TP3

September 19, 2022

1 TP3 (Factorisation d'entiers, Fermat, RSA)

1.0.1 Exo 1

[1]:
```
def Facto1(N):
    k=2
    L=[]
    while k <= float(sqrt(N)):
        if N%k==0:
            L.append(k)
            N=N//k        # k divise N, on actualise N et on garde le même k (si␣
↪par ex. k^2 divise N)
        else:
            k=k+1      # k ne divise pas N, on passe au k suivant
    # Le N courant est premier (divisible par aucun entier <= sqrt(N)) et␣
↪divise le N d'entrée.
    # On l'ajoute à la liste des diviseurs, sauf s'il vaut 1.
    if N==1:
        return L
    L.append(N)
    return L
```

[2]:
```
%time Facto1(3**23+1)
```

```
CPU times: user 4.82 s, sys: 16.5 ms, total: 4.84 s
Wall time: 4.83 s
```

[2]:
```
[2, 2, 23535794707]
```

[3]:
```
def Facto2(N):
    L=[]
    while N%2==0:
        L.append(2)
        N=N//2
    k=3
    while k <= float(sqrt(N)):
        if N%k==0:
            L.append(k)
```

```
            N=N//k        # k divise N, on actualise N et on garde le même k (si␣
↪par ex. k^2 divise N)
        else:
            k=k+2      # k ne divise pas N, on passe au k suivant en ne␣
↪parcourant que les impairs
    # Le N courant est premier (divisible par aucun entier <= sqrt(N)) et␣
↪divise le N d'entrée.
    # On l'ajoute à la liste des diviseurs, sauf s'il vaut 1.
    if N==1:
        return L
    L.append(N)
    return L
```

[4]:
```
%time Facto2(3**23+1)
```

```
CPU times: user 2.43 s, sys: 39 µs, total: 2.43 s
Wall time: 2.43 s
```

[4]:
```
[2, 2, 23535794707]
```

Deux fois moins de temps, ce qui semble cohérent…

[5]:
```
def Facto3(N):
    L=[]
    while N%2==0:
        L.append(2)
        N=N//2
    while N%3==0:
        L.append(3)
        N=N//2
    k=5
    while k <= float(sqrt(N)):
        if N%k==0:
            L.append(k)
            N=N//k        # k divise N, on actualise N et on garde le même k (si␣
↪par ex. k^2 divise N)
        elif k%6==-1: # k ne divise pas N, on passe au k suivant en ne␣
↪parcourant que les +/-1 mod 6
            k=k+2
        else:
            k=k+4
    # Le N courant est premier (divisible par aucun entier <= sqrt(N)) et␣
↪divise le N d'entrée.
    # On l'ajoute à la liste des diviseurs, sauf s'il vaut 1.
    if N==1:
        return L
    L.append(N)
```

```
    return L
```

[6]:
```
%time Facto3(3**23+1)
```

```
CPU times: user 1.24 s, sys: 3.99 ms, total: 1.25 s
Wall time: 1.25 s
```

[6]:
```
[2, 2, 23535794707]
```

Trois fois moins de temps que Facto 1 (environ), ce qui semble cohérent.

[7]:
```
def Facto4(N):
    L=[]
    k=2
    while k <= float(sqrt(N)):
        if N%k==0:
            L.append(k)
            N=N//k        # k divise N, on actualise N et on garde le même k (si␣
↪par ex. k^2 divise N)
        else: # k ne divise pas N, on passe au k suivant en ne parcourant que␣
↪les +/-1 mod 6
            k=next_prime(k)
    if N==1:
        return L
    L.append(N)
    return L
```

[8]:
```
%time Facto4(3**23+1)
```

```
CPU times: user 486 ms, sys: 3.97 ms, total: 490 ms
Wall time: 491 ms
```

[8]:
```
[2, 2, 23535794707]
```

Encore plus rapide, ce qui semble cohérent (en supposant next_prime rapide bien entendu)

[9]:
```
%time factor(3**23+1)
```

```
CPU times: user 74 µs, sys: 0 ns, total: 74 µs
Wall time: 76.8 µs
```

[9]:
```
2^2 * 23535794707
```

Sage utilise manifestement un algorithme beaucoup plus performant…

1.0.2 Exo 2 (pseudo-primalité de Fermat, puis de Miller-Rabin)

[10]:
```
def Fermat(N,a):
    b=power_mod(a,N-1,N)
    return (b==1) # True si pseudo-premier de Fermat et False sinon (auquel cas␣
 ↪N est non premier)


N=1
c=0

while Fermat(N,3)==False:
    N=randint(1,10^200)
    c+=1
print(c)
is_prime(N)
```

```
406
```

[10]:
```
True
```

Attention, power_mod retourne un entier (regarder .parent() pour vérifier). Si on veut vraiment travailler modulo N, on peut faire : R=ZZ.quotient_ring(N) puis écrire R(a), R(b), etc.

