L3 - Mathématiques pour l'informatique 4

Arithmétique modulaire (suite)

Exponentiation modulaire

La fonction `powermod` calcule efficacement $a^m \bmod n$ par le procédé suivant (voir [ici](https://fr.wikipedia.org/wiki/Exponentiation_modulaire) (https://fr.wikipedia.org/wiki/Exponentiation_modulaire)). On écrit (en pensée) l'exposant m en binaire $m = b_k \dots b_1 b_0$. Les bits b_i sont calculés au fur et à mesure. Au départ, le résultat intermédiaire `res` est 1. On calcule $b_0 = m \bmod 2$. Si $b_0 = 1$ on multiplie `res` par a modulo n . On divise m par 2, on remplace a par $a^2 \bmod n$, et on recommence. Ceci calcule

$$\bar{a}^m = (\bar{a})^{b_0} (\bar{a}^2)^{b_1} (\bar{a}^{2^2})^{b_2} \dots (\bar{a}^{2^k})^{b_k}$$

en remarquant que $(x^{2^i})^2 = x^{2^{i+1}}$ (programme de maths de 4ème ...).

On calcule les bits b_i au fur et à mesure. Au départ, $b_0 = m \bmod 2$ est `m % 2` ou `m & 1`. Si c'est 1, on multiplie le résultat `res` par a , sinon on ne fait rien. Puis on remplace a par $a^2 \bmod n$ et m par `m//2` ou `m>>1` et on recommence.

```
In [1]: def powermod(a,m,n):
        res = 1
        while m:
            if m & 1: res = (res*a) %n
            m >>= 1
            a = (a*a) %n
        return res
```

```
In [9]: powermod(3,33333,11)
```

```
Out[9]: 5
```

```
In [8]: powermod(5,666,2**101-1)
```

```
Out[8]: 2221376488713205668017410030004
```

Tests de primalité

Le théorème de Fermat permet de prouver qu'un nombre n n'est pas premier en exhibant un a tel que $a^{n-1} \not\equiv 1 \pmod n$. C'est la *test de Fermat*. On peut montrer que chaque fois qu'on obtient 1, la probabilité que n ne soit pas premier est *presque* divisée par 4. Malheureusement, il existe des nombres non premiers (les *nombre de Carmichael* (https://fr.wikipedia.org/wiki/Nombre_de_Carmichael)) qui passent tous les tests de Fermat avec $a \wedge n = 1$. Ils sont très rares, mais suffisent à faire capoter l'entreprise.

Le *test de Miller-Rabin* permet de remédier à ce problème. C'est une petite modification du calcul de $a^{n-1} \bmod n$ dans laquelle on rajoute un test.

L'idée est que dans un corps, l'équation $x^2 - 1 = 0$ ne peut pas avoir plus de deux solutions. Dans les entiers modulo n , on a toujours $\bar{1}$ et $-\bar{1}$, et $x^2 - \bar{1} = (x - \bar{1})(x + \bar{1})$. Si on en trouve une autre $\bar{b}$, alors $(\bar{b} - \bar{1})(\bar{b} + \bar{1}) = \bar{b}^2 - \bar{1} = \bar{0}$, donc on a des diviseurs de 0 et n n'est pas premier.

Pour cela, on écrit $n - 1 = 2^k m$ avec m impair, et on commence par calculer $b = a^m \bmod n$. Si $b = 1$, on renvoie True (probablement premier). Sinon, on calcule les $b^{(2^i)} \bmod n$ et la première fois que le résultat vaut 1, on teste si la valeur précédente était $\equiv -1 \pmod n$. Si ce n'est pas le cas, on renvoie False (composé). Si on arrive à $b^{(2^k)}$ sans avoir rencontré de racine carrée non triviale de 1, on renvoie True.

