tp2-corrige

December 9, 2019

1 L3 Algèbre effective — TP 1 du mardi 1er octobre 2019

1.1 Factorisation d'entiers et cryptographie RSA

1.2 I-Factorisation d'entiers

La question de factorisation des entiers est au coeur de nombreux problèmes mathématiques. En sage, on dispose de la commande *factor*.

```
In [1]: factor(56)
```

```
Out[1]: 2^3 * 7
```

Le résultat est certes joli, mais en pratique on veut souvent réutiliser le résultat. Pour cela, on peut le convertir en liste.

```
In [2]: f=factor(56);
        list(f)
```

```
Out[2]: [(2, 3), (7, 1)]
```

Sage est très fort et sait factoriser rapidement de grands entiers. Faire quelques tests.

```
In [3]: factor(10^100+17)
```

```
Out[3]: 3^3 * 7 * 18617 * 25903 * 109718138717586334693746452264738879049551083759485526527666
```

Exercice. Pour voir que ce n'est pas si facile d'aller aussi vite, écrire vous même un algorithme de factorisation naïf (en testant les divisions successives).

N.B. Naïf ne veut pas dire qu'on ne peut pas être malin. Pensez notamment aux bornes des diviseurs à tester!

Pour alléger votre code, vous pouvez ne pas regrouper les facteurs multiples.

```
In [1]: def fact_aux(n,k): #fonction qui factorise n sachant qu'il n'a aucun facteur premier <
            i=k
            u=sqrt(n)
            while (i<=u):
                if (n%i)==0:
                    l=fact_aux(n//i,i)
```

```
                    if l[0][0]==i:
                        l[0][1]=l[0][1]+1
                        return l
                    else:
                        return [[i,1]]+l
                else:
                    i=i+1
            return [[n,1]]
```

```
In [2]: def factorise(n):
            if (n==1):
                return []
            else:
                return fact_aux(n,2)
```

Jusqu'à quelle taille d'entiers arrive t-on à factoriser?

C'est beaucoup plus lent que Sage, mais on arrive à factoriser quand la taille des facteurs premiers ne dépasse pas une dizaine de chiffres...

```
In [3]: factorise(10^9+17)
```

```
Out[3]: [[3, 2], [111111113, 1]]
```

2 II- Nombres premiers

Une question liée à la précédente est celle du test de primalité. On a vu en cours que le critère de Fermat permet de montrer que certains entiers ne sont pas premiers sans avoir à exhiber une factorisation explicite. On se propose de tester cette méthode.

Avant de commencer: à votre avis, quelle différence fondamentale y a t-il entre les deux calculs suivants qui explique la différence de temps d'exécution?

La première commande commence par calculer l'entier 254^{977821} **et seulement ensuite le réduit modulo** 977821**. Dans la seconde, on calcule les puissances succesives en réduisant à chaque fois modulo** 977821**.**

```
In [5]: time(254^977821 %977821)
```

```
CPU times: user 40 ms, sys: 0 ns, total: 40 ms
Wall time: 40.8 ms
```

```
Out[5]: 40697
```

```
In [6]: time(Mod(254,977821)^977821)
```

```
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 2.48 ms
```

```
Out[6]: 40697
```

```
In [0]:
```

Qu'en déduit-on sur 977821?

Ecrire une fonction *Fermat(a,n)* qui renvoie vrai si a vérifie la propriété de Fermat dans $\mathbf{Z}/n\mathbf{Z}$ et faux sinon.

```
In [4]: def Fermat(a,n):
            return (Mod(a,n)^n==Mod(a,n))
```

```
In [0]:
```

```
In [5]: Fermat(4,97)
```

```
Out[5]: True
```

On décide de procéder comme suit: pour décider si n est pseudo-premier, on tire au hasard 5 éléments de $\mathbf{Z}/n\mathbf{Z}$ et on teste s'ils vérifient la propriété de Fermat. S'ils passent tous le test, on le déclare pseudopremier (avec une certaine probabilité).

```
In [0]: randint?
```

```
In [9]: def Test_Fermat(n):
            for i in range(5):
                a=randint(2,n-1)
                if not Fermat(a,n):
                    #print(i)
                    return false
            return true
```

```
In [0]:
```

