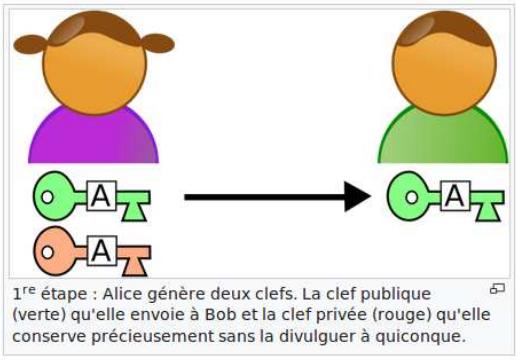
SPÉCIALITÉ NSI ET PROGRAMMATION PYTHON
Initiation à la cryptographie

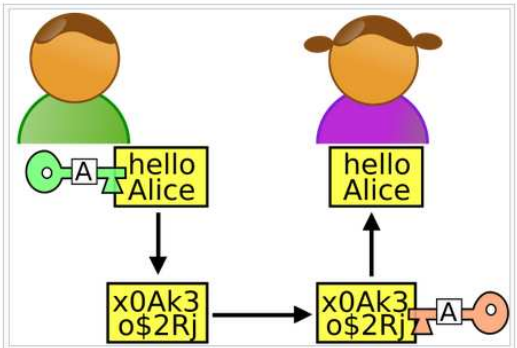
L'objectif de cette session d'AP est d'apercevoir au travers d'un TP sur plusieurs séances, quelques notions qui seront développées en première dans la spécialité NSI : Numérique et sciences informatiques.
J'ai choisi comme support le cryptosystème de Merkle-Hellman [1], pour lequel il faudra s'initier aux notions suivantes :

- la numération binaire, hexadécimale et encodage des caractères ;
- conception d'algorithmes de conversion de base ;
- programmation de ces algorithmes en Python ;
- notion de fonction en programmation ;
- les différents types de données en Python et manipulation de ces objets ;
- gestion des fichiers en Python ;

Merkle-Hellman est un cryptosystème asymétrique : la clé publique est utilisée pour le chiffrement, et la clé privée pour le déchiffrement.



1re étape : Alice génère deux clefs. La clef publique (verte) qu'elle envoie à Bob et la clef privée (rouge) qu'elle conserve précieusement sans la divulguer à quiconque.



2e et 3e étapes : Bob chiffre le message avec la clef publique d'Alice et envoie le texte chiffré. Alice déchiffre le message grâce à sa clef privée.

Nous travaillerons de manière progressive et guidée avec pour objectif de produire un programme Python générant des clefs privées et publiques, capable également de coder et décoder un message que vous vous échangerez à travers une messagerie.

1. https://fr.wikipedia.org/wiki/Cryptosyst%C3%A8me_de_Merkle-Hellman

I - Les systèmes de numération et l'encodage des caractères

«Un ordinateur est une machine qui manipule des valeurs numériques représentées sous forme binaire.»

1. Numération binaire

Dans l'écriture décimale des nombres, nous utilisons dix chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9 : si a, b et c sont des chiffres décimaux, le nombre abc_{10} est $a \times 10^2 + b \times 10^1 + c \times 10^0$.

Exemple 1
$207 = 2 \times 10^2 + 0 \times 10^1 + 7 \times 10^0$.

En binaire, nous n'utilisons que deux chiffres : 0 et 1 (ce qui convient parfaitement à l'informatique : les processeurs des ordinateurs sont composés de transistors ne gérant chacun que deux états) : les chiffres s'appellent les **bits** : si a, b et c sont des bits, le nombre abc_2 est $a \times 2^2 + b \times 2^1 + c \times 2^0$.

Exemple 2
$101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$.

Exercice 1

1. Écrire les huit premières puissances de 2 :

n	0	1	2	3	4	5	6	7
2^n								

2. Écrire en base 10, les nombres binaires 111_2, $1\,0000_2$ et $1111\,1111_2$.

3. Combien de nombres différents peut-on écrire avec 8 bits (c'est-à-dire un **octet**) ?

4. Écrire en base 2, les nombres 10_{10}, 17_{10} et 132_{10}.