Par exemple, 561 est un nombre de Carmichael. Si on calcule $2^{560} \bmod 561$, on trouve 1. Mais si on décompose le calcul en écrivant

$$560 = 2^4 \cdot 35, \quad 2^{560} = (2^{35})^{2^4}$$

et en calculant d'abord $b = 2^{35} \bmod 561 = 263$, puis $b^2, b^4, b^8, b^{16} = 2^{560}$, on peut observer que $b^4 = 67$ et $(b^4)^2 = 1$. On a donc trouvé une racine carrée de 1 modulo 561 autre que ± 1 . Donc 561 n'est pas premier.

```
In [27]: from ent3 import *
         factor(560)
```

```
Out[27]: [(2, 4), (5, 1), (7, 1)]
```

```
In [28]: b = powermod(2,5*7,561)
```

```
In [29]: # 2**9 = 512
         for i in range(5): print(powermod(b,2**i,561), end=' ')
         263 166 67 1 1
```

```
In [32]: powermod(67,2,561)
Out[32]: 1

In [33]: # en voilà un assez grand
c = 1590231231043178376951698401
for a in range(2,12): print (powermod(3,c-1,c), end= ' ')
1 1 1 1 1 1 1 1 1 1

In [34]: print (factor(c))
[(17, 1), (19, 1), (23, 1), (29, 1), (31, 1), (37, 1), (41, 1), (43, 1), (61, 1), (67, 1), (71, 1),
(73, 1), (79, 1), (97, 1), (113, 1), (199, 1)]

In [35]: miller_rabin(c)
Out[35]: False
```

Le théorème des restes chinois

Il permet de résoudre le système de congruences

$$\begin{cases} x \equiv x_1 \pmod{m_1} \\ x \equiv x_2 \pmod{m_2} \\ \vdots \quad \vdots \\ x \equiv x_n \pmod{m_n} \end{cases}$$

lorsque les modules m_i sont deux à deux premiers entre eux.

Pour chaque i , il est facile de trouver a_i qui vérifie $a_i \equiv 0 \pmod{m_j}$ pour tous les $j \neq i$. Il suffit de prendre $a_i = \prod_{j \neq i} m_j$. Ce nombre est premier avec m_i , donc inversible modulo m_i . Calculons les coefficients de Bézout tels que $1 = u_i a_i + v_i m_i$. Alors, $y_i = u_i a_i$ vérifie $y_i \equiv 1 \pmod{m_i}$ et $y_i \equiv 0 \pmod{m_j}$ pour $j \neq i$, ce qui entraîne que

$$X := y_1 x_1 + y_2 x_2 + \dots + y_n x_n$$

est solution du système. Il y en a d'autres : en ajoutant le produit $m = m_1 m_2 \dots m_n$ à une solution, on obtient une autre solution, et deux solutions quelconques x', x'' diffèrent forcément d'un multiple de m , puisque leur différence est divisible par tous les m_i .

La plus petite solution positive du système est donc

$$x = y_1 x_1 + y_2 x_2 + \dots + y_n x_n \pmod{m}.$$

Exemple Le problème des pirates (posé au CAPES de Maths)

Une bande de 17 pirates possède un trésor constitué de pièces d'or d'égale valeur. Ils projettent de se le partager également, et de donner le reste au cuisinier chinois. Celui-ci recevrait alors 3 pièces. Mais les pirates se querellent, et six d'entre eux sont tués. Un nouveau partage donnerait au cuisinier 4 pièces. Dans un naufrage ultérieur, seuls le trésor, six pirates et le cuisinier sont sauvés, et le partage donnerait alors 5 pièces d'or à ce dernier. Quelle est la fortune minimale que peut espérer le cuisinier s'il décide d'empoisonner le reste des pirates ?

On doit donc résoudre

$$\begin{cases} x \equiv 3 \pmod{17} \\ x \equiv 4 \pmod{11} \\ x \equiv 5 \pmod{6} \end{cases}$$

On cherche a_1 qui vérifie $a_1 \equiv 0 \pmod{11, 6}$. On prend $a_1 = 6 \times 11 = 66$, et on calcule son inverse modulo 17. Comme $66 = 4 \times 17 - 2 \equiv -2$ et $2 \times 9 = 18 \equiv 1$, on a (de tête) $(-2)(-9) \equiv 1$, l'inverse cherché est $-9 \equiv 8$. On a donc que $y_1 = 8 \times 66 = 528$ est congru à 1 modulo 17 et à 0 modulo 11 et 6.

