Préambule

Le but de ce TP est d'écrire une implémentation en Python du protocole de Merkle-Hellman puis des attaques de ce protocole. Vous pouvez utiliser les notes du TD 2. L'énoncé et le code sont sur **Moodle**. Des fonctions utilitaires sont fournies. Les fonctions essentielles sont esquissées : **ne modifiez pas leur prototype.** Les parties qui restent à écrire sont balisées par des lignes "TODO TODO TODO". Il est fortement recommandé de tester unitairement les fonctions sur des entrées judicieusement choisies au fur et à mesure qu'on les écrit. Les tests demandés explicitement par l'énoncé doivent être regroupés dans le module `experiments.py`. Les réponses aux questions théoriques doivent être rédigées sur une copie qu'on rendra à la fin de la séance. Les questions étiquetées avec une **(*)** sont plus difficiles ou proposées en bonus.

Modalités : à rendre, par binôme, par email, à la fin de la séance. Selon l'encadrant :

— thibault.hilaire@u-bordeaux.fr
— pierre.ramet@u-bordeaux.fr
— joachim.rosseel@inria.fr
— geraud.senizergues@u-bordeaux.fr

1. Protocole de Merkle-Hellman

1.1 Problème du sac à dos

On s'intéresse au module `merkle.py`. Le cryptosystème (feuille TD 2, exercice 1) repose sur le problème SAC À DOS :

entrée : Une suite ln de n nombres entiers $\ln[1], \ldots, \ln[i], \ldots, \ln[n]$ et un nombre entier s.
sortie : Une suite lb de n entiers dans $\{0, 1\}$, telle que : $\sum_{i=1}^{n} \text{lb}[i] \cdot \ln[i] = s$
NON s'il n'en existe pas.

Comme le problème SUBSETSUM (somme de sous-ensembles) est *NP*-complet, on ne peut espérer résoudre SAC À DOS en temps polynomial.

La suite ln est *super croissante* si pour tout $i \in [2, n]$, $\sum_{j=1}^{i-1} a_j < a_i$.

Q1 Ecrire une fonction *greedy_ solution(sakado :list,s :int)* qui prend en entrée un couple $(\ln, s)$ et renvoie une solution correcte au problème du sac à dos lorsqu'il est super-croissant.

Par exemple : `greedy_solution([2, 5, 9, 30, 50, 100], 135)` renvoie `010101`

1.2 Le protocole

Q2 Compléter la méthode *generate_ keys(self).*

Q3 Compléter la méthode *encrypt(self,message).*

Q4 Compléter la méthode *decrypt(self,cipher).*

Q5 Tester votre implémentation sur au moins 3 exemples.

Dans la suite, nous allons considérer une attaque de ce cryptosystème qui utilise l'algorithme LLL, proposé par Lenstra, Lenstra et Lovacz en 1982. Cet algorithme célèbre de réduction de réseaux euclidiens est, lui-même, basé sur le processus d'orthogonalisation de Gram-Schmidt. On s'intéressera en particulier à la variante de Lagarias-Odlyzko.

2. Orthogonalisation de Gram-Schmidt

On s'intéresse au module `gramschmidt.py`. Soit $b = (b_1, \ldots, b_i, \ldots, b_n)$ une base de $\mathbb{R}^n$, espace vectoriel sur $\mathbb{R}$, qui est muni du produit scalaire standard. Pour tout $i \in [1, n]$, notons E_i le sous-espace vectoriel engendré par $\{b_1, \ldots, b_j, b_{j+1}, \ldots, b_i\}$ et $E_0 = \{\vec{0}\}$. Définissons b_i^* comme :

$$b_i^* := b_i - pr_{E_{i-1}}(b_i),$$

où pr_{E_i} dénote la projection orthogonale sur le sous-espace E_i et la famille b^* comme :

$$b^* := (b_1^*, \ldots, b_i^*, \ldots, b_n^*) \tag{1}$$

Q6 Montrer que, pour tout $i \in [1, n]$, $(b_1^*, \ldots, b_i^*)$ est une base orthogonale de E_i.
Q7 Tester la fonction `gs` sur 3 exemples simples.