Exercice 2

1. a) Compléter les divisions euclidiennes successives :

$$\begin{aligned} 37 &= \ldots \times 2 + \ldots \\ 18 &= \ldots \times 2 + \ldots \\ 9 &= \ldots \times 2 + \ldots \\ \ldots &= \ldots \times 2 + \ldots \\ \ldots &= \ldots \times 2 + \ldots \\ \ldots &= \ldots \times 2 + \ldots \end{aligned}$$

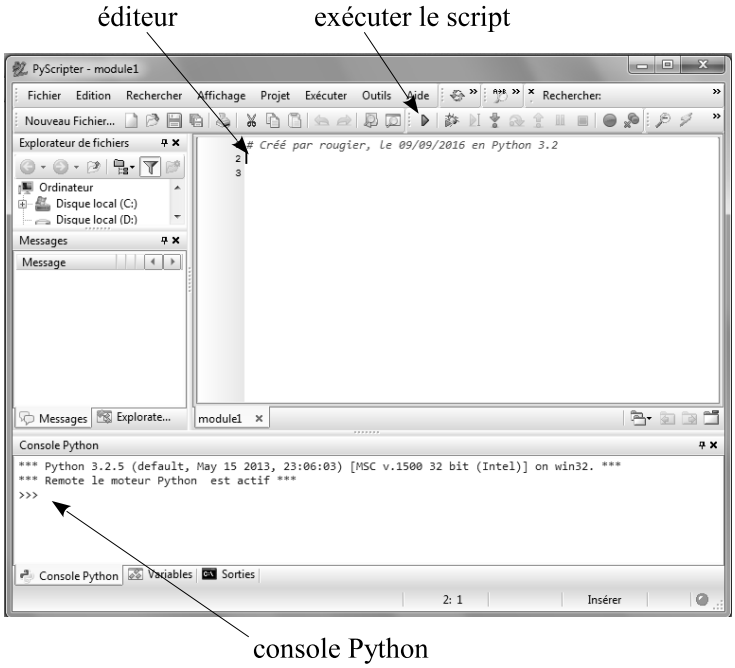
b) En déduire que $37 = ((((((2 \times 0 + 1) \times 2 + 0) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1$, puis l'écriture en base 2 de 37_{10}.

2. Reprendre le travail précédent pour écrire 147_{10}, puis 452_{10} en base 2.

3. En déduire un algorithme de conversion d'un nombre en écriture décimale en binaire.

2. Edupython : un environnement de programmation en Python pour le lycée

Vous trouverez cette distribution installé sur les ordinateurs du lycée. Il y a un raccourcis sur le bureau dans le dossier "Maths".



Vous écrirez des commandes pour faire des essais dans la console et des scripts (programme) dans l'éditeur. Vous exécuterez le programme en cliquant sur le petit triangle vert.

EXERCICE 3

Tester les instructions ou script suivant :

Dans la console :

```
>>> c = ""
>>> c = c + "A"
>>> c = c + "B"
>>> print(c)
```

Avec l'éditeur :

Script 1 – Premier script

```
## Un commentaire
mot = input("Saisir un mot : ")
print("voici le mot ", mot)
```

Remarque : Dans la plupart des autres langages, **un bloc d'instructions** doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que `begin` et `end`). En *C*, *Java* ou encore *Pascal*, par exemple, un bloc d'instructions doit être délimité par des accolades.

En C

```
for(int k = 0; k < 26; k++)
{
printf(k);
}
```

En Pascal

```
for k := 0 to 25 do
begin
write(k);
end;
```

Avec Python, vous devez utiliser les sauts à la ligne et l'indentation.

En contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. Tester les instructions suivantes dans la console :

En Python

```
>>> for k in range(26):
...     print(k)
...
```

3. Un algorithme de conversion, notion de liste et de fonction

Début
  Saisir n
  Tant que $n \neq 0$ **faire**
    Ajouter le reste de la division euclidienne de n par 2 à la liste L
    $n \leftarrow$ quotient entier de n par 2
  FinTantque
  Renverser la liste L
  Afficher l
Fin

Algorithme 1 : Conversion binaire

Script 2 – Conversion binaire

```
n = eval(input("Saisir un nombre : "))
L = []
while n != 0:
    L.append(n%2)
    n = n // 2
L.reverse()
print("Voici son écriture binaire : ", L)
```

Le script ci-dessus est une traduction de l'algorithme proposé en Python. Afin, de comprendre les différentes instructions utilisée, voici pour commencer une information trouvée dans la documentation Python :

Information
Python connaît différents types de données combinés, utilisés pour regrouper plusieurs valeurs. La plus souple est la **liste**, qui peut être écrite comme une suite, placée entre crochets, de valeurs (éléments) séparés par des virgules. Les éléments d'une liste ne sont pas obligatoirement tous du même type, bien qu'à l'usage ce soit souvent le cas.

