TP2

September 12, 2022

1 TP 2 : Euclide et compagnie

1.0.1 Exo 1 (division euclidienne)

[1]:
```
# division euclidienne

def Division(a,b):
    r=a; q=0
    while r.degree() >= b.degree():
        c=r.lt()//b.lt()  # quotient des termes dominant avec // pour rester
↪dans Q[x] (et non Q(X))
        q=q+c
        r=r-c*b
    return (q,r)
```

[2]:
```
R=QQ['X']
X=R.gen()
a=R.random_element(degree=10)
b=R.random_element(degree=5)
q,r = Division(a,b)
print(q)
print(r)
q,r = a//b, a%b
print(q)
print(r)
```

```
3/2*X^5 - 3/4*X^4 - 3/8*X^3 - 33/16*X^2 - 33/32*X - 129/64
-111/128*X^4 - 5735/128*X^3 - 775/256*X^2 - 113/64*X + 1
3/2*X^5 - 3/4*X^4 - 3/8*X^3 - 33/16*X^2 - 33/32*X - 129/64
-111/128*X^4 - 5735/128*X^3 - 775/256*X^2 - 113/64*X + 1
```

[3]:
```
a=R.random_element(degree=1000)
b=R.random_element(degree=10)
len(str(a%b))
```

[3]: 20871

1.0.2 Exo 2 (multiplication dans les corps de nombres)

[4]:
```
# Multiplication dans un corps de nombres (éléments représentés par des listes)

def Mult(a,b):
    A=sum(a[i]*X^i for i in range(len(a)))
    B=sum(b[i]*X^i for i in range(len(b)))
    C=A*B
    D=C%(X^4-2)
    res=D.list()
    return(res)
```

[5]:
```
a=[1,1,1,1]
P=Mult(Mult(a,a),Mult(a,a))
print(P)

# Pour calculer a^{1000}, utiliser l'exponentiation rapide
# et penser bien sûr à réduire modulo X^4-2 à chaque multiplication.
```

```
[195, 164, 138, 116]
```

[6]:
```
# Definition d'un corps de nombre avec Sage

K.<z>=R.quo(X^4-2)
a=z**3+z**2+z+1
a**4
```

[6]:
```
116*z^3 + 138*z^2 + 164*z + 195
```

1.0.3 Exo 3 (pgcd)

[7]:
```
def pgcd(a,b):
    if b==0 :
        return a
    return pgcd(b,a%b)


def pgcdIt(a,b):
    while b!=0:
        [a,b]=[b,a%b]
    return a

print(pgcd(246,114))
print(pgcd(X**3-1,X**2-1))

## avec sage
```

```
print(gcd(123,456))
print(gcd(X^2-4,X-2))
print((X**3-1).gcd(X**2-1))
# attention, gcd(t,5*t) ne fonctionne pas si la variable "t" n'est pas déclarée.
```

```
6
X - 1
3
X - 2
X - 1
```

[8]:
```
def pgcdCentre(a,b):
    while b!=0:
        r=a%b
        if r>b/2:
            r=r-b
        [a,b]=[b,r]
    return a
```