On prend ensuite $a_2 = 17 \times 6 = 102$ qui est congru à 3 modulo 11. Son inverse est donc 4, et $y_2 = 4 \times 102 = 408$.

Finalement, $a_3 = 17 \times 11 = 187$ est congru à 1 modulo 6 donc $y_3 = 187$.

La plus petite solution positive sera

$$x = 3 \times 528 + 4 \times 408 + 5 \times 187 \pmod{17 \times 11 \times 6} = 4151 \pmod{1122} = 785.$$

Voir ici (https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_des_restes_chinois).

Autre exemple, avec $(x_1, x_2, x_3, x_4) = (1, 2, 3, 4)$ et $(m_1, m_2, m_3, m_4) = (1, 13, 17, 19)$, on trouve $x = 41823$:

```
solve_chinese([1,2,3,4], [11,13,17,19])
41823
```

```
In [48]: 17*11*6
Out[48]: 1122
```

In [49]: (3*528+4*408+5*187)

Out[49]: 4151

Quelques résultats de théorie des groupes

On connaît déjà quelques exemples de groupes finis : les $(\mathbb{Z}/n\mathbb{Z})^\times$, les éléments inversibles de $\mathbb{Z}/n\mathbb{Z}$, forment un groupe dont le nombre d'éléments est noté $\varphi(n)$ (indicateur d'Euler).

Le $\times$ en exposant signifie *groupe multiplicatif*, il est formé des éléments inversibles.

Par exemple,

$$(\mathbb{Z}/12\mathbb{Z})^\times = \{1, 5, 7, 11\}.$$

Le théorème de Lagrange

On appelle *ordre* d'un élément g d'un groupe fini G (noté multiplicativement, d'élément neutre 1), le plus petit entier positif m tel que $g^m = 1$.

Il existe : puisque G est fini, la suite g^k des puissances de g contient forcément deux éléments égaux, et si $g^k = g^l$, (\$k

On appelle *ordre* du groupe G son nombre d'éléments.

Ce choix judicieux des définitions permet d'énoncer élégamment le *théorème de Lagrange* :

Dans un groupe fini, l'ordre de tout élément est un diviseur de l'ordre du groupe.

En effet, prenons un élément h de G , d'ordre m dans un groupe d'ordre n . Alors, ou bien $m = n$, et c'est fini, ou bien \$m

Fermat

Si $G = (\mathbb{Z}/p\mathbb{Z})^\times$ avec p premier, $n = p - 1$, et donc tout élément vérifie $\bar{a}^{p-1} = \bar{1}$.

Euler

Si $G = (\mathbb{Z}/n\mathbb{Z})^\times$, d'ordre $\varphi(n)$, tout élément vérifie $\bar{a}^{\varphi(n)} = \bar{1}$.

Calcul de $\varphi(n)$

Si $n = p_1^{m_1} \dots p_r^{m_r}$ est la décomposition de n en facteurs premiers, le théorème des restes chinois montre que

$$\mathbb{Z}/n\mathbb{Z} \simeq \mathbb{Z}/p_1^{m_1}\mathbb{Z} \times \dots \times \mathbb{Z}/p_r^{m_r}\mathbb{Z}$$

les opérations s'effectuant composante à composante sur les r -uplets. Donc, un élément sera inversible si et seulement si ses résidus modulo tous les $p_i^{m_i}$ le sont.

Il suffit donc de savoir calculer $\varphi(p^m)$. Les éléments *non inversibles* de $\mathbb{Z}/p^m\mathbb{Z}$ sont les entiers \$0\$le a

Le produit de ces nombres peut s'écrire

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

(produit sur les diviseurs premiers de n).