EXERCICE 4

1. a) Saisir les instructions suivantes dans une console :

```
>>> n = input()
>>> print(n)
>>> type(n)
```

```
>>> n = eval(input("Hello : "))
>>> print(n)
>>> type(n)
```

b) Rédiger une synthèse résumant ce que vous avez compris concernant ces instructions.

2. a) Saisir les instructions suivantes dans une console :

```
>>> L = []
>>> type(L)
>>> L.append(1)
>>> print(L)
```

```
>>> L.append(0)
>>> print(L)
>>> L.reverse()
>>> print(L)
```

b) Rédiger une synthèse résumant ce que vous avez compris concernant ces instructions.

3. Quel est la fonction des opérateurs % et // en Python ?

L'excellent livre de Gérard SWINNEN [2] propose cette introduction à la notion de fonction :

Information
L'un des concepts les plus importants en programmation est celui de fonction. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable de calculer une racine carrée, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la réécrire à chaque fois.

2. Apprendre à programmer avec Python 3 : `http://inforef.be/swi/download/apprendre_python3.pdf`

a) Définir une fonction

Script 3 – Définition d'une fonction

```
def nomFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

Remarques :

- Comme les instructions **if** et **while** que vous connaissez déjà, l'instruction **def** est une *instruction composée*. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'*indenter*.
- La fin du corps, et donc de la définition de la fonction, est indiquée par l'apparition de lignes ayant la même indentation que l'en-tête.
- La liste de *paramètres* spécifie quelles informations il faudra fournir en guise d'*arguments* lorsque l'on voudra utiliser cette fonction (les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments).

b) Appeler une fonction

Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un appel de fonction est constitué du nom de la fonction suivi de parenthèses. Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés.

Script 4 – Appeler une fonction

```
nomFonction(liste d'arguments)
```

Très souvent, le corps d'une fonction contient une ou plusieurs instructions de la forme :

Script 5 – Fonction retournant un objet

```
def nomFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
    return objet
```

Dans ce cas, l'appel de la fonction, à l'endroit où il est écrit, représente cette valeur renvoyée ; cet appel prend plutôt la forme :

Script 6 – Appeler une fonction

```
resultat = nomFonction(liste d'arguments)
```

c) Des exemples

Script 7 – Table de 7

```
### Définition de la fonction ###
def table7():
    for n in range(1,11):
        print(n*7)

### Appel de la fonction ###
table7()
```

Script 8 – Table quelconque et liste

```
### Définition de la fonction ###
def table(base):
    t = []
    for n in range(1,11):
        t.append(n*base)
    return t

### Appel de la fonction ###
print(table(13))
```

EXERCICE 5

1. Tester les deux exemples précédents.

2. En utilisant le script 2, écrire une fonction nommée `binaire`, prenant comme paramètre un nombre entier et retournant les listes des bits de sont écriture en base 2.

3. Modifier la fonction `binaire` pour qu'elle retourne la liste des 8 bits de son écriture binaire si le nombre entier est inférieur à 256 et une liste vide sinon.
Il faudra éventuellement ajouter des 0...

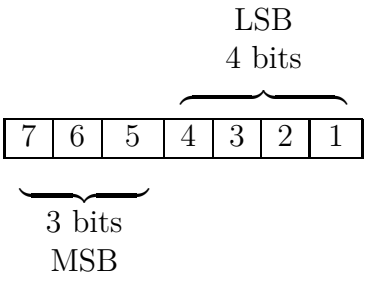
4. Encodage des caractères

Chaque information étant codée par un nombre en binaire dans un ordinateur et c'est en particulier le cas des caractères utilisés dans les langues du monde entier. Le système le plus largement utilisé aujourd'hui est UTF-8 :

UTF-8
UTF-8 (abréviation de l'anglais Universal Character Set Transformation Format1 - 8 bits) est un codage de caractères informatiques conçu pour coder l'ensemble des caractères du « répertoire universel de caractères codés », il reste compatible avec la norme ASCII limitée à l'anglais de base, mais très largement répandue depuis des décennies.

Voici un extrait de la table ASCII :

LSB \ MSB	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	}
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	{
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL



Exemple 3
Le code ASCII de la lettre Y est $101\,1001_2$ c'est-à-dire 89_{10}.

Le langage Python permet facilement de faire la correspondance entre les caractères et le nombre associé dans la table ASCII et plus généralement UTF-8 à l'aide des fonction `ord(caractère)` et `chr(nombre)`.

EXERCICE 6

1. Pour commencer et comprendre le fonctionnement de ces fonctions, tester les instructions suivantes :