[11]:
```
b=power_mod(5,6,7)
print("b=",b)
print(b.parent())
print(b==8)
print()
R=ZZ.quotient_ring(7)
b=R(5)**6
print("b=",b)
print(b.parent())
print(b==8)
print()
```

```
b= 1
Integer Ring
False

b= 1
Ring of integers modulo 7
True
```

[12]:
```
def Miller(N,a):
```

```
    a=a%N
    if a==0:
        print("mauvais a")
        return(False)
    if gcd(a,N)>1:
        return(False) # non premier
    t=N-1
    s=0
    while t%2==0:
        t=t//2
        s=s+1
    # N-1=t.2^s avec t impair
    b=power_mod(a,t,N)
    if b==1 or b==N-1:  # attention, b est un entier
        return(True)  # pseudo-premier de Miller
    for i in range(1,s):
        b=(b**2)%N
        if b==N-1:
            return(True)  # pseudo-premier de Miller
    return(False)  # non premier
```

[13]:
```
c=1
N=3

while Miller(N,3)==False:
    N=randint(1,10^200)
    c=c+1
print(N,c)
is_prime(N)
```

```
mauvais a
984705936544131290058122076503534803856187837840569561910347214729667136123668480102004162424565134555981100132848305962302240261196769875445697214447645401310699304402292236937878777894029152338608 19 99
```

[13]:
```
True
```

[14]:
```
c=1
N=3

while Fermat(N,3)==False or Miller(N,3)==True:
    N=randint(1,10^5)
    c=c+1
print(N,c)
is_prime(N)
```

```
32791 2783
```

[14]: False

1.0.3 Exo 3 (nombres de Carmichael)

[15]:
```
def Carmichael(N):
    if is_prime(N)==True:
        return False
    for a in range(2,N):
        if power_mod(a,N,N)!=a:
            return False
    return True
```

[16]:
```
print(Carmichael(560),Carmichael(561))
```

```
False True
```

[17]:
```
def ListeCarmichael(M):
    L=[]
    for N in range(2,M):
        if Carmichael(N)==True:
            L.append(N)
    return L
```

[18]:
```
print(ListeCarmichael(100000))
```

```
[561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657,
52633, 62745, 63973, 75361]
```

Si N est un nombre de Carmichael (en particulier, N est composé) et si $1 \leq a \leq N-1$, alors $a^{N-1} = 1 \mod N$ (i.e. le test de Fermat se trompe) si et seulement si a est premier avec N i.e. si $a \in \mathbb{Z}/N\mathbb{Z}^\times$ (en effet, si a n'est pas premier à N, alors $a^{N-1} = 1 \mod N$ impliquerait que le $pgcd(a, N) > 1$ divise 1, contradiction). D'où une probabilité d'échecs $\varphi(N)/(N-1)$. On rappelle que si $N = \prod p_i^{r_i}$, alors $\varphi(N) = \prod_i p_i^{r_i-1}(p_i - 1)$. En particulier,

$\varphi(75361) = \varphi(11 \times 13 \times 17 \times 31) = 10 \times 12 \times 16 \times 30$

La proba d'échec du test de Fermat est donc $10 \times 12 \times 16 \times 30/75360 \approx 0.76 > 1/2$.

[19]:
```
def Phi(n):
    L=list(factor(N))
    res=prod(L[i][0]**(L[i][1]-1)*(L[i][0]-1) for i in range(len(L)))
    return res

Phi(75361)==10*12*16*30
```

[19]: False

[20]:
```
def ProbaErreurFermat(N,M):
    c=0
    for i in range(M):
        a=randint(2,N-1)
        if Fermat(N,a)==True:
            c=c+1
    return float(c/M)

N=75361
M=1000
proba_theorique = float(Phi(N)/(N-1))
proba_experimentale = ProbaErreurFermat(75361,1000)

print(proba_theorique,proba_experimentale)
```

```
0.7643312101910829 0.751
```

[21]:
```
def ProbaErreurMiller(N,M):
    c=0
    for i in range(M):
        a=randint(2,N-1)
        if Miller(N,a)==True:
            c=c+1
    return float(c/M)

N=75361
M=1000

print(ProbaErreurMiller(75361,1000))
```

```
0.004
```

Bien que 75361 soit de Carmichael, le taux d'échec du test de Miller-Rabin est très faible (moins de 1%).

[22]:
```
def Carmichael2(N):
    L=list(factor(N))
    if len(L)==1:
        return False
    for (p,m) in L:
        if m>1 or (N-1)%(p-1)!=0:
            return False
    return True

def ListeCarmichael2(M):
    L=[]
    for N in range(2,M):
        if Carmichael2(N)==True:
```

```
            L.append(N)
    return L

from time import time

%time L=ListeCarmichael(100000)
print(L)
%time L2=ListeCarmichael2(100000)
print(L2)
```

```
CPU times: user 1.39 s, sys: 34 µs, total: 1.39 s
Wall time: 1.39 s
[561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657,
52633, 62745, 63973, 75361]
CPU times: user 604 ms, sys: 8 µs, total: 604 ms
Wall time: 604 ms
[561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657,
52633, 62745, 63973, 75361]
```

La seconde approche est environ deux fois plus rapide.