3. Réduction de bases de réseaux

On s'intéresse au module `lll.py`. Soit $b = (b_1, \ldots, b_i, \ldots, b_n)$ une base de $\mathbb{R}^n$, espace vectoriel sur $\mathbb{R}$. On note

$$L(b) := \{\sum_{i=1}^{n} x_i \cdot b_i \mid x_1, \ldots, x_i, \ldots x_n \in \mathbb{Z}\}$$

C'est le *réseau* euclidien engendré par b. On note $min(L) := \min\{x \cdot x \mid x \in L \setminus \{\vec{0}\}\}$. C'est la plus petite norme au carré d'un vecteur de L.
L'algorithme LLL, implémenté par la fonction *LLL(binit :array)*, prend en entrée une base b de $\mathbb{R}^n$ et fournit en sortie une nouvelle base (que nous noterons encore b) qui vérifie les propriétés suivantes : pour tous $1 \leq j < i \leq n$

$$|m_{i,j}| \leq 1/2 \tag{2}$$

$$\|b_i^* + m_{i,i-1} b_{i-1}^*\|^2 \geq 3/4 \cdot \|b_{i-1}^*\|^2. \tag{3}$$

où le coefficient $m_{i,j}$ est défini par :

$$m_{i,j} := \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$$

La propriété (**??**) est notée $IP(i, j)$ et (**??**) est notée $IL(i)$.

Q8 Montrer que cet algorithme fournit une base du réseau $L(b)$ engendré par les lignes de b.
Q9 (*) Montrer que, lorsque l'exécution passe par l'entrée de la boucle `while`, les propriétés suivantes sont vraies :

$$\forall i, \forall j, \;\; 1 \leq j < i \leq k-1 \;\Rightarrow\; IP(i, j) \text{ et } IL(i)$$

Note : on admettra que cet algorithme termine en temps polynomial et que la base est formée de vecteurs de norme "petite" par rapport à $min(L)$.
Q10 Tester la fonction *LLL* sur 3 exemples simples. En particulier, comparer avec les mêmes exemples que ceux utilisés pour la question Q7. Que remarquez-vous ? Justifiez.

4. Attaques du protocole de Merkle-Hellman

4.1 Résolution du problème du sac à dos via les réseaux

On s'intéresse au module `attacks.py`. Le but de cette partie est de réduire le problème du sac à dos à une recherche de court vecteur dans un réseau euclidien. L'idée de Lagarias-Odlyzenko est la suivante :
soit *sac* une suite de n entiers et s un entier-cible. On note $sac[i]$ le i-ième élément de la suite *sac*, pour $1 \leq i \leq n$. On définit pour tout entier K une suite sv de $n+1$ vecteurs à $n+2$ composantes par :

$$sv[i] = (0, \ldots, 0, 1, \ldots, 0, K \cdot sac[i]) \quad \text{pour } 1 \leq i \leq n \tag{4}$$

où la seule composante 1 se trouve en position i et

$$sv[n+1] = (0, \ldots, 0, \ldots, 0, 1, -K \cdot s) \tag{5}$$

i.e. toutes les composantes sont nulles, exceptées $sv[n+1, n+1] = 1$ et $sv[n+1, n+2] = -K \cdot s$.
Q11 Montrer que, si les ϵ_i sont des entiers dans $\{0, 1\}$ et si $\sum_{i=1}^{n} \epsilon_i \cdot sac[i] = s$ alors le vecteur

$$x = (\sum_{i=1}^{n} \epsilon_i \cdot sv[i]) + sv[n+1]$$

est un vecteur de $L(sv)$ (le réseau engendré par les lignes de sv), de norme au carré $\leq n+1$.

On choisit désormais un nombre $K > n+1$.
Q12 Montrer que réciproquement, si $x = (\sum_{i=1}^{n} x_i \cdot sv[i]) + x_{n+1} \cdot sv[n+1]$ est défini par des coordonnées $x_i \in \mathbb{Z}$ et vérifie $x \cdot x \leq n+1$, alors $\sum_{i=1}^{n} x_i \cdot sac[i] = x_{n+1} \cdot s$.
Q13 Compléter la fonction *sakado2base(sakado :list,s :int)* de façon qu'elle transforme un problème de sac à dos de longueur n en la matrice de $\mathbb{R}^{(n+1)\times(n+2)}$ décrite en (**??**,**??**).
Q13 Tester la fonction *lll_ solution(sakado :list,s :int)* sur dix exemples de sac à dos. On pourra utiliser les fonctions de génération aléatoire du module `experiments.py` pour tester des sacs de longueur 20, avec des entiers de taille moyenne 30 (en nombre de bits).

4.2 Attaques du protocole de Merkle-Hellman

Q14 Déduire, de la réduction de Lagarias-Odlyzenko, une attaque du protocole de Merkle-Hellman fondée sur l'algorithme LLL. La tester.
Q15 (*) Compléter la fonction *bf_ sol(sakado :list,debut :int,fin :int,s :int)* en écrivant une solution brute-force du problème du sac à dos.
Q16 (*) En déduire une attaque brute-force du protocole de Merkle-Hellman et tester pour des messages de 3 caractères (chacun représentable sur 8 bits). Comparer avec l'attaque utilisant la réduction de Lagarias-Odlyzenko.