[9]:
```
pgcdCentre(156,42)
```

[9]:
```
6
```

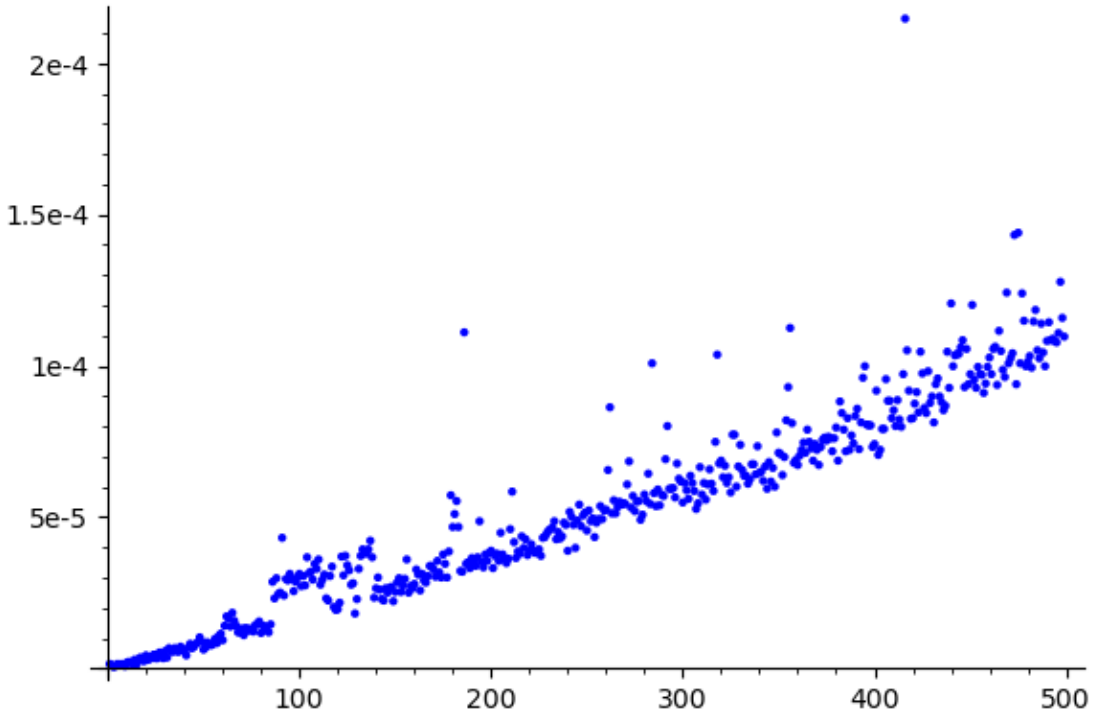
[10]:
```
from time import time

def tempsPgcd(N):
    a=randint(0,2^N)
    b=randint(0,2^N)
    t0=time()
    pgcd(a,b)
    return(time()-t0)
```

[11]:
```
tempsPgcd(1000)
```

[11]:
```
0.0004467964172363281
```

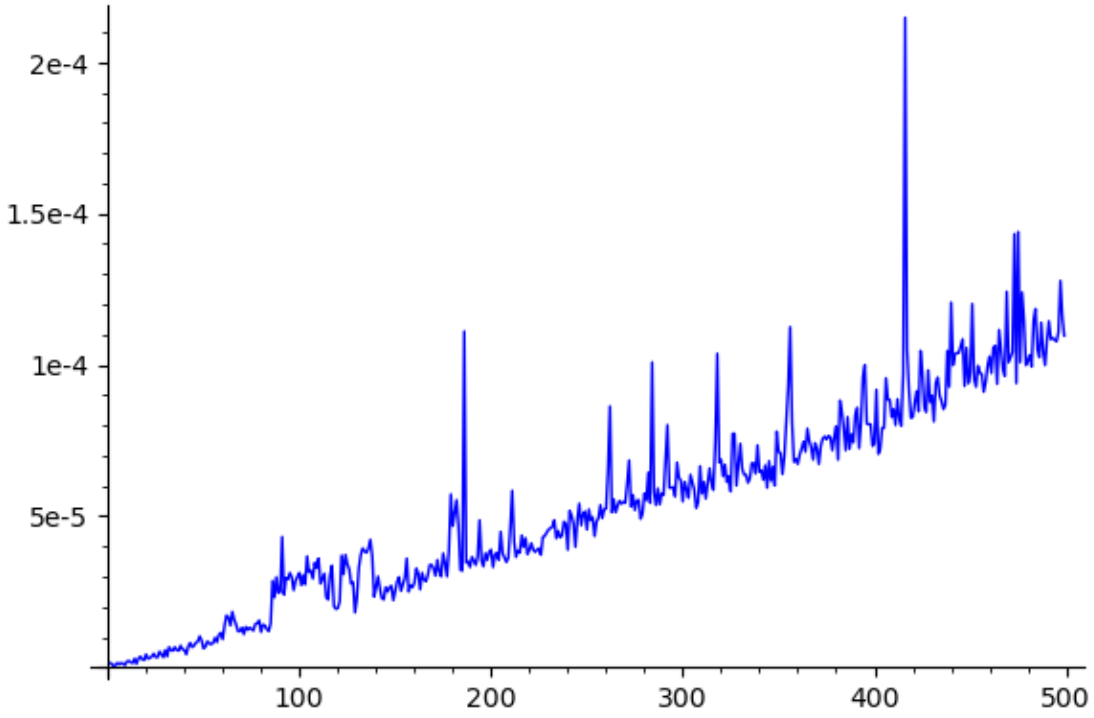
[12]:
```
L=[]
for N in range(1,500):
    L.append((N,tempsPgcd(N)))
point(L)
```