On peut exprimer cette formule grace à la *fonction de Möbius* μ

$$\mu(n) = \begin{cases} 1 & \text{si } n = 0 \\ (-1)^r & \text{si } n = p_1 \dots p_r \text{ est produit de } r \text{ nombres premiers distincts} \\ 0 & \text{sinon} \end{cases}$$

En développant le produit, on a

$$\varphi(n) = \sum_{d|n} \mu(d) \frac{n}{d}.$$

Remarquons que $\varphi(n)$ est aussi le nombre d'éléments d'ordre (additif) n dans $\mathbb{Z}/n\mathbb{Z}$. En effet, $kx \equiv 0 \pmod n \Rightarrow n|k$ si et seulement si $x \wedge n = 1$.

Pour un diviseur d de n , $\varphi(d)$ est donc le nombre d'éléments d'ordre d dans $\mathbb{Z}/n\mathbb{Z}$, et on a

$$n = \sum_{d|n} \varphi(d).$$

L'équivalence des deux expressions est un cas particulier de la *formule d'inversion de Möbius* (https://fr.wikipedia.org/wiki/Fonction_de_M%C3%B6bius), qui relie deux fonctions f et g sur les entiers positifs

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d)g\left(\frac{n}{d}\right).$$

In [56]: factor(666)

Out[56]: [(2, 1), (3, 2), (37, 1)]

```
In [57]: 666*(1-1/2)*(1-1/3)*(1-1/37)
```

```
Out[57]: 216.00000000000003
```

Groupes cycliques

Un groupe formé des puissances d'un seul élément est dit *cyclique* (pourquoi, à votre avis ?).

Par exemple, le groupe *additif* $(\mathbb{Z}/n\mathbb{Z}, +)$ est cyclique : il est formé des "puissances additives" de $\bar{1}$.

Le théorème de Lagrange entraîne que *tout groupe fini d'ordre premier est cyclique*.

La propriété suivant est moins évidente : *Si p est premier, le groupe multiplicatif $G_p = (\mathbb{Z}/p\mathbb{Z})^\times$ est cyclique*.

Autrement dit, il existe (au moins) un élément g , appelé *générateur* ou *élément primitif* dont les puissances modulo p prennent toutes les valeurs de 1 à $p - 1$.

Par exemple, $g = 3$ est primitif modulo $p = 31$:

```
In [58]: p = 31
         for i in range(31): print (pow(3,i,31),end=' ')
1 3 9 27 19 26 16 17 20 29 25 13 8 24 10 30 28 22 4 12 5 15 14 11 2 6 18 23 7 21 1
```

On peut montrer que d'une manière générale, le groupe multiplicatif d'un corps fini est toujours cyclique.

Dans le cas des entiers modulo p , on peut le voir ainsi. Soit d un diviseur de $n = p - 1$. Les éléments de G_p qui vérifient $x^d = 1$ forment un sous-groupe : si $a^d = 1$ et $b^d = 1$, alors $(ab)^d = a^d b^d = 1$. De plus, le polynôme $x^d - 1$ a exactement d racines distinctes modulo p . En effet, d'après Fermat,

$$x^{p-1} - \bar{1} = (x - \bar{1})(x - \bar{2}) \cdots (x - \overline{p-1})$$

et $x^d - \bar{1}$ divise ce polynôme :

$$x^{p-1} - \bar{1} = (x^d - \bar{1})g(x)$$

avec g de degré $p - 1 - d$. Comme g ne peut pas avoir plus de racines que son degré, il faut que $x^d - \bar{1}$ en ait exactement d .

Ses racines sont les éléments de G_p dont les ordres sont des diviseurs c de d . Si on note $\psi(c)$ le nombre d'éléments d'ordre c , on a donc

$$d = \sum_{c|d} \psi(c)$$

pour tout diviseur d de n .