Faîtes quelques tests. Comparer par exemple la liste des nombres "pseudo-premiers"$\leq$ 1000 que vous trouvez ainsi avec la liste des nombres premiers que Sage peut vous donner.

```
In [16]: def prems(n):
             return [i for i in range(3,n) if Test_Fermat(i)]
```

```
In [31]: l=prems(1000);
```

```
In [0]:
```

```
In [23]: P=Primes()
```

```
In [32]: ll=[P[i] for i in range (200) if P[i]<1000];
```

On teste la différence entre nos deux listes.

```
In [29]: [i for i in l if i not in ll]
```

```
Out[29]: [561]
```

```
In [30]: factor(561)

Out[30]: 3 * 11 * 17
```

On a trouvé tous les nombres premiers, et seulement aussi 561 qui est le premier nombre de Carmichael.

N.B: Au début des années 2000, 3 étudiants indiens de votre âge ont montré qu'il existe un algorithme polynomial permettant de tester si un entier est premier ou non. *"Primes is in P."*

```
In [0]:
```

3 III- Fonction d'Euler

On note φ la fonction d'Euler. Ainsi $\varphi(n)$ compte le nombre d'éléments inversibles de $\mathbf{Z}/n\mathbf{Z}$.

Ecrire une fonction *phi* qui calcule cette valeur à partir d'une factorisation de n.

```
In [35]: def phi(n):
             f=list(factor(n))
             res=1
             for x in f:
                 res=res*(x[0]-1)*x[0]^(x[1]-1)
             return res

In [36]: phi(36)

Out[36]: 12
```

Parfois, on peut être amené à avoir besoin de la liste de tous les éléments de $(\mathbf{Z}/n\mathbf{Z})^\times$.

Ecrire une fonction *liste_inversibles(n)* qui renvoie la liste des inversibles de $\mathbf{Z}/n\mathbf{Z}$.

```
In [38]: def liste_inversibles(n):
             return [i for i in range(1,n) if gcd(i,n)==1]

In [39]: liste_inversibles(36)

Out[39]: [1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35]

In [40]: len(Out[39])

Out[40]: 12
```

4 IV- Cryptographie RSA

Pour coder un message avec RSA, il faut d'abord convenir d'une façon de représenter les messages par des nombres qu'on pourra ensuite crypter en utilisant RSA. Pour simplifier, on se limite à des messages n'utilisant que les lettres "a", "b",... "z" et " "(l'espace). Donc PAS DE MAJUSCULES, d'ACCENTS, etc...

On a ainsi 27 caractères, qu'on associe aux entiers de 0 à 26 comme suit: a=1, b=2, .., z=26, " "=0. Il est alors naturel d'encoder une chaîne de caractères par l'entier correspond en base 27. Ainsi, le mot "abc" est encodé par l'entier $1 \times 27^2 + 2 \times 27 + 3$.

En SAGE, une chaîne de caractères (*string*) est une liste de lettres auxquelles on peut accéder comme dans une liste.

```
In [11]: mot="a sjkg hdfjkg"

In [12]: mot[5], mot[1],mot[0]

Out[12]: ('g', ' ', 'a')
```

On peut renvoyer le code ascii de chaque lettre grâce à la fonction *ord()*.

```
In [12]: ord('a'), ord("b"), ord(' ')

Out[12]: (97, 98, 32)
```

On donne donc les fonctions *Lettreanombre()* et *nombrealettre()* suivantes.

```
In [43]: def lettreanombre(l):
             if l==' ':
                 return 0
             else:
                 return ord(l)-96

In [44]: lettreanombre('e'), lettreanombre("z")

Out[44]: (5, 26)

In [45]: def nombrealettre(n):
             if n==0:
                 return ' '
             else:
                 return chr(n+96)

In [46]: nombrealettre(0)

Out[46]: ' '
```

Ecrire une fonction *encode()* qui prend une chaîne de caractère et l'encode selon le principe précédent.

```
In [47]: def encode(l):
             i=0;
             n=0;
             while (i<len(l)):
                 n=lettreanombre(l[i])+27*n
                 i=i+1
             return n

In [48]: encode("abc")

Out[48]: 786

In [0]:
```

Ecrire une fonction *decode()* qui fait l'inverse. (N.B.:on peut concaténer des lettres grâce à +: 'a' +'b'="ab").

```
In [49]: def decode(n):
             a=n
             l="";
             while (a>0):
                 l=nombrealettre(a%27)+l;
                 a=a//27
             return l
```

On en vient à RSA proprement dit. Codez la phrase: "l algebre effective c est trop cool". On devrait trouver 556700255287378873824315538905643384473623616997 50 (il y a 50 chiffres).

```
In [51]: m=encode("l algebre effective c est trop cool"); m

Out[51]: 556700255287378873824315538905643384473623616997 50

In [44]: log(556700255287378873824315538905643384473623616997 50.,10)

Out[44]: 49.7456214204624522016925743933936888025739933537 3
```

Voici ma clé RSA que je donne à tout le monde: *cle* est le produit de deux nombres premiers que je tiens secret mais dont je donne le produit et *exp* est un exposant que j'ai choisi pour être premier avec $\varphi(cle)$.

```
In [52]: cle=515642158548951662411000000000000355793089398776647063 59

In [53]: exp=6753790093096013771
```

Envoyez moi la phrase précédente codée. Vous devriez trouver:'qixllymaugladgrhwmyeiakpqosmqdkhmfxsszp'

```
In [54]: u=Mod(m,cle)^exp

In [55]: decode(428099845047693418458999877152770840679039822452832560 41)

Out[55]: 'qixllymaugladgrhwmyeiakpqosmqdkhmfxsszp'

In [0]:
```

Je n'ai pas été très malin, ma clé n'a que 55 chiffres. C'est trop peu. Vous pouvez décoder les messages que vous interceptez en utilisant judicieusement l'ordinateur. Faîtes le avec le message suivant:

```
In [56]: message="fihhn kir merxmxlbzsqkyvcvpecldodleohxy"

In [57]: factor(cle)

Out[57]: 515642158548951662411 * 10000000000000000000000000000000069
```

Comme j'arrive à factoriser la clé, je n'ai qu'à calculer l'exposant de décodage, c'est à dire l'inverse de exp modulo $\varphi(cle)$.

```
In [59]: i=Mod(exp,(515642158548951662411-1)*(1000000000000000000000000000000069-1))^-1; i

Out[59]: 34230564750163176167625658058097972274217504066779809851
```

```
In [61]: mm=encode(message);
         decode(int(Mod(mm,cle)^i))

Out[61]: 'en plus les tp sont vraiment sympas'
```

À vous de vous créer une clé publique et un exposant. Il vous faut pour cela trouver 2 nombres premiers grands. Faîtes le.

```
In [18]: next_prime?
```

A partir de là, comment calculer $\varphi(n)$ et trouver un exposant premier à $\varphi(n)$?

```
In [63]: next_prime(16515614238753865435678915684646213846546+51)

Out[63]: 16515614238753865435678915684646213846701

In [0]:
```

Je me suis finalement rendu compte que vous lisiez mon courrier (au passage, ça vous fait moins 5 points!) et j'ai augmenté la taille de ma clé secrète (environ 200 chiffres). Du coup, je me suis dit que je pouvais économiser sur la taille de l'exposant (ça fait moins de calculs pour chiffrer un message).

```
In [0]:

In [0]: cle=3499999999999999999999999999999999999999999999999999999999999999999999999999943580545084

In [47]: exp=3
```

Montrez que ce n'est pas une bonne idée. Déchiffrez le message suivant (que vous conjecturez n'être pas trop long.)

```
In [64]: message='lcyjtiklnclxgnttwtiuilfgvpaloynzsaadvzdbmpogcjnzxn  qgczihpvlclpvaosnmcyymflg
```

L'idée est que si la taille du message élevé à la puissance exp est trop petite devant la taille de la clé, il se peut qu'il n'y ait pas de réduction modulo la clé. Ici en élevant au cube, on reste plus petit que la clé; Pour décoder, il suffit de calculer la racine cubique!

```
In [67]: (encode(message)^(1/3))

Out[67]: 4875244189423680332340448746799229553828991784

In [68]: decode(4875244189423680332340448746799229553828991784)

Out[68]: 'trop elementaire mon cher watson'

In [0]:
```