

TP-factorisation_RSA

September 30, 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 * 1097181387175863346937464522647388790495510837594
```

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 [4]: def factorise(n):  
        return []
```

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

```
In [ ]:
```

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?

```
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 [2]: def Fermat(a,n):
        return true
```

```
In [19]: Fermat(4,97)
```

```
Out[19]: 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 [ ]: randint?
```

```
In [7]: def pseudo_prem(n):
        return true
```

```
In [ ]:
```

Faites quelques tests. Comparer par exemple la liste des nombres "pseudo-premiers" ≤ 1000 que vous trouvez ainsi avec la liste des nombres premiers que Sage peut vous donner.

```
In [9]: P=Primes()
```

```
In [17]: [P[i] for i in range(200) if P[i]<1000];
```

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 [2]: def phi(n):  
        f=factor(n)  
        return 0
```

```
In [0]:
```

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 [1]: def liste_inversibles(n):  
        return []
```

```
In [0]:
```

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

```
In [36]: lettreatnombre('e'), lettreatnombre("z")
```

```
Out[36]: (5, 26)
```

```
In [25]: def nombrealettre(n):  
        if n==0:  
            return ' '  
        else:  
            return chr(n+96)
```

```
In [33]: nombrealettre(0)
```

```
Out[33]: ' '
```

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

```
In [34]: def encode(s):  
        return 0
```

```
In [0]: encode("abc")
```

```
In [0]:
```

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

```
In [39]: def decode(n):  
        return ""
```

On en vient à RSA proprement dit. Codez la phrase: "l algèbre effective c est trop cool". On devrait trouver 55670025528737887382431553890564338447362361699750 (il y a 50 chiffres).

```
In [ ]:
```

```
In [44]: log(55670025528737887382431553890564338447362361699750., 10)
```

```
Out[44]: 49.74562142046245220169257439339368880257399335373
```

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 [45]: cle=51564215854895166241100000000000035579308939877664706359
```

```
In [0]: exp=6753790093096013771
```

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

```
In [0]:
```