```
>>> 0b1011001
>>> chr(89)
```

```
>>> ord('Y')
>>> print(chr(0b111))
```

```
>>> chr(22909)
```

2. Écrire une fonction nommée `encodage_phrase` prenant en entrée une chaîne de caractère et retournant la liste des codes associés à chaque lettre composant cette chaîne.

En utilisant successivement les deux fonctions `binaire` et `encodage_phrase`, nous pouvons maintenant encoder un texte en une liste de nombres écris en binaire.

Script 9 – Exercice 5 - Correction

```
def binaire(entier):
    liste_bits =[]
    if entier < 256:
        while entier != 0 :
            liste_bits.append(entier%2)
            entier = entier//2
        long = len(liste_bits)
        if long <= 8:
            for loop in range(8-long):
                liste_bits.append(0)
        liste_bits.reverse()
    return liste_bits
```

Script 10 – Exercice 6 - Correction

```
def encodage_phrase(texte):
    liste_codes = []
    for caractere in texte:
        liste_codes.append(ord(caractere))
    return liste_codes
```

5. Un petit résumé des différents types de données observés

a) Les entiers

Les entiers sont du type `int` :

```
>>> n = 17
>>> n//5
>>> n%5
```

b) Les nombres décimaux

Les nombres décimaux sont codés par des « nombres à virgule flottante » en Python du type `float` :

```
>>> x = 35e78
>>> x/1000
>>> 1/x
```

c) Les chaînes de caractères

Les chaînes de caractères sont du type `str`, c'est le type retourné par la fonction `input()` :

```
>>> c = input('Hello : ')
>>> c = c + ' beau message'
>>> print(c)
>>> len(c)
```

d) Les listes

Les listes peuvent contenir des éléments de type différent et sont du type `list` :

```
>>> L = [1, 3.45]
>>> L.append('mot')
>>> L.reverse()
>>> print(L)
>>> print(L[2])
>>> len(L)
```

II - Le principe de l'algorithme de Merkle-Hellman

1. Génération d'une clé privée

a) Choix d'une séquence super-croissante

Information

On dit qu'une séquence $\{a_1, a_2, ..., a_n\}$ avec $n \in \mathbb{N}$ et $n \geqslant 3$ est **super-croissante** lorsque pour tout entier $i \in \{3, ..., n\}$:

$$a_1 + a_2 + ... + a_{i-1} < a_i$$

Exercice 7

1. Vérifier que la séquence $\{2, 7, 11, 21, 42, 89, 180, 354\}$ est super-croissante.

2. Le script suivant permet de générer les deux premiers nombres, compris entre 1 et 50, d'une séquence super-croissante. Le compléter afin de générer une séquence super-croissante de 8 valeurs.

Script 11 – Séquence super-croissante

```
from random import *

def super_croissante():
    n = 50
    a1 = randint(1,n)
    a2 = randint(1,n)
    while a2 == a1:
        a2 = randint(1,n)
    if a1 < a2:
        sequence = [a1,a2]
    else:
        sequence = [a2,a1]
```

b) Clé privée

Information
Une **clé privée** pour cet algorithme est une liste :

$$(N, A, \{a_1, a_2, ..., a_8\})$$

constituée d'une séquence super-croissante $\{a_1, a_2, ..., a_8\}$ et de deux entiers N et A tels que :

$$N > a_1 + a_2 + ... + a_8 \qquad \text{et} \qquad \text{pgcd}(A, N) = 1 \text{ avec } 1<A<N$$

où **pgcd(A, N)** est le plus grand diviseur commun à A et N.
Dans le cas, où le pgcd vaut 1, on dit que les nombres A et N sont **premiers entre eux**.

Exercice 8

1. Vérifier que les nombres $N = 881$ et $A = 588$ conviennent avec la séquence proposée dans l'exercice précédent.

2. Écrire une fonction nommée `cle1` prenant comme paramètre une liste super-croissante et retournant un nombre N vérifiant la condition décrite ci-dessus.

3. La fonction `pgcd` ci-dessous retourne le pgcd des entiers a et b, avec $a \leqslant b$, donnés en paramètres.

Script 12 – PGCD