Par exemple, pour $p = 13$, $p - 1 = 12$ et

- $1 = \psi(1)$, donc $\psi(1) = 1$ (et 1 est l'unique élément d'ordre 1)
- $2 = \psi(1) + \psi(2) = \psi(2) + 1$, donc $\psi(2) = 1$ (-1 est l'unique élément d'ordre 2)
- $3 = \psi(1) + \psi(3) = \psi(3) + 1$, donc $\psi(3) = 2$
- $4 = \psi(1) + \psi(2) + \psi(4) = \psi(4) + 2$, donc $\psi(4) = 2$
- $6 = \psi(1) + \psi(2) + \psi(3) + \psi(6) = \psi(6) + 4$, donc $\psi(6) = 2$
- $12 = \psi(1) + \psi(2) + \psi(3) + \psi(4) + \psi(6) + \psi(12)$ donc $\psi(12) = 12 - 8 = 4$

Ceci entraîne que $\psi(p - 1) = \varphi(p - 1) \neq 0$, et donc qu'il existe au moins un élément d'ordre $p - 1$.

Une application en cryptographie : RSA

L'apparition des premiers réseaux informatiques dans les années 1970 a obligé à repenser la cryptographie. On ne connaissait alors que des systèmes symétriques : deux parties A et B qui veulent communiquer partagent une information secrète, la clef, qui leur permet de chiffrer et déchiffrer des messages.

Dans un réseau, il est impossible d'avoir autant de clefs que de liens bidirectionnels possibles.

On a donc cherché à construire des systèmes dans lesquels chaque noeud possède deux clefs : un secrète et une publique.

La connaissance de la clef publique doit permettre de chiffrer un message, mais il doit être impossible de le déchiffrer sans connaître la clef secrète.

Le premier système à résister sérieusement a été RSA, initiales de Rivest-Shamir-Adelman. Il est encore le plus utilisé. Il repose sur la difficulté de la factorisation des grands entiers. Si on multiplie deux grands nombres premiers p et q , il est impossible en pratique de retrouver p et q à partir de $n = pq$.

Pour construire une clef RSA, on tire au hasard deux nombres premiers p et q (module). On calcule aussi $\varphi(n) = (p-1)(q-1)$ (indicatrice d'Euler).

On choisit ensuite un *exposant public* $2 < e < n-1$, qui doit être inversible modulo $\varphi(n)$, et on calcule son inverse $d = e^{-1} \pmod{\varphi(n)}$ (l'*exposant privé*). On a donc $ed = 1 + k\varphi(n)$ pour un certain k .

Dans la description originale, d est choisi au hasard et e calculé ensuite, mais pour des raisons pratiques on utilise souvent les valeurs $e = 3, 257$ ou 65537 .

La clef publique est (n, e) , la clef secrète est (p, q, d) .

Les messages à chiffrer sont des blocs de $2N$ bits, interprétés comme des entiers modulo n . Pour chiffrer un tel message m , on calcule

$$c = m^e \pmod n.$$

Pour déchiffrer c , le destinataire calcule

$$c^d \pmod n = m^{ed} \pmod n = m^{1+k\varphi(n)} \pmod n = m^1 (m^{\varphi(n)})^k \pmod n = m$$

au moins si m est premier avec $\varphi(n)$ d'après le théorème d'Euler, et en fait dans tous les cas, à cause de la forme spéciale de n .

On a

$$\begin{aligned} c^d &= m(m^{k(q-1)})^{p-1} \equiv m \pmod p \\ c^d &= m(m^{k(p-1)})^{q-1} \equiv m \pmod q \end{aligned}$$

d'après le théorème de Fermat, et on peut donc retrouver m à partir de $m_p = c^d \pmod p$ et $m_q = c^d \pmod q$ en appliquant le théorème des reste chinois.

