leq pq$ et $k = \lambda q \Leftrightarrow \lambda q \leq pq \Leftrightarrow \lambda \leq p$
- $k \leq pq$ et $k = \lambda p \Leftrightarrow \lambda p \leq pq \Leftrightarrow \lambda \leq q$
- $k \leq pq$ et $k = \lambda pq \Leftrightarrow \lambda pq \leq pq \Leftrightarrow \lambda \leq 1$ -> donc $\lambda = 1$

Donc $card(\{k \in \mathbb{N}^* \mid k \wedge pq \neq 1\})$ =
$card(\{\lambda \in \mathbb{N}^* \mid \lambda \leq p\} \cup \{\lambda \in \mathbb{N}^* \mid \lambda \leq q\}) - 1)$
$\Leftrightarrow$
$card(\{k \in \mathbb{N}^* \mid k \wedge pq \neq 1\}) = p + q - 1$

On peut donc déduire que
$card(\{k \in \mathbb{N}^* \mid k \leq n\}) - card(\{k \in \mathbb{N}^* \mid k \wedge n \neq 1\}) = pq - (p + q - 1)$
$= pq - p - q + 1 = (p - 1)(q - 1)$

Les fonctions de cette partie sont dans le code source accompagnées de leurs commentaires

Partie 2 – Codes correcteurs *(preuves par Python)*

Q2.1 Tables d'opération de F_2 :

+	0	1
0	0	1
1	1	0

.	0	1
0	0	0
1	0	1

Q2.2

L'ensemble F_2^4 est donc l'ensemble des vecteurs possibles allants du vecteur nul (0,0,0,0) à (1,1,1,1). Grâce à python nous pouvons montrer qu'à l'aide de e1,e2,e3 et e4 il est possible d'exprimer tous les vecteurs non nulles de F_2^4, donc un ensemble de cardinal 16 (4x4 possibilités de combinaisons) :

```python
def vector_set(): ## Preuve de la Q2.2
    vectors = [
    np.array([1,0,0,0]),
    np.array([0,1,0,0]),
    np.array([0,0,1,0]),
    np.array([0,0,0,1])
    ]
    result = [] #Ensemble F4²
    result.append(np.array([0,0,0,0])) ## Vecteur nulle qui appartient forcément à F4²
    for i in range(len(vectors)):
        result.append(vectors[i])
        for j in range(i+1,len(vectors)):
            result.append(vectors[i]+vectors[j])
            for k in range(j+1,len(vectors)):
                result.append(vectors[i]+vectors[j]+vectors[k])
                for v in range(k+1,len(vectors)):
                    result.append(vectors[i]+vectors[j]+vectors[k]+vectors[v])
    for el in result:
        print(el)
    print("card :",len(result))
```

```
[~/S102-Comparaison-Algo]$ python3 -i  SAE_S1_02_Corr.py
>>> vector_set()
[0 0 0 0]
[1 0 0 0]
[1 1 0 0]
[1 1 1 0]
[1 1 1 1]
[1 1 0 1]
[1 0 1 0]
[1 0 1 1]
[1 0 0 1]
[0 1 0 0]
[0 1 1 0]
[0 1 1 1]
[0 1 0 1]
[0 0 1 0]
[0 0 1 1]
[0 0 0 1]
card : 16
>>>
```

Q2.3

L'image de l'application linéaire est :

Fonctions utilisées trouvables et commentées dans SAE_S1_02_Corr.py

```
>>>
>>> image = im(vector_set())
>>> for i in image:
...     print(i)
...
[0 0 0 0 0 0 0]
[1 1 1 0 0 0 0]
[0 1 1 1 1 0 0]
[0 0 1 0 1 1 0]
[1 1 1 1 1 1 1]
[1 0 1 0 1 0 1]
[1 0 1 1 0 1 0]
[0 1 1 0 0 1 1]
[0 0 1 1 0 0 1]
[1 0 0 1 1 0 0]
[1 1 0 0 1 1 0]
[0 0 0 1 1 1 1]
[0 1 0 0 1 0 1]
[0 1 0 1 0 1 0]
[1 0 0 0 0 1 1]
[1 1 0 1 0 0 1]
>>>
```

On l'obtient tout simplement en multipliant tous les vecteurs obtenus à la question précédente par la matrice (image de ces vecteurs par l'application). Niveau code on itère tout simplement à travers la liste des vecteurs donnés par la fonction précédente et on fait matrice*vecteur (numpy.matmul).

Q2.4

Pour cette question nous avons aussi démontré à l'aide d'un algorithme qui itère chaque image et pour chaque image on vérifie la distance avec toutes les autres. (la distance $d(v,u)$correspondant au poids de $v + u$ pour tous $u, v \in F_2^m$ avec ici m=7

```
def check():
    vecset = vector_set()
    image = im(vecset)
    for i in image:
        for j in image:
            if np.array_equal(i,j):
                continue
            if poids((i+j)%2) < 3:
                print("La distance entre",i,"et",j,"est inférieur à 3")
                return
    print("Toute les combinaisons possibles de u et v donnent d(u,v) >= 3")
    return
```

```
Windows PowerShell
PS E:\IUT\S102-Comparaison-Algo> python -i .\SAE_S1_02_Corr.py
Toute les combinaisons possibles de u et v donnent d(u,v) >= 3
>>>
```

Q2.5

Pour cette dernière question il fallait démontrer que pour tout $u, v \in Im(\phi)$, $v \neq u$ on a $d(u, \sim u) < d(v, \sim u)$ donc nous avons écrit une fonction spécialement à cet effet qui parcourt chaque vecteur de l'image (il s'agit de u) et qui vérifie pour chaque bit, qu'un vecteur $\sim u$ copie de u avec un bit altéré (bit en question +1 puis modulo 2) est toujours à une distance plus courte de u que d'un autre vecteur $v \in Im(\phi)$.

```
def checkQ25():
    image = im(vector_set())
    for vecU in image:
        for bit in range(len(vecU)):
            vecUdiff = np.copy(vecU)
            vecUdiff[bit] = (vecUdiff[bit]+1)%2
            for vecV in image:
                if np.array_equal(vecV,vecU):
                    continue
                if poids((vecU+vecUdiff)%2) > poids((vecU+vecV)%2):
                    return "La distance entre u et ~u est plus grande qu'entre u et v !"
    return "La distance entre u et ~u est toujours la plus petite !"

print(checkQ25())
```

```
Windows PowerShell
PS E:\IUT\S102-Comparaison-Algo> python .\SAE_S1_02_Corr.py
La distance entre u et ~u est toujours la plus petite !
PS E:\IUT\S102-Comparaison-Algo>
```

Partie 3 – Communication sécurisée

Dans ce bilan nous "simulons" un envoie chiffré, pour celà on commence par générer nos clés à l'aide de la fonction écrite à la première partie du projet. On demande ensuite une entrée utilisateur (le message à chiffrer) que l'on transforme ensuite en liste de caractère ASCII. Afin d'éviter les vulnérabilités liées à l'analyse fréquentielle nous concaténons ensuite les code ASCII dans une séquence (un string) en suivant un format/une procédure précis(e) avant d'encrypter :

- Un caractère est identifié par un code à 3 chiffres
- Si le code ASCII initial du caractère est sur moins de 3 chiffres on ajoute les 0 manquant, ce qui permet d'englober du code 97 à 122 (a-z, même si le programme peut coder toute la table ascii mis à part les codes < 30)
- Pour repasser sur le code ASCII original on vérifie si le 3ème chiffre est 0 et que le premier est supérieur à 2 afin de le remettre sur 2 chiffres si besoin. Exemple: 97 sera codé comme 970 on voit que le 3ème chiffre est 0 et le premier est supérieur à 2 on retire le 0 on obtient 97.

Vient ensuite le moment de chiffrer en utilisant la fonction **encryption(msg,key)** lors de la première partie, nous chiffrons la séquence de codes ASCII découpée en groupe de 4 chiffres (¾ chiffres représentent un code ascii et le dernier le premier chiffre du prochain) afin de brouiller l'analyse fréquentielle comme dit auparavant. Pour le passage en binaire nous n'avons pas réussi à trouver un moyen de découper directement les parties du message en binaire en morceaux de 4 bits sans perdre les représentations uniques et nous avons donc procéder autrement. Nous ré-utilisons notre principe de séquence mais cette fois pour les codes ASCII chiffrés qui sont donc mis à la chaîne.

Chaque chiffre de cette séquence est donc codé sur 4 bits (ce qui augmente l'occupation de la mémoire mais rend la compréhension plus simple pour nous) et regroupé dans des tableaux par code ASCII chiffré.

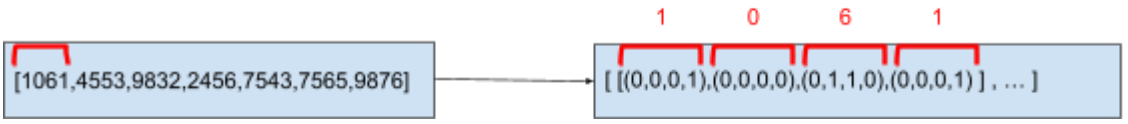

Schéma explicatif de notre codage en vecteurs binaires

Ces vecteurs binaires sont ensuite passés dans l'application linéaire afin de les transformer en vecteurs binaires de 7 bits. On "transmet" ensuite ces données qui sont soumises à un bruitage aléatoire. L'intérêt de l'application linéaire qui transforme nos vecteurs de 4 bits vers des vecteurs de 7 bits est de pouvoir appliquer la théorie de la deuxième partie : pour tout $u, v \in Im(\phi),\ v \neq u$ on a $d(u, \sim u) < d(v, \sim u)$. Pour chaque vecteur bruité reçu nous cherchons dans $Im(\phi)$ un vecteur u qui ne diffère que d'un ou aucun bit, c'est à dire le vecteur u tel que $d(u, \sim u)$ est la plus petite distance possible ($\sim u$ est notre vecteur bruité). Le vecteur u trouvé est donc le vecteur corrigé.

Grâce à une fonction cherchant l'antécédent de notre vecteur u par l'application ϕ (trouvable commentée dans *SAE_S1_02_Corr.py*) on récupère le vecteur sur 4 bits d'origine et on ré-assemble les codes ASCII chiffrés comme décrit précédemment. On peut ensuite déchiffrer les codes ASCII en les reprenant 3 à 3 et en retirant les 0. On obtient donc les codes ASCII déchiffrés d'origine que l'on a plus qu'à re-convertir en caractères, on obtient enfin le texte d'origine.