```
def pgcd(a,b):
    if a == 0:
        return b
    else:
        return pgcd(b%a,a)
```

Cette fonction est une fonction **récursive**. La tester à la « main » pour $a = 12$ et $b = 18$.

4. Utiliser la fonction `pgcd`, pour écrire une fonction nommée `cle2` prenant comme paramètre un entier N et retournant un nombre A aléatoire, vérifiant les conditions décrites ci-dessus.

5. Écrire pour terminer, une fonction nommée `cle_privee`, utilisant les différentes fonctions définies précédemment et générant une clé privée pour l'algorithme de Merckle-Hellman.

Script 13 – Exercice 7 - Correction

```
def super_croissante():
    n = 50
    a1 = randint(1,n)
    a2 = randint(1,n)
    while a2 == a1:
        a2 = randint(1,n)
    if a1 < a2:
        sequence = [a1,a2]
    else:
        sequence = [a2,a1]
    somme = a1+a2
    for i in range(6):
        a = somme + randint(1,n)
        sequence.append(a)
        somme = somme + a
    return sequence
```

Script 14 – Exercice 8 - Correction

```
def cle1(sequence):
    somme = 0
    for valeur in sequence:
        somme = somme + valeur
    return somme+randint(1,50)

def cle2(N):
    A = randint(2,N)
    while pgcd(A,N)!=1:
        A = randint(2,N)
    return A

def cle_privee():
    sequence = super_croissante()
    N = cle1(sequence)
    A = cle2(N)
    return [N, A, sequence]
```

2. Génération d'une clé publique

Information
Une **clé publique** pour cet algorithme va se générer à l'aide de la clé privée $\{N, A, \{a_1, a_2, ..., a_8\}\}$, on calcule le reste b_i dans la division euclidienne par N des nombres Aa_i où $i \in \{1, 2, ..., 8\}$.
La séquence $\{b_1, b_2, ..., b_8\}$ ainsi crée sera la clé publique servant à coder le message.

Exercice 9

1. a) Avec $N = 881$, $A = 588$ et $a_1 = 2$, on a $588 \times 2 = 1176$ et $1176 = 881 \times 1 + 295$.
Quelle est la valeur b_1 associée ?

b) Étant donnée la clé privée $\{881, 588, \{2, 7, 11, 21, 42, 89, 180, 354\}\}$, vérifier que la clé privée associée est $\{295, 592, 301, 14, 28, 353, 120, 236\}$.

2. Écrire une fonction `cle_publique` prenant pour argument une liste contenant une clé privée et retournant la liste contenant la séquence de la clé publique (on rappelle qu'en Python $a\%b$, donne le reste dans la division euclidienne de a par b).

Script 15 – Exercice 9 - Correction

```
def cle_publique(cle_pr):
    cle_pu = []
    A = cle_pr[1]
    N = cle_pr[0]
    for a in cle_pr[2]:
        cle_pu.append((A*a)%N)
    return cle_pu
```

3. Chiffrement

Voilà, Alice a maintenant créé ses deux clés : la clé privée qu'elle garde secrètement et la clé publique qu'elle donne à Bob pour qu'il code le message secret.

Information
Le procédé de chiffrement est le suivant : étant donné un caractère du message à crypter codé en binaire sur 8 bits : $\{w_1, w_2, ..., w_8\}$, on lui associe le nombre obtenu par le calcul suivant :

$$c = w_1 \times b_1 + w_2 \times b_2 + ... + w_8 \times b_8$$

Exercice 10

1. Montrer que le nombre c obtenu par ce procédé pour la lettre « a » est $c = 1129$.
2. Écrire une fonction `codage_lettre` prenant pour paramètres une liste contenant la clé publique et une variable contenant la lettre à coder, et retournant le nombre associé par ce procédé (vous utiliserez la fonction `binaire()` déjà programmée).
3. Écrire une fonction `codage` prenant pour paramètres une liste contenant la clé publique et une variable contenant le texte à coder, et retournant la liste contenant les nombre associés à chacun des caractères du texte (cette fonction utilisera la fonction `codage_lettre`.

Script 16 – Exercice 10 - Correction

```
def codage_lettre(cle_pu, lettre):
    w = binaire(ord(lettre))
    c = 0
    i = 0
    for b in cle_pu:
        c = c + w[i]*b
        i = i +1
    return c
```

Script 17 – Exercice 10 - Correction