1.0.4 Exo 4 (test de primalité de Lucas)

[23]:
```
def Lucas(N,a):
    a=a%N
    if power_mod(a,N-1,N)!=1:
        return False  # N non premier
    L=prime_factors(N-1)
    for q in L:
        if power_mod(a,(N-1)//q,N)==1:
            return("ce choix de a ne permet pas de conclure")
    return True

def TestLucas(N):
    a=2
    rep=Lucas(N,a)
    while rep=="ce choix de a ne permet pas de conclure":
        a=a+1
        rep=Lucas(N,a)
    return [rep,a]
```

[24]:
```
for n in [2017,2**16 + 1,2**34*3**29*5**2*7**29 + 1]:
    print(TestLucas(n))
```

```
[True, 5]
[True, 3]
[True, 17]
```

[25]:
```
N=2**22*7**15+1
print(TestLucas(N),TestLucas(2*N+1))
```

```
[True, 3] [True, 2]
```

1.0.5 Exo 5 (RSA)

[26]:
```
def GenCle():
    p=random_prime(10**6,10**7)
    q=random_prime(10**6,10**7)
    N=p*q
    phi=(p-1)*(q-1)
    e=phi
    while gcd(e,phi)!=1:
        e=randint(2,phi/2)
    d=inverse_mod(e,phi)
    return ([N,e],[p,q,d])
```

[27]:
```
([N,e],[p,q,d])=GenCle()
M=1234567890
m=power_mod(M,e,N)
M0=power_mod(m,d,N)
print(M0)
```

```
1234567890
```

Si le destinataire publie aussi φ par inadvertance, l'entier d se calcule via $ed = 1 \mod \varphi$ et on retrouve p, q via les racines de $x^2 - (p+q)x + pq = x^2 - (N - \varphi + 1)x + N$.

[28]:
```
def Casse(N,e,phi):
    d=inverse_mod(e,phi)
    P=x**2-(N-phi+1)*x+N
    [(p,m1),(q,m2)]=P.roots()
    return(p,q,d)
```

Bizarrement, x n'a pas besoin d'être déclarée variable. Un autre choix (par exemple y) aurait requis de taper var('y') avant d'appeler roots()

Noter aussi la fonction solve([P==0],x). Mais la sortie est une expression symbolique dont il est plus difficile d'extraire les valeurs p et q (d'habitude avec args(), mais là cela ne fonctionne pas manifestement).

[29]:
```
([N,e],[p,q,d])=GenCle()
phi=(p-1)*(q-1)
print(Casse(N,e,phi))
print(p,q,d)
```

```
(890053, 575303, 457922360801)
890053 575303 457922360801
```

[30]:
```
def Casse_avec_d(N,e,d):
    a=randint(2,N-1)
    g=gcd(a,N)
    if g>1:
        return g
    M=e*d-1  # Comme a est premier avec N et M est multiple de phi(N), on a␣
↪donc a^M=1 mod N.
    t=M
    s=0
    while t%2==0:
        t=t//2
        s=s+1
    # Donc M = t 2^s avec t impair
    b=power_mod(a,t,N) # Donc b^(2^s)=1 mod N et on cherche la plus petite␣
↪puissance i<=s telle que b^i=1 mod N
    B=power_mod(b,2,N)
    while B!=1:
        b=B
        B=power_mod(b,2,N)
    if b==1 or b==N-1:
        print("rien trouvé")
        return Casse_avec_d(N,e,d)
    p=gcd(b-1,N)
    return([p,N//p])
```

[31]:
```
([N,e],[p,q,d])=GenCle()
print(Casse_avec_d(N,e,d))
print(p,q)
```

```
[461413, 347239]
347239 461413
```

1.0.6 Exo 6 (racines carrées modulo N)

[32]:
```
def RacineCarreeModulaire(N):
    c=0
    for x in range(N):
        if x**2%N==1:
            c=c+1
    return c
```

[33]:
```
for N in [1049, 1079, 1139, 1209, 1289, 4913]:
    print (RacineCarreeModulaire(N),factor(N))
```

```
2 1049
4 13 * 83
4 17 * 67
```

```
8 3 * 13 * 31
2 1289
2 17^3
```

On peut conjecturer que le nombre de racines modulo N vaut $2^{w(N)}$ où $w(N)$ est le nombre de facteurs premiers de N. Cette conjecture est vraie si 2 ne divise pas N. Si $N = 2^a M$, le nombre de racines est $k2^{w(M)}$ où $k = 1$ si $a = 0$ ou 1, $k = 2$ si $a = 2$ et $k = 4$ si $a \geq 3$. La preuve se base sur l'équation $(x-1)(x+1) = 0 \mod N$ que l'on résoud via le théorème des restes chinois (cf corrigé CM2).

[34]:
```
N=8*3*5*7*11*13
print(RacineCarreeModulaire(N))
print(4*2**5)
```

```
128
128
```

[ ]:
```

```