[12]:



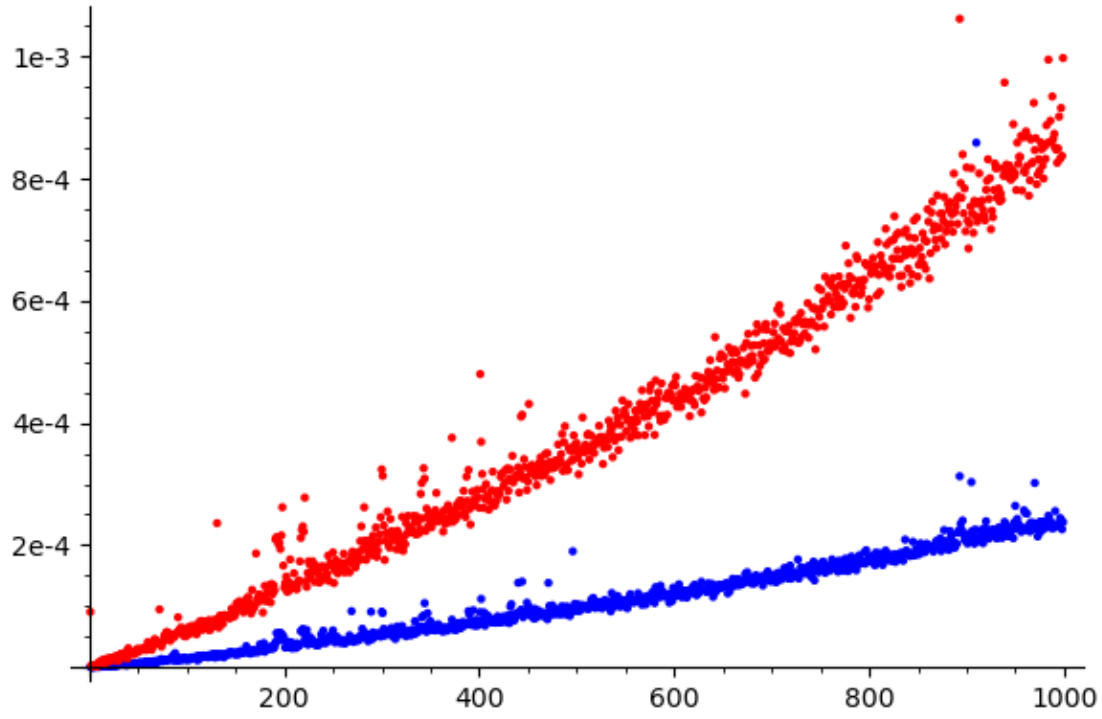
[13]:
```
line2d(L)
```

[13]:



[14]:
```
def tempsPgcdCentre(N):
    a=randint(0,2^N)
    b=randint(0,2^N)
    t0=time()
    pgcdCentre(a,b)
    return(time()-t0)

L=[]
Lcentre=[]
for N in range(1,1000):
    L.append((N,tempsPgcd(N)))
    Lcentre.append((N,tempsPgcdCentre(N)))
point(L)+point(Lcentre,color='red')
```

[14]:



1.0.4 Exo 4 (Nombres d'itérations de l'algorithme d'Euclide)

[15]:
```
def NbIterationsPgcd(a,b):
    c=0
    while b!=0:
        c=c+1
        [a,b]=[b,a%b]
```

```
    return c

def StatPgcd(N,B):
    s=0
    m=0
    for i in range(N):
        a=randint(0,B)
        b=randint(0,B)
        n=NbIterationsPgcd(a,b)
        s=s+n
        m=max(m,n)
    return(float(s/N),m)
```

[16]:
```
print (StatPgcd(1000,100000))
```

```
(9.778, 19)
```

[17]:
```
float(12*log(2)*log(100000)/pi^2)
```

[17]: 9.70270114392034

Calcul du k-ème terme de Fibonacci Fibonnaci naïf :

[18]:
```
def FibNaif(k):
    if k==0:
        return(0)
    if k==1:
        return (1)
    return (FibNaif(k-2)+FibNaif(k-1))
```

[19]:
```
FibNaif(20)
```

[19]: 6765

Complexité exponentielle catastrophique dûe au caractère récursif ! (chaque terme est calculé un grand nombre de fois).

On préfèrera utiliser par exemple une version matricielle :

[20]:
```
def Fib(k):
    M=matrix([[0,1],[1,1]])
    return matrix([0,1])*M^k

Fib(20)
```

[20]:
```
[ 6765 10946]
```

Sinon, ce qui revient essentiellement au même :

[21]:
```
def Fib2(k):
    u=0
    v=1
    for i in range(k):
        temp=u
        u=v
        v=temp+v
    return ([u,v])

Fib2(20)
```

[21]:
```
[6765, 10946]
```

Itérations maximales du pgcd versus Fibonacci

[22]:
```
for k in range(10,101,10):
    B0,B1=list(Fib(k))[0]
    print(StatPgcd(10000,B1))
    print(NbIterationsPgcd(B0,B1))
```

```
(3.8263, 10)
10
(7.8681, 16)
20
(12.0013, 21)
30
(16.0, 29)
40
(20.0406, 34)
50
(24.0792, 38)
60
(28.1903, 44)
70
(32.2853, 49)
80
(36.2679, 52)
90
(40.3066, 59)
100
```

1.0.5 Exo 4 (euclide étendu)

Version itérative (matricielle)

[23]:
```
def Bezout(a,b):
    M=matrix([[0,1],[1,0]])
    while b!=0:
        q,r=a//b, a%b
        a,b=b,r
        M=matrix([[0,1],[1,-q]])*M
    return M[0][1],M[0][0],a
```

[24]:
```
Bezout(13,29)
```

[24]:
```
(9, -4, 1)
```

Version récursive

[25]:
```
def BezoutRec(a,b):
    if b==0:
        return [1,0,a]
    q,r=a//b, a%b
    [u,v,d]=BezoutRec(b,r)
    return [v,u-q*v,d]
```

[26]:
```
BezoutRec(13,29)
```

[26]:
```
[9, -4, 1]
```

[27]:
```
xgcd(13,29)
```

[27]:
```
(1, 9, -4)
```

1.0.6 Exo 5 : équations modulo N

L'équation $ax+b = 0 \mod N$ a une solution ssi il existe $x,k \in \mathbb{Z}$ tels que $ax+kN = -b$, i.e. ssi b appartient à l'idéal (a,N), i.e. ssi $d := pgcd(a,N)$ divise b. On cherche d'abord (u,v) tels que $au+vN = d$ à l'aide d'Euclide étendu. On effectue ensuite la division euclidienne $-b = qd+r$ de $-b$ par d. Si $r \neq 0$, d ne divise pas b et il n'y a aucune solution. Si $r = 0$ on obtient $aqu+vqN = -b$. Ainsi $x = qu$ est solution. En posant $a' = a/d$, $N' = N/d$ et $b' = b/d$, on obtient $a'qu+N'vq = -b'$ avec a',N' premiers entre eux. L'entier $x = qu$ est donc unique modulo N'. Il y a donc d solutions modulo N, qui sont les $qu+kN'$ avec $k = 0,\ldots,d-1$.

[28]:
```
def resoud(a,b,N):
    [d,u,v]=xgcd(a,N)
    q,r=-b//d,-b%d
    if r!=0:
        print("pas de solutions")
        return []
    N0=N//d
```

```
    return [(q*u+k*N0)%N for k in range(d)]
```

[29]:
```
Res=resoud(126,258,48)
print(Res)
L=[(126*x+258)%48 for x in Res]
print(L)
```

```
[33, 41, 1, 9, 17, 25]
[0, 0, 0, 0, 0, 0]
```

Directement avec Sage :

[30]:
```
var('x')
S=solve_mod([126*x+258==0],48)
S
```

[30]:
```
[(33,), (1,), (17,), (9,), (25,), (41,)]
```

[31]:
```
print(S[0][0])
S[0][0].parent()
```

```
33
```

[31]:
```
Ring of integers modulo 48
```

1.0.7 Exo 6 (restes chinois)

1) Modules premiers entre eux

[32]:
```
def RestesChinois(a1,a2,n1,n2):
    d,u,v = xgcd(n1,n2)
    x= a2*u*n1 + a1*v*n2
    return x%(n1*n2)
```

[33]:
```
RestesChinois(5,7,18,25)
```

[33]:
```
257
```

[34]:
```
crt(5,7,18,25)
```

[34]:
```
257
```

2) Modules non premiers entre aux

(a) Si x est solution, on a $x = a_1 \mod d$ et $x = a_2 \mod d$ du fait que d divise n_1 et n_2. Il s'ensuit que $a_1 - a_2 = 0 \mod d$.

(b) La relation de Bezout $un_1 + vn_2 = d$ fournit l'égalité :

$$x := a_1 + u\frac{n_1}{d}(a_2 - a_1) = a_1\left(\frac{un_1 + vn_2}{d}\right) + u\frac{n_1}{d}(a_2 - a_1) = a_2u\frac{n_1}{d} + a_1v\frac{n_2}{d}$$

De même, on trouve $a_2 + v\frac{n_2}{d}(a_1 - a_2) = a_2u\frac{n_1}{d} + a_1v\frac{n_2}{d} = x$.

(c) Si $a_1 - a_2 = 0 \bmod d$, alors $(a_2 - a_1)/d \in \mathbb{Z}$ et on déduit immédiatement des égalités précédentes que $x = a_1 \bmod n_1$ et $x = a_2 \bmod n_2$. S'il existe une autre solution y, on a $x = y \bmod n_1$ et $x = y \bmod n_2$. Donc $x = y \bmod ppcm(n_1, n_2)$ et on conclut avec l'égalité $ppcm(n_1, n_2) = n_1n_2/pgcd(n_1, n_2)$.

[35]:
```
def RestesChinoisNonPremiers(a1,a2,n1,n2):
    d,u,v = xgcd(n1,n2)
    if (a1-a2)%d != 0:
        return()
    x=(a2*u*n1+a1*v*n2)//d
    return x%(n1*n2//d)
```

[36]:
```
print(RestesChinoisNonPremiers(19,13,18,21))
print(crt(19,13,18,21))
print(RestesChinoisNonPremiers(19,14,18,21))
#print(crt(19,14,18,21))
```

```
55
55
()
```

3) Restes chinois multiples On pose $n = n_1 \cdots n_k$, $m_i = n/n_i$ et $u_in_i + v_im_i = 1$. Une solution est alors $x = \sum_{i=1}^{k} a_iv_im_i$, unique modulo n.

[37]:
```
def RestesChinoisMultiple(A,N):
    k=len(N)
    n=prod(N)
    x=0
    for  i in range(len(N)):
        mi= n //N[i]
        vi= xgcd (N[i],mi)[2]
        x=x + A[i]*vi*mi
    return x%n
```

[38]:
```
r=RestesChinoisMultiple([3,5,8],[5,26,37])
s=crt([3,5,8],[5,26,37])
print(r,s)
```

```
4633 4633
```

4) Diviser pour régner Si $k = 1$, on retourne $P = a_1$. Sinon, on résoud récursivement les deux systèmes de congruences constitués des $k/2$ premières équations et des

$k/2$ dernières. On obtient des solutions P_1 et P_2 pour chaque système. On calcule $Q_1 = \prod_{i=1}^{k/2} x_i$, $Q_2 = \prod_{i=k/2+1}^{k} x_i$ et U, V tels que $UQ_1 + VQ_2 = 1$. On vérifie que $P = P_2UQ_1 + P_1VQ_2 = \mod Q$ est solution du problème (unique modulo Q).