```
def codage(cle_pu, texte):
    c = []
    for lettre in texte:
        c.append(codage_lettre(cle_pu, lettre))
    return c
```

4. Déchiffrement

Maintenant Bob a envoyé le message codé à Alice qu'elle va devoir décoder avec sa clé privée, deux étapes vont êtres nécessaires.

a) Première étape

Rappel
On dit qu'un nombre B est l'inverse du nombre A lorsque $A \times B = 1$ et dans ce cas on note $B = A^{-1}$.

Remarque : On dira également que B est l'inverse de A « modulo N » lorsque le reste de la division euclidienne de $A \times B$ par N est 1.

Propriété - *Admise*
Lorsque deux entiers A et N sont premiers entre eux alors l'entier A admet un unique inverse « modulo N », on le notera A^{-1}.

Information
Pour chacun des nombres c du code, on calcule le reste p dans la division euclidienne de $A^{-1} \times c$ par N.

Exercice 11

1. Montrer que le nombre $B = 442$ est l'inverse du nombre $A = 588$ « modulo $N = 881$ ».
2. Vérifier que le nombre p associé au nombre $c = 1129$ est $p = 372$.
3. Écrire une fonction `inverse` prenant pour argument un entier A et un entier N.
4. Écrire une fonction `decodage1` prenant pour paramètres la liste contenant les nombres $\{c_1, c_2, ..., c_8\}$ obtenue par le procédé de codage, ainsi que les nombres N et A de la clé privée et retournant la liste $\{p_1, p_2, ..., p_8\}$ des nombres obtenus par le procédé ci-dessus.

Script 18 – Exercice 11 - Correction

```
def inverse(A, N):
    B = 2
    while (B*A)%N != 1:
        B = B + 1
    return B
```

Script 19 – Exercice 11 - Correction

```
def decodage1(liste_c, N, A):
    B = inverse(A,N)
    p = []
    for c in liste_c:
        p.append((B*c)%N)
    return p
```

b) Deuxième étape : un algorithme glouton

En informatique, un algorithme glouton[3] (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local.

Exemple

Suivant le système de pièces, l'algorithme glouton est optimal ou pas. Dans le système de pièces européen (en centimes : 1, 2, 5, 10, 20, 50, 100, 200), où l'algorithme glouton donne la somme suivante pour 37 : 20 + 10 + 5 + 2, on peut montrer que l'algorithme glouton donne toujours une solution optimale.

Information

Pour chaque valeur de p obtenue à la première étape, on détermine maintenant, avec un algorithme glouton, la séquences $\{x_1, x_2, ..., x_8\}$ telle que :

$$p = x_1a_1 + x_2a_2 + ... + x_8a_8$$

où $\{a_1, a_2, ..., a_8\}$ est la séquence de la clé privée.
On sélectionne le a_i le plus grand qui est inférieur ou égale à p, puis on recommence avec $p - a_i$ jusqu'à obtenir 0.

Exercice 12

1. Vérifier que la valeur $p = 372$ conduit à la séquence $\{0, 1, 1, 0, 0, 0, 0, 1\}$ qui correspond à la lettre « a ».
2. Écrire une fonction `decimal` prenant pour paramètre une liste de 8 bits et retournant le nombre écrit en base 10.
3. Écrire la fonction `decodage` prenant pour paramètres la liste $\{c_1, c_2, ..., c_8\}$ envoyée par Bob ainsi que la clé privée et retournant le texte de Bob (on utilisera la fonction `decodage1` pour obtenir la liste $\{p_1, p_2, ..., p_8\}$).

3. https://fr.wikipedia.org/wiki/Algorithme_glouton

Script 20 – Exercice 12 - Correction

```
def decimal(nombre_bin):
    nombre = 0
    n = 7
    for b in nombre_bin:
        nombre = nombre + b*2**n
        n = n - 1
    return nombre
```

Script 21 – Exercice 12 - Correction

```
def decodage(liste_c, cle_pr):
    liste_p = decodage1(liste_c, cle_pr[0], cle_pr[1])
    cle = cle_pr[2]
    texte = ''
    for p in liste_p:
        x = [0, 0, 0, 0, 0, 0, 0, 0]
        rang = 7
        while p != 0:
            if p >= cle[rang]:
                p = p - cle[rang]
                x[rang] = 1
            rang = rang - 1
        texte = texte + chr(decimal(x))
    return texte
```