Travaux Pratiques
Séance n° 2

Dans ce TP on va s'intéresser à des algorithmes d'exponentiation rapide. On travaillera avec le groupe $(\mathbb{Z}/p\mathbb{Z})^*$, et on prend pour p le nombre premier suivant (recommandé dans les standards dans les années 2000) :

$$p = 2^{1536} - 2^{1472} - 1 + 2^{64}\left(\lfloor 2^{1406}\pi \rfloor + 741804\right).$$

Premiers algorithmes.

1. Vérifier que p est un nombre premier, et que $p-1$ est deux fois un nombre premier.
2. Écrire une fonction `exp_LR(x,n)` qui calcule x^n avec l'algorithme d'exponentiation rapide de gauche à droite.
   Pour manipuler les bits de n, on pourra utiliser `n.binary()` ou `n.digits(2)`.
3. Tester votre fonction avec des éléments de $\mathbb{Z}/p\mathbb{Z}$. On utilisera les commandes `A = Zmod(p)` pour créer l'anneau $\mathbb{Z}/p\mathbb{Z}$, et `A.random_element()` pour choisir des éléments au hasard.
4. Certains protocoles requièrent des calculs de la forme $x^m y^n \bmod p$. Plutôt que de calculer x^m puis y^n puis leur produit, on peut adapter l'algorithme d'exponentiation de gauche à droite. L'idée est de précalculer le produit xy, puis à chaque étape, après la mise au carré, de multiplier par x, y, xy ou rien selon que les bits correspondants de m et n valent 1 et 0, 0 et 1, 1 et 1, ou 0 et 0.
   Par exemple, pour calculer $x^m y^n$ avec $m = 21 = 10101_2$ et $n = 12 = 01100_2$, on procèdera ainsi :

$$1 \xrightarrow{\wedge 2} 1 \xrightarrow{\times x} x \quad \xrightarrow{\wedge 2} x^2 \xrightarrow{\times y} x^2 y \quad \xrightarrow{\wedge 2} x^4 y^2 \xrightarrow{\times xy} x^5 y^3 \quad \xrightarrow{\wedge 2} x^{10} y^6 \xrightarrow{\wedge 2} x^{20} y^{12} \xrightarrow{\times x} x^{21} y^{12}.$$

   (a) Programmer (et tester !) cet algorithme. Faire bien attention au fait que m et n peuvent ne pas avoir le même nombre de bits.
   (b) Quel est l'intérêt de cette méthode par rapport au calcul en deux étapes (x^m puis y^n puis produit) ?
   (c) Peut-on utiliser ce principe avec un algorithme d'exponentiation rapide de droite à gauche ?

Attaque par chronométrage.

En cryptographie l'exposant n utilisé est souvent secret. Le problème avec l'algorithme d'exponentiation rapide est qu'il laisse fuiter de l'information sur n, comme on va le voir dans les questions suivantes.

On rappelle que le *poids de Hamming* d'un entier est le nombre de 1 dans son écriture en binaire ; ainsi, le poids de Hamming de 127 est 7, tandis que celui de 129 est 2.

5. Écrire une fonction `random_HW(N,w)` qui renvoie un entier aléatoire de N bits (c'est-à-dire compris entre 2^{N-1} et $2^N - 1$) et de poids de Hamming w.
   On pourra commencer par créer la liste des bits du résultat, et utiliser la commande `randrange(N)` pour tirer des entiers entre 0 et $N-1$.

6. Exécuter le code suivant. Que fait-il ? Que peut-on observer ?

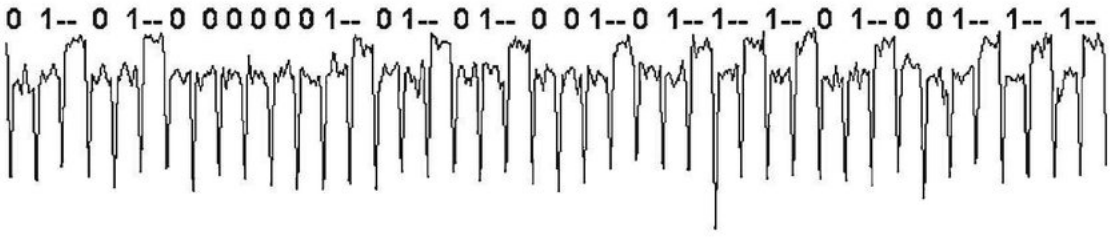
```
from time import time
L=[A.random_element() for _ in range(10)]
Timings=[]
for w in range(1,1535,25):
      tw=0
      for _ in range(10):
           e=random_HW(1535,w)
           t0=time()
           for x in L:
                exp_LR(x,e)
           dt=time()-t0
           tw+=dt
      Timings.append((w,tw))

list_plot(Timings)
```

7. Tirer un entier aléatoire secret avec les commandes
   `poids=randrange(1,1536); secret=random_HW(1535,poids)`.
   En mesurant des temps d'exécution de `exp_LR(x,secret)`, essayer de retrouver une valeur approximative du poids de Hamming de l'entier secret. On pourra effectuer une régression linéaire à partir des données de la question précédente avec les commandes
   `var('alpha,beta,t')`
   `model(t)=alpha*t+beta`
   `find_fit(Timings,model)`

Une contre-mesure.

Avec des mesures plus poussées que des simples chronométrages, on est capable d'obtenir beaucoup plus d'information que le poids de Hamming de l'exposant secret. Voilà par exemple une mesure de la consommation électrique d'un composant réalisant l'algorithme d'exponentiation rapide :



Un attaquant qui aurait accès à cet information (on parle d'attaques par canaux auxilliaires ou SCA – Side Channel Analysis) peut aisément retrouver l'exposant secret.

Une contre-mesure est d'utiliser un algorithme qui fait les mêmes opérations quelle que soit la valeur du bit. On présente ci-dessous la méthode de l'*échelle de Montgomery*, qui est surtout utilisée pour des courbes elliptiques.

Pour calculer x^n avec $n = (b_k b_{k-1} \dots b_1 b_0)_2$:
— initialiser deux variables a, b avec 1 et x

- parcourir les bits de n de gauche à droite ; à la i-ème étape,
  - si $b_{k-i} = 0$, remplacer a, b par $a^2, a.b$
  - si $b_{k-i} = 1$, remplacer a, b par $a.b, b^2$
- renvoyer a.

8. Implémenter (et tester !) cet algorithme. Démontrer qu'il renvoie bien le résultat attendu.
9. En reprenant le code de la question 6, vérifier que le temps de calcul ne dépend plus du poids de Hamming de l'exposant.