[39]:
```
def RestesChinoisMultipleRapide(A,L):
    r=1
    while r<len(L):
        r=2*r
    if r==1:
        return A[0]
    L1=L[:r//2]
    L2=L[r//2:]
    A1=A[:r//2]
    A2=A[r//2:]
    P1=RestesChinoisMultipleRapide(A1,L1)
    P2=RestesChinoisMultipleRapide(A2,L2)
    Q1=prod(L1)
    Q2=prod(L2)
    D,U,V=xgcd(Q1,Q2)
    P=P2*U*Q1 + P1*V*Q2
    return(P%(Q1*Q2))
```

[40]:
```
r=RestesChinoisMultipleRapide([1,2,3,4],[5,6,7,11])
s=crt([1,2,3,4],[5,6,7,11])
print (r,s)
```

```
1676 1676
```

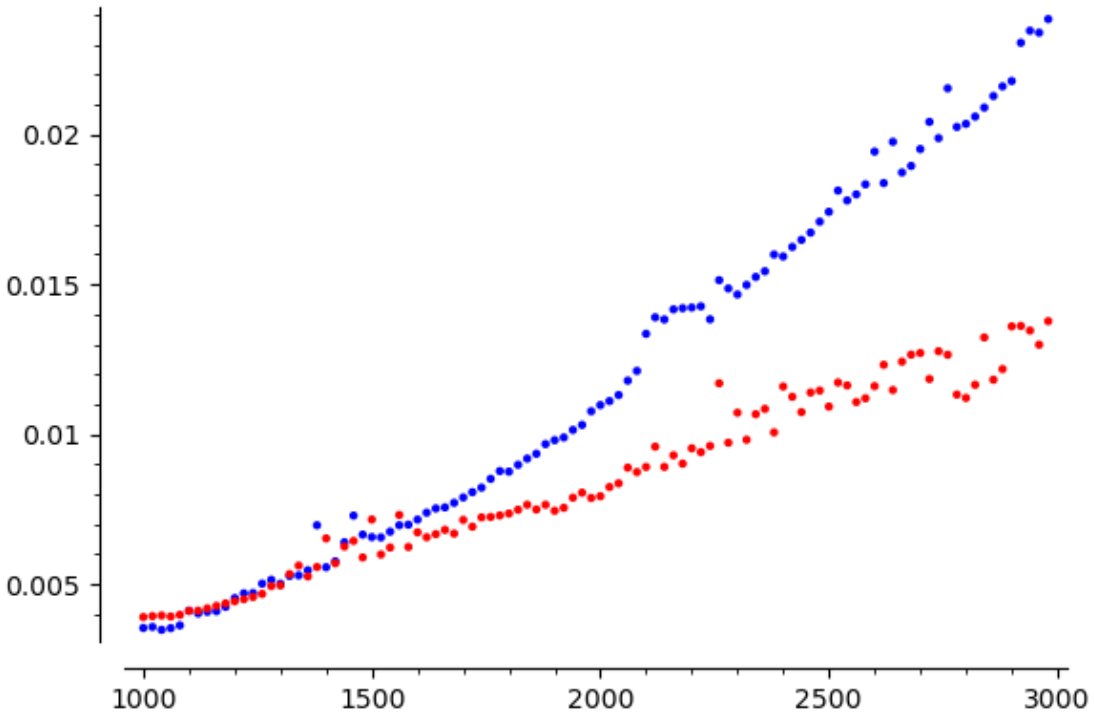
5) Courbes de complexité

[41]:
```
LL=list(primes(100000))
AA=[randint(0,100000) for i in range(100000)]

def StatChinois(N):
    L=LL[:N]
    A=AA[:N]
    t0=time()
    RestesChinoisMultiple(A,L)
    t1=time()
    return(t1-t0)

def StatChinoisRapide(N):
    L=LL[:N]
    A=AA[:N]
    t0=time()
    RestesChinoisMultipleRapide(A,L)
    t1=time()
```

```
    return(t1-t0)
```

[42]:
```
T0=[]
for N in range(1000,3000,20):
    t=StatChinois(N)
    T0.append([N,t])

T1=[]
for N in range(1000,3000,20):
    t=StatChinoisRapide(N)
    T1.append([N,t])

point(T0,color='blue')+point(T1,color='red')
```

[42]:



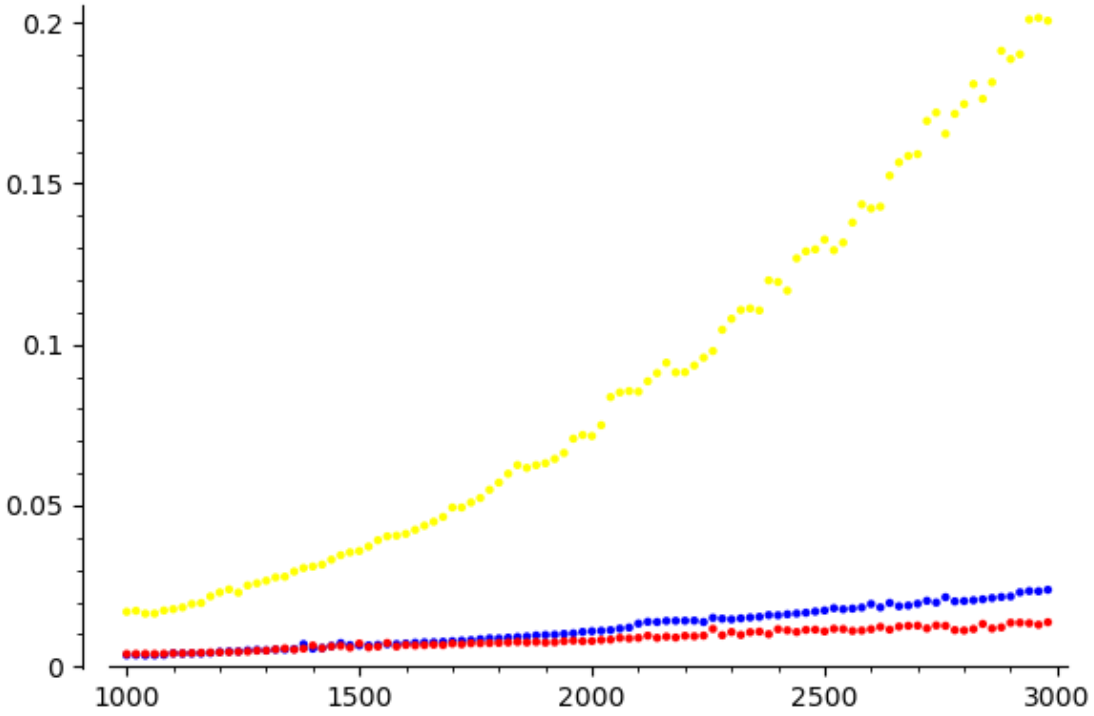
On constate une complexité visiblement quadratique pour Chinois multiple classique vs une complexité qui semble quasi-linéaire pour Chinois multiple rapide.

On peu se poser la question de l'efficacité de Sage :

[43]:
```
def StatChinoisSage(N):
    L=LL[:N]
    A=AA[:N]
    t0=time()
    crt(A,L)
```

```
    t1=time()
    return(t1-t0)

T2=[]
for N in range(1000,3000,20):
    t=StatChinoisSage(N)
    T2.append([N,t])

point(T0,color='blue')+point(T1,color='red')+point(T2,color='yellow')
```

[43]:



Bouhhh !! Le crt multiple de Sage semble très mal programmé lorsqu'il y a beaucoup d'équations.

6) Complexité théorique : A écrire...

7) Interpolation polynomiale via CRT. C'est exactement la même idée que précédemment, en remplaçant $n_i \in \mathbb{Z}$ par $X - x_i \in K[X]$ dans l'algorithme précédent (on a $P(x_i) = P \bmod (X - x_i)$). On calcule $Q = (X - x_1)\cdots(X - x_k)$, $M_i = Q/(X - x_i)$ et $U_i(X - x_i) + V_i M_i = 1$. Une solution est alors $P = \sum_{i=1}^{k} a_i V_i M_i$, unique modulo Q.

Attention cependant de bien définier l'anneau dans lequel vivent les polynômes.

[44]:
```
def Interpolation(A,L):
    Lpol=[X-c for c in L]
```

```
    P=RestesChinoisMultiple(A,Lpol)
    return P

R.<X>=PolynomialRing(QQ)

P=Interpolation([1,2,3],[5,2,7])
print(P)
print()
print(P(5),P(2),P(7))
print()
Q=crt([1,2,3],[X-5,X-2,X-7])
print(Q)
```

```
4/15*X^2 - 11/5*X + 16/3

1 2 3

4/15*X^2 - 11/5*X + 16/3
```