En pratique, on complète la clef secrète avec

$$\begin{aligned} d_p &= d \pmod{p-1} \\ d_q &= d \pmod{q-1} \\ q_{inv} &= q^{-1} \pmod p \end{aligned}$$

et le déchiffrement est obtenu en calculant

$$\begin{aligned} m_1 &= c^{d_p} \pmod p \\ m_2 &= c^{d_q} \pmod q \\ h &= (m_1 - m_2)q_{inv} \pmod p \\ m &= m_2 + hq. \end{aligned}$$

```
In [78]: # Construisons une clef RSA de 512 bits
from random import randrange
a = randrange(2**255, 2**256-1)
if not a%2: a+=1
a
```

```
Out[78]: 30358304457746634348968807386149670891610749136371796026609015759619707146199
```

```
In [79]: while not miller_rabin(a): a+=2
```

```
In [80]: p = a
p
```

```
Out[80]: 30358304457746634348968807386149670891610749136371796026609015759619707146401
```

```
In [81]: b = randrange(p, 2**256-1)
if not b%2: b+=1
while not miller_rabin(b): b+=2
q = b
```

```
In [82]: q
```

```
Out[82]: 104338636943374768349563363376815813699796604768751332947405120187362535670357
```

```
In [83]: n = p*q
n
```

```
Out[83]: 3167544107033261896874401363624155761574014046766123471938601248291409921144867595060348751729727952
267595440487817438133268275119735917598695510074935157
```

```
In [84]: print ('%x' % p)
print ('%x' % q)

431e310030ef52de19206cc9ae6fdfe013a6e1f289ac8c31bff3fc3d0b9490a1
e6ad93630c0d1bee715a73276aa5e168c632b7ff4a978c6f5e4a9564b9820255
```

```
In [85]: phi_n = (p-1)*(q-1)
e = 257
d = inversemod(e,phi_n)
print (d)

2021312192814999809678606317643430135790421415056981515167045154551716836839440448605087203788454571
579301657200502214193834851854027051236929021651624193
```

```
In [86]: d1, d2, q_inv = d % (p-1), d % (q-1), inversemod(q,p)
```

```
In [87]: print (d1)
print (d2)
print (q_inv)

26460156414534031494821061690651853228485633488510826108795406732119900392193
46282508216127329151168184532906625532205497835165960918304216736806727884905
14858605438657278859230927185393511292505254705106928885455764613764966977598
```

```
In [88]: from codecs import encode, decode
def text2int(t):
    assert len(t) <= 64 # 64*8 = 512 bits
    if isinstance(t,str):
        return int(encode(t.encode(),'hex'), 16) # t.encode() nécessaire en Python 3 pour avoir des
        bytes
    elif isinstance(t,bytes):
        return int(encode(t,'hex'), 16)

def int2text(m):
    s = '%x' % m
    if len(s)%2: s = '0'+s
    return decode(s, 'hex')
```

```
In [89]: msg = "L'arithmétique, c'est bien difficile !"
m = text2int(msg)
print (m)

2482049485033484295509148869034669333161266776700492625573073315249287942938347748044373827617
```

```
In [90]: msg.encode() # l'encodage par défaut est celui du système
```

```
Out[90]: b"L'arithm\xc3\xa9tique, c'est bien difficile !"
```

```
In [91]: # Chiffrement
c = pow(m,e,n)
print (c)

1061320271622173132144251972774038769862043843117729431061962149664361510522455936806165545568752838
340173176220571371208202500488348647879700941847410539
```

```
In [92]: # Déchiffrement
test = pow(c,d,n)
print (test)
int2text(test)

2482049485033484295509148869034669333161266776700492625573073315249287942938347748044373827617
```

```
Out[92]: b"L'arithm\xc3\xa9tique, c'est bien difficile !"
```

```
In [93]: print(_.decode('utf8'))

L'arithmétique, c'est bien difficile !
```

```
In [94]: # Déchiffrement accéléré avec les paramètres précalculés
m1, m2 = pow(c,d1,p), pow(c,d2,q)
h = (m1-m2)*q_inv % p
test2 = m2 + h*q
print (test2)
print (int2text(test2).decode('utf8'))

2482049485033484295509148869034669333161266776700492625573073315249287942938347748044373827617
L'arithmétique, c'est bien difficile !
```

```
In [ ]:
```

In []: