MASTER 1 MEEF : CAPES MATHS OPTION INFORMATIQUE

ECRIT BLANC - EPREUVE DE 5 HEURES

Les résultats des questions pourront être réutilisés ainsi que les différentes fonctions demandées non traitées au cours de l'épreuve. L'usage de la calculatrice est interdit.

Partie A : Cryptage affine

Le but de ce problème est l'étude d'un procédé de cryptage des lettres majuscules de l'alphabet français. Chacune des 26 lettres est associée à l'un des entiers de 0 à 25, selon le tableau de correspondance suivant.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Le cryptage se fait à l'aide d'une clé, qui est un nombre entier k fixé, compris entre 1 et 25. Pour crypter une lettre donnée on suit le processus $\wp$ suivant :

- on repère le nombre x associé à la lettre dans le tableau de correspondance précédent
- on multiplie ce nombre x par la clé k
- on calcule le reste r de la division euclidienne du nombre obtenu par 26
- on repère la lettre associée au nombre r dans le tableau de correspondance, qui devient la lettre cryptée

Par exemple, pour crypter la lettre P avec la clé $k = 11$:

- le nombre x associé à la lettre P est le nombre 15
- on multiplie 15 par la clé k, ce qui donne $11 \times 15 = 165$
- on calcule le reste de la division euclidienne par 26 : on obtient 165 % 26 = 9
- on repère finalement la lettre associée à 9 dans le tableau, c'est-à-dire J

Ainsi avec la clé k = 11, la lettre P est cryptée en la lettre J.

On crypte un mot en cryptant chacune des lettres de ce mot. En Python, on crée une liste L qui contient les 26 lettres de l'alphabet rangées dans l'ordre alphabétique à l'aide de la commande ci-dessous :

```
L = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

1. Que penser du cryptage obtenu lorsque la clé k est égale à 1 ?

Lorsque la clé est égale à 1, pour tout $x \in [0,25], k \times x = x$ donc le reste de la division euclidienne de $k \times x$ par 26 est x. Chaque lettre est cryptée en elle-même, le cryptage n'a donc pas d'effet.

2. En quoi la lettre A constitue-t-elle un cas particulier dans le processus de cryptage ?

Lorsque la lettre est A, d'indice x = 0, pour tout $k \in [1,25], k \times x = 0$ et le reste de la division euclidienne de $k \times x$ par 26 est 0. La lettre A est cryptée par elle-même, quelle que soit la clé de cryptage utilisée. Elle est invariante.

3. Dans le cas où la clé est égale à 11, crypter le mot MIRO.

Pour la lettre M d'indice 12, on a $12 \times 11 = 132$ et $132 \% 26 = 2$, indice correspondant à la lettre C
Pour la lettre I d'indice 8, on a $8 \times 11 = 88$ et $88 \% 26 = 10$, indice correspondant à la lettre K
Pour la lettre R d'indice 17, on a $17 \times 11 = 187$ et $187 \% 26 = 5$, indice correspondant à la lettre F
Pour la lettre O d'indice 14, on a $14 \times 11 = 154$ et $154 \% 26 = 24$, indice correspondant à la lettre Y
Le mot MIRO est donc crypté par le mot CKFY.

4. Ecrire une fonction **indice** qui prend en paramètre une lettre de l'alphabet et qui renvoie son indice dans la liste L (L étant supposée définie comme variable globale).

```
def indice (lettre) :
   return L.index(lettre)
```

Ou bien (sans la fonction index) :

```
def indice (lettre) :
   i = 0
   while L[i] != lettre :
      i += 1
   return i
```

5. Ecrire une fonction **crypterLettre** qui prend en paramètre une chaîne de caractères constituées d'une lettre majuscule de l'alphabet et une clé et qui renvoie la lettre cryptée. Cette fonction utilisera la fonction **indice** précédente. En supposant qu'un appel à la fonction **indice** compte pour une opération élémentaire, quel est le nombre d'opérations élémentaires effectuées à chaque appel de la fonction **crypterLettre** ?

```
def crypterLettre (lettre, k) :
   c = (indice(lettre)*k)%26
   return L[c]
```

Le nombre d'opérations élémentaires effectuées à chaque appel de la fonction crypterLettre est 4 : un appel à indice, une multiplication, un modulo, et une affectation.

6.
(a) Ecrire une fonction **crypterTexte** qui prend en paramètre une chaîne de caractères dont les éléments sont soit des lettres majuscules soit des espaces (qui ne seront pas modifiés), et une clé. La fonction renvoie la chaîne de caractères cryptés.

```
def crypterTexte (texte, k) :
   T = ‘’
   for lettre in texte :
      if lettre in L :
         T += crypterLettre(lettre,k)
      else :
         # tout ce qui n’est pas une majuscule
         T += lettre
   return T
```

(b) Donner l’appel à effectuer pour répondre à la question 3

```
crypterTexte(‘MIRO’,11)
```

(c) Soit N la longueur de la chaîne de caractères en paramètre de la fonction **crypterTexte** et M le nombre d’espaces qu’elle contient. Déterminer en fonction de N et M la complexité de la fonction **crypterTexte**.

On compte une opération pour l’initialisation de T puis, pour chaque caractère, il y a un test et un ajout dans T. Si le caractère est un espace, c’est tout, sinon il faut ajouter un appel à crypterLettre (coût de 4), ce qui totalise :

$$1 + 2N + 4 \times (N - M) = 1 + 6N - 4M$$

7. On dit qu’une clé est une bonne clé de cryptage si elle possède une clé associée k’, qui est un nombre entier compris entre 1 et 25 tel qu’en appliquant le processus $\wp$ avec la clé k’ à une lettre cryptée (avec la clé k) on obtient la lettre initiale. k’ est alors appelée clé de décryptage associée à k.

On admet qu’une clé k est une bonne clé de cryptage si et seulement si $k \neq 1$ et $k \wedge 26 = 1$, c’est-à-dire si k est différent de 1 et si le seul diviseur commun dans $\mathbb{N}$ à k et à 26 est 1.

(a) On suppose que 19 est une clé de décryptage associée à la clé k = 11. Avec la clé k = 11, un mot a été crypté. On a obtenu le mot HARK. Retrouver le mot initial à l’aide de la clé de décryptage « à la main », puis donner l’appel à la fonction **crypterTexte** donnant le même résultat.

Le nombre associé à la lettre H est 7. On a $7 \times 19 = 133$ et $133 \,\%\, 26 = 3$ donc décrypté en D
La lettre A est invariante.
Le nombre associé à la lettre R est 17. On a $17 \times 19 = 323$ et $323 \,\%\, 26 = 11$ donc décrypté en L
Le nombre associé à la lettre K est 10. On a $10 \times 19 = 190$ et $190 \,\%\, 26 = 8$ donc décrypté en I
Le mot initial est donc DALI.

En utilisant la fonction, on écrit l’instruction :

```
crypterTexte(‘HARK’,19)
```

(b) Soit F la liste qui contient les termes de la suite de Fibonaci strictement inférieurs à 26 rangés par ordre croissant. On rappelle que la suite de Fibonacci est définie par ses deux premiers termes 0 et 1 et par le fait qu’à partir de son troisième terme, chaque terme est égal à la somme des deux précédents. Expliciter F.

$F = [0,1,1,2,3,5,8,13,21]$

(c) Déterminer la liste G des éléments de F qui sont de bonnes clés de cryptage.

Par définition, 0 n'est pas une clé de cryptage et 1 n'est pas une bonne clé de cryptage.

$2 \wedge 26 = 2 \neq 1$ $3 \wedge 26 = 1$ $5 \wedge 26 = 1$ $8 \wedge 26 = 2 \neq 1$

$13 \wedge 26 = 13 \neq 1$ $21 \wedge 26 = 1$

Donc $G = [3,5,21]$

(d) On admet que la clé de décryptage k' associée à une bonne clé de cryptage k est unique et que c'est le nombre entier strictement compris entre 0 et 26 qui est tel que le reste de la division euclidienne par 26 de $k \times k'$ est 1. Vérifier que 19 est la clé de décryptage associée à la clé k = 11.

Le reste de la division euclidienne par 26 de $11 \times 19 = 209$ est 1, donc 19 est la clé de cryptage associée à la clé k = 11.

8. Ecrire une fonction **cleDecryptage** qui prend en paramètre un entier k premier avec 26 et qui renvoie la clé de décryptage associée k'. On dispose de bonnes clés de cryptage avec les éléments de G, la fonction **cleDecryptage** nous permet ainsi de calculer aussi les clés de décryptage associées.

```
def cleDecryptage (k) :
   # k est premier avec 26
   for i in range(26) :
      if (i*k) % 26 == 1 :
        return i
```

9. Un petit futé cherche à décrypter un long message crypté écrit en français sans connaître ni la clé de cryptage, ni la clé de décryptage. Pour cela, il repère dans la chaîne de caractères que constitue le message la lettre la plus fréquente et en déduit que c'est la lettre cryptée qui correspond à E (la lettre la plus utilisée en français).

(a) Que renvoie la fonction suivante ? On ne justifiera pas la réponse.

```
import numpy as np
def mystere (C) :
   """ C est une chaîne de caractères dont les éléments sont soit des lettres
   majuscules, soit des espaces """
   V = np.zeros(26)
   max, indexMax = 0 , 0
   for j in range(26) :
      for i in range(len(C)) :
        if C[i] == L[j] :
           V[j] += 1
      if V[j] > max :
        max, indexMax = V[j], j
   return L[indexMax]
```

La fonction mystere renvoie la lettre la plus fréquente dans la chaîne de caractères C passé en paramètre.

(b) Donner une preuve rapide de la terminaison de cette fonction.

La boucle portant sur i est exécutée `len(C)` fois et le corps de cette boucle ne contient qu'une comparaison et une addition/affectation. Elle se termine donc.
La boucle portant sur j est exécutée 26 fois et le corps de cette boucle contient la boucle portant sur i, une comparaison et deux affectations. La fonction se termine donc bien.

(c) *Question bonus.* En utilisant les fonctions **`count`**, **`max`** et **`index`** de Python, réécrire la fonction précédente avec le moins de lignes possible, et donner lui aussi un nom plus explicite.

```
def maxOccurence (C) :
   V = [C.count(L[j]) for j in range(26)]
   return L[V.index(max(V))]
```

10. Connaissant E et la lettre cryptée correspondante, le petit futé peut en déduire la clé de cryptage et ainsi « cracker » le code. Ecrire une fonction **`cracker`** qui prend en paramètre la lettre cryptée correspondant à E et qui renvoie la liste des clés de cryptage possibles (liste pouvant être vide) permettant de crypter E en cette lettre.

On parcourt toutes les lettres possibles, et on conserve celles qui coderaient la lettre E (d'indice 4) en la lettre en paramètre.

```
def cracker (lettre) :
   N = []
   for i in range(26) :
      if (4*i) % 26 == indice(lettre) :    # ou: crypterLettre('E',k) == lettre
         N += [i]
   return N
```

11. Que se passe-t-il s'il essaie de « cracker » un message crypté où la lettre E n'est pas la lettre la plus fréquente du message ?

Il ne parviendra pas à « cracker » le message puisque la clé de décryptage calculée ne sera pas la bonne (indice de la lettre E de 4 utilisé pour décrypter une lettre qui n'est pas E). Un message « décrypté » sera obtenu mais pas le bon.

Partie B : Codages de César et Vigenère

On cherche à crypter un texte t de longueur n composé de caractères en minuscules (soit 26 lettres différentes) représentés par des entiers compris entre 0 et 25 (0 pour a, 1 pour b, etc.). Nous ne tenons pas compte des éventuels espaces. Ainsi, le texte `capesmathoptioninfo` est représenté par le tableau suivant où la première ligne représente le texte, la seconde les entiers correspondants, et la troisième les indices dans le tableau t.

c	a	p	e	s	m	a	t	h	o	p	t	i	o	n	i	n	f	o
2	0	15	4	18	12	0	19	7	14	15	19	8	14	13	8	13	5	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Dans cette partie, on supposera que les textes sont déjà représentés par des listes de nombres et que donc toutes les fonctions de codage et décodage prendront en arguments des listes de nombres et renverront des listes de nombres.

B.1 Codage de César

Ce codage est le plus rudimentaire que l'on puisse imaginer. Il a été utilisé par Jules César (et même auparavant) pour certaines de ses correspondances. Le principe est de décaler les lettres de l'alphabet vers la droite d'une ou plusieurs positions. Par exemple, en décalant les lettres d'une position, le caractère a se transforme en b, le b en c, etc. et le z en a. Le texte avecesar devient donc bwfdftbs.

12. Que donne le codage du texte `info` en utilisant un décalage de 5 ?

Le texte crypté est nskt.

13. Ecrire une fonction **`codageCesar`** qui prend en paramètres une liste t d'entiers (représentant un texte) et un entier d et qui retourne une liste de même taille que t représentant le texte décalé de d positions.

```
def codageCesar (t,d) :
   return [(i+d)%26 for i in t]
```

14. Ecrire de même une fonction **`decodageCesar`** qui prend les mêmes paramètres mais qui réalise le décalage dans l'autre sens (vers la gauche).

```
def decodageCesar (t,d) :
   return codageCesar(t,-d)
```

Pour réaliser ce décodage, il faut connaître la valeur de décalage. Une manière de la déterminer automatiquement est d'essayer de deviner cette valeur (afin de « cracker » le code). L'approche la plus couramment employée consiste à regarder la fréquence d'apparition de chaque lettre dans le texte crypté. On utilise ensuite encore le fait que la lettre e est la plus fréquente en français.

15. Ecrire une fonction **`frequenceLettres`** qui prend en paramètre une liste t représentant le texte crypté et qui retourne une liste de taille 26 dont la case d'indice i contient le nombre d'apparitions du nombre i dans t divisé par la longueur de t.

Pour déterminer la fréquence d'apparition d'un symbole, on compte ses occurrences et on divise par le nombre total de symboles.

```
def frequenceLettres (t) :
   return [t.count(i) / len(t) for i in range(26)]
```

16. Ecrire une fonction **decodageAuto** qui prend en argument une liste t représentant le texte crypté, et qui retourne le texte d'origine (en calculant le décalage pour que la lettre e soit la plus fréquente dans le message décrypté).

L'indice du maximum de la liste des fréquences de t doit correspondre au chiffre 4 (lettre e) décalé.

```
def decodageAuto (t) :
   f = frequenceLettres(t)
   d = ( f.index(max(f)) - 4 ) % 26
   return decodageCesar(t,d)
```

B.2 Codage de Vigenère

Au XVI^{e} siècle, Blaise de Vigenère a modernisé le codage de César, très peu résistant, de la manière suivante. Au lieu de décaler toutes les lettres du texte de la même manière, on utilise un texte clé qui donne une suite de décalages. Prenons par exemple la clé `concours`. Pour crypter un texte, on code la première lettre en utilisant le décalage qui renvoie a sur c (la première lettre de la clé). Pour la deuxième lettre, on prend le décalage qui envoie le a sur le o (deuxième lettre de la clé) et ainsi de suite. Pour la neuvième lettre du texte (la clé en contient que huit) on reprend à partir de la première lettre de la clé. Sur l'exemple `capesmathoptioninfo` avec la clé `concours`, on obtient :

c	a	p	e	s	m	a	t	h	o	p	t	i	o	n	i	n	f	o
2	0	15	4	18	12	0	19	7	14	15	19	8	14	13	8	13	5	14
c	o	n	c	o	u	r	s	c	o	n	c	o	u	r	s	c	o	n
2	14	13	2	14	20	17	18	2	14	13	2	14	20	17	18	2	14	13
4	14	2	6	6	6	17	1	9	2	2	21	22	8	4	0	15	19	1
e	o	c	g	g	g	r	b	j	c	c	v	w	i	e	a	p	t	b

La quatrième ligne donne le code de la clé (décalage à appliquer), la cinquième ligne donne la somme de la deuxième et la quatrième (modulo 26), la sixième ligne donne le texte crypté.

17. Ecrire une fonction **codageVigenere** qui prend en paramètres une liste représentant le texte à crypter et une liste d'entiers donnant la clé servant au codage, et qui retourne une liste contenant le texte crypté.

On procède comme pour le codage de César mais le décalage varie en fonction de la position de la lettre : au lieu de faire tout le temps un décalage de d, on effectue pour la lettre en position i un décalage de c[i%len(c)].

```
def codageVigenere (t,c) :
   n, k = len(t), len(c)
   return [(t[i]+c[i%k]) % 26 for i in range(n)]
```

Afin d'essayer de deviner le texte original sans la clé (de « cracker » le code), on procède en deux temps. D'abord on détermine la longueur k de la clé c, ensuite on détermine les lettres composant c.

La première étape est la plus difficile. On remarque que deux lettres identiques dans t, le texte crypté, espacées de $l \times k$ caractères (l entier) sont codées par la même lettre dans t. Cette condition n'est pas suffisante pour déterminer la longueur k de la clé c car des répétitions peuvent apparaître dans t sans qu'elles existent dans le texte original (les trois g consécutifs dans l'exemple ont pour origine trois lettres différentes). De plus, la même lettre dans le texte original peut être cryptée différemment dans t.

Nous allons rechercher des répétitions non pas d'une lettre mais de séquences de lettres dans t puisque deux séquences de lettres répétées dans le texte original, dont les premières lettres sont espacées par $l \times k$ caractères sont aussi cryptées par deux mêmes séquences dans t.

Dans la suite de l'énoncé, on ne considère que des séquences de taille 3 en supposant que toute répétition d'une séquence de trois lettres dans t provient exclusivement d'une séquence de trois lettres répétées dans le texte original. Ainsi, la distance séparant ces répétitions donne des multiples de k (i.e. les entiers l). La valeur de k est obtenue en prenant le PGCD de tous ces multiples. Si le nombre de répétitions est suffisant (i.e. si le texte est assez long), on a de bonnes chances d'obtenir la valeur de k. On suppose donc que cette assertion est vraie.

18. Ecrire une fonction **pgcd** qui calcule le PGCD de deux entiers positifs ou nuls passés en paramètres, et ce par soustractions successives.

```
def pgcd (a,b) :
   if a == 0 :
      return b
   elif b == 0 :
      return a
   while a != b :
      if a > b :
         a -= b
      else :
         b -= a
   return a
```

19. Ecrire une fonction **pgcdDistancesEntreRepetitions** qui prend en paramètres le texte crypté t de longueur n et un entier $i \in [0, n-3]$ qui est l'indice d'une lettre dans t. La fonction retourne le PGCD de toutes les distances entre les répétitions de la séquence de 3 lettres t[i], t[i+1], t[i+2] dans le texte t[i+3] t[i+4] ... t[n-1]. Cette fonction retourne 0 s'il n'y a pas de répétition.

On parcourt la liste à partir de l'indice i+3 pour trouver les occurrences de la séquence de lettres t[i], t[i+1], t[i+2]. Lorsqu'on en trouve une, on calcule sa distance à la précédente en utilisant la variable pos qui mémorise la dernière position trouvée.

```
def pgcdDistancesEntreRepetitions (t,i) :
   d = 0                  # initialisation du pgcd
   pos = i                # dernière position de la séquence cherchée
   n = len(t)
   for j in range(i+3,n-2) :
      if (t[i], t[i+1], t[i+2]) == (t[j], t[j+1], t[j+2]) :
         # j-pos est la nouvelle distance
         # on actualise la nouvelle valeur du pgcd
         d = pgcd(d,j-pos)
```

```
        pos = j          # j est la dernière position de la séquence
    return d
```

20. Ecrire une fonction **longueurCle** qui prend en paramètres le texte crypté t et qui retourne la longueur k probable de la clé de codage.

On utilise la fonction précédente pour créer une liste de tous les pgcd des distances entre les répétitions des séquences t[i], t[i+1], t[i+2] pour i allant de 0 à n – 6 où n est la longueur de t. Puis, on calcule le pgcd de tous les éléments de cette liste.

```
def longueurCle (t) :
    n = len(t)
    L = [pgcdDistancesEntreRepetitions(t,i) for i in range(n-5)]
    d = 0
    for k in L :
        if k != 0 :
            d = pgcd(k,d)
    return d
```

Notons que les hypothèses faites sur le texte pour que cet algorithme fonctionne sont assez fortes: le texte doit être suffisamment long, les répétitions de trois lettres ne doivent provenir dans le texte codé que d'une répétition dans le texte initial, qui doit donc en contenir. En pratique, il y a donc peu de chances que cet algorithme fonctionne correctement et retourne la bonne longueur de clé lorsqu'on le teste sur un petit texte choisi au hasard.

21. Donner le nombre maximal d'opérations réalisées par la fonction **longueurCle** en fonction de la longueur de t. On ne comptera que le nombre d'appels à la fonction **pgcd**.

On commence par compter le nombre d'appels à la fonction pgcd lors de l'exécution de pgcdDistancesEntreRepetitions. Dans le pire des cas, il y a un appel à chaque passage dans la boucle for, ce qui totalise n-i-1 appels.
Ensuite, pour construire la liste L dans la fonction longueurCle, les n-5 appels à pgcdDistancesEntreRepetitions engendrent en plus :

$$\sum_{i=0}^{n-6}(n-i-1) = (n-1)(n-5) - \frac{(n-6)(n-5)}{2}$$

appels à pgcd.
S'y ajoutent au pire (k !=0) autant d'appels direct à pgcd qu'il y a d'éléments dans L, c'est-à-dire n-5. Par conséquent, le nombre d'appels total à pgcd est au plus égal à :

$$\frac{(n-5)(n+6)}{2}$$

Il s'agit d'une complexité quadratique, cet algorithme n'est pas très efficace.

22. Une fois la longueur de la clé connue, donnez une idée d'algorithme permettant de retrouver chacune des lettres de la clé. Il s'agit de décrire rapidement l'algorithme et non d'écrire le programme.

Une fois la longueur de la clé connue, on peut faire du décodage de César automatique sur les parties de textes codées par la même lettre (pour la jème lettre, les lettres d'indice j-1,k+j-1,2k+j-1,...). On détermine ainsi les lettres de la clé par analyse des fréquences puis on décode.

23. *Question bonus.* Ecrire une fonction **`decodageVigenereAuto`** qui prend en paramètres le texte crypté t et qui retourne le texte original probable.

On utilise deux fonctions : une première qui permet de retrouver la clé et une seconde fonction qui décode.

```
def estimerCle (t) :
   k = longueurCle(t)
   cle = [0 for j in range(k)]
   q = len(t) // k
   # on crée un tableau M contenant le texte de sorte que les éléments d'une
   # même colonne soient codés avec le même décalage
   M = np.zeros((q+1,k))
   for i in range(q) :
      M[i,:] = [t[i*k+j] for j in range(k)]
   M[q,0:len(t)-q*k] = [t[q*k+j] for j in range(len(t)-q*k]
   for j in range(k) :
      # on determine la lettre de la colonne j de la clé
      L = frequenceLettres(list(M[:q,j]))
      m, max = 0, L[0]
      for i in range(1,len(L)) :
         if L[i] > max :
            m, max = i, L[i]
      cle[j] = (m-4) % 26
   return cle

def decodageVigenereAuto (t) :
   c = estimerCle(t)
   n, k = len(t), len(c)
   return [(t[i]-c[i%k]) % 26 for i in range(n)]
```

Partie C : Codage RSA

Le but de cette partie est de présenter le codage RSA (du nom de ses inventeurs Rivest, Shamir et Adleman). Ce système découvert dans les années 1970 est l'un des plus utilisés actuellement. C'est un codage asymétrique : la clé de codage et la clé de décodage sont différentes. La clé de codage est publique, tandis que la clé de décodage est privée et très difficile en pratique à obtenir à partir de la clé publique.

Le chiffrement RSA est le plus souvent utilisé pour communiquer des clés symétriques afin de poursuivre l'échange de donnée de façon confidentielle. Une personne A communique la clé publique à une personne B et garde la clé privée pour elle. La personne B envoie une clé symétrique cryptée à A avec le chiffrement RSA en utilisant la clé publique. La personne A décrypte la clé symétrique grâce à sa clé privée. Les deux personnes continuent leurs échanges avec les clés symétriques.

L'étape de création des clés est à la charge de A. Le renouvellement des clés n'intervient que si la clé privée est compromise, ou par précaution au bout d'un certain temps (qui peut se compter en années).

1. Choisir p et q, deux nombres premiers distincts
2. Calculer leur produit $n = p \times q$, appelé module de chiffrement
3. Calculer $\varphi(n) = (p-1)(q-1)$ (c'est la valeur de l'indicatrice d'Euler en n)
4. Choisir un entier naturel e premier avec φ(n) et strictement inférieur à φ(n), appelé exposant de chiffrement
5. Calculer l'entier naturel d, inverse de e modulo φ(n), et strictement inférieur à φ(n), appelé exposant de déchiffrement ; d peut se calculer efficacement par l'algorithme d'Euclide

Comme e est premier avec $\varphi(n)$, d'après le théorème de Bachet-Bézout il existe deux entiers d et k tels que $ed + k\varphi(n) = 1$, c'est-à-dire que $ed \equiv 1 \pmod{\varphi(n)}$: e est bien inversible modulo $\varphi(n)$.

Le couple (n,e) est la clé publique de chiffrement, alors que (n, d) est la clé privée.

Une valeur originale V est ainsi chiffrée en la valeur C par la formule : $C = V^e \bmod n$, et déchiffrée en la valeur V' par la formule : $V' = C^d \bmod n$.

Un exemple avec de petits nombres premiers (en pratique il faut de très grands nombres premiers afin que la factorisation de n soit trop longue à exécuter, et donc le code difficile à « cracker ») :

1. On choisit deux nombres premiers p = 3, q = 11
2. Leur produit $n = 3 \times 11 = 33$ est le module de chiffrement
3. $\varphi(n) = (3 - 1) \times (11 - 1) = 2 \times 10 = 20$
4. On choisit e = 3 (premier avec 20) comme exposant de chiffrement
5. L'exposant de déchiffrement est d = 7, l'inverse de 3 modulo 20 (en effet $ed = 3 \times 7 \equiv 1 \bmod 20$)

La clé publique est (n, e) = (33, 3), et la clé privée est (n, d) = (33, 7).

Lorsque la personne B veut transmettre un message à A (par exemple une clé symétrique pour simplifier les échanges futurs), on a par exemple l'envoi du message représenté par la valeur 4 (ex. caractère e ou le décalage d'un code de César) :

- Chiffrement de la valeur 4 par B avec la clé publique : $4^3 \equiv 31 \bmod 33$, la valeur envoyée à A par B est 31
- Déchiffrement de la valeur 31 par A avec la clé privée : $31^7 \equiv 4 \bmod 33$, A retrouve bien la valeur originale 4

C.1 Fonctions préliminaires

Vous supposerez connu les deux fonctions suivantes.

```
pgcd (a,b)
"""retourne le pgcd de a et b"""

euclide (a,b)
"""retourne le triplet r, u, v tels que au+bv=r par l'algorithme d'Euclide"""
```

24. Ecrire une fonction **`inverseModulaire`** prenant en paramètres deux entiers e et m et retournant l'entier d compris entre 1 et m-1 tel que $ed \equiv 1$ [m] lorsque e et m sont premiers entre eux et rien sinon.

La fonction euclide, appliquée à e et m premiers entre eux, nous fournit un entier u tel que $eu \equiv 1$ [m]. On veut la solution dans [0,m-1] donc on effectue la division euclidienne de u par m et on garde le reste.

```
def inverseModulaire (e,m) :
    r, u, v = euclide(e,m)
    if r == 1 :
        return u % m
```

25. Ecrire une fonction booléenne **`estPremier`** permettant de tester si un entier en paramètre est un nombre premier.

```
def estPremier (n) :
```

```
    i = 2
    while i**2 <= n :
        if n % i == 0 :
            return False
        i += 1
    return True
```

26. Ecrire une fonction **premierAleatoire** prenant en paramètres deux entiers inf et lg et choisissant aléatoirement des nombres dans [inf,inf+lg] jusqu'à ce qu'elle trouve un nombre premier, qu'elle doit retourner. En pratique, il faut l'appliquer avec lg suffisamment grand pour qu'il existe un nombre premier dans l'intervalle.

```
from random import randint
def premierAleatoire (inf,lg) :
    n = randint(inf,inf+lg)
    while not estPremier(n) :
        n = randint(inf,inf+lg)
    return n
```

27. Ecrire une fonction **premierAleatoireAvec** prenant en paramètre un entier n et tirant aléatoirement des nombres entiers compris entre 2 et n-1 jusqu'à ce qu'elle en trouve un premier avec n, qu'elle doit retourner. Rappel : vous avez la fonction **pgcd** à votre disposition.

```
def premierAleatoireAvec (n) :
    m = randint(2,n-1)
    while pgcd(m,n) != 1 :
        m = randint(2,n-1)
    return m
```

28. Ecrire une fonction **expoModulaire** qui calcule a^n modulo m lorsque a, n et m sont passés en paramètres. Vous écrirez cette fonction de manière récursive et en vous assurant que le nombre de multiplications effectuées est de l'ordre de $\log_2 n$ (et non de n).

```
def expoModulaire (a,n,m) :
    if n == 0 :
        return 1
    if n % 2 == 0 :
        return (expoModulaire(a,n//2,m)**2) % m
    return (a * expoModulaire(a,n//2,m)**2) % m
```

C.2 Codage et décodage RSA

29. Ecrire une fonction **choixCle** de paramètres deux nombres inf et lg, permettant de créer un triplet (p,q,e) tel que p et q sont deux nombres premiers distincts où $p \in [inf, inf + lg]$ et $q \in [p + 1, p + lg + 1]$, et e compris entre 2 et (p-1)(q-1) et e est premier avec (p-1)(q-1).

```
def choixCle (inf,lg) :
   p = premierAleatoire(inf,lg)
   q = premierAleatoire(p+1,lg)
   e = premierAleatoireAvec((p-1)*(q-1))
   return p, q, e
```

30. Ecrire une fonction **clePublique** permettant de calculer la clé publique (n,e) à partir de (p,q,e) et une fonction **clePrivee** permettant de calculer la clé privée (n,d) à partir de (p,q,e).

On se sert de la relation $n = p \times q$ et de l'inverse modulaire de e et phi(n).

```
def clePublique (p,q,e) :
   return p*q, e

def clePrivee (p,q,e) :
   n = p*q
   phi = (p-1)*(q-1)
   d = inverseModulaire(e,phi)
   return n, d
```

31. Ecrire une fonction **codageRSA** qui prend en paramètres un nombre M < n et la clé publique (n,e) et qui permet de coder M.

Pour coder un nombre strictement plus petit que n, il suffit de calculer ce nombre à la puissance e, modulo n.

```
def codageRSA (M,clePub) :
   n, e = clePub
   if M < n :
      return expoModulaire(M,e,n)
```

32. Ecrire une fonction **decodageRSA** qui prend en paramètres un nombre M < n et la clé privée (n,d) et qui permet de décoder M.

Pour décoder un nombre strictement plus petit que n, il suffit de calculer ce nombre à la puissance d, modulo n.

```
def decodageRSA (M,clePriv) :
   n, d = clePriv
   if M < n :
      return expoModulaire(M,d,n)
```

C'est la même fonction que pour coder, juste avec une clé différente en paramètre.