TP2

TP2: Autour de RSA

DUONG Minh Nghia - KRAAIJENBRINK Antony

1 Préliminaire : SageMath et les nombres entiers

SageMath possède quelques commandes très utiles pour travailler avec les nombres premiers : is_prime, prime_range, next_prime, factor, prime_pi.

2 Nombres premiers

2.1 Sur la répartition des nombres premiers

1. Evaluer les limites de la commande prime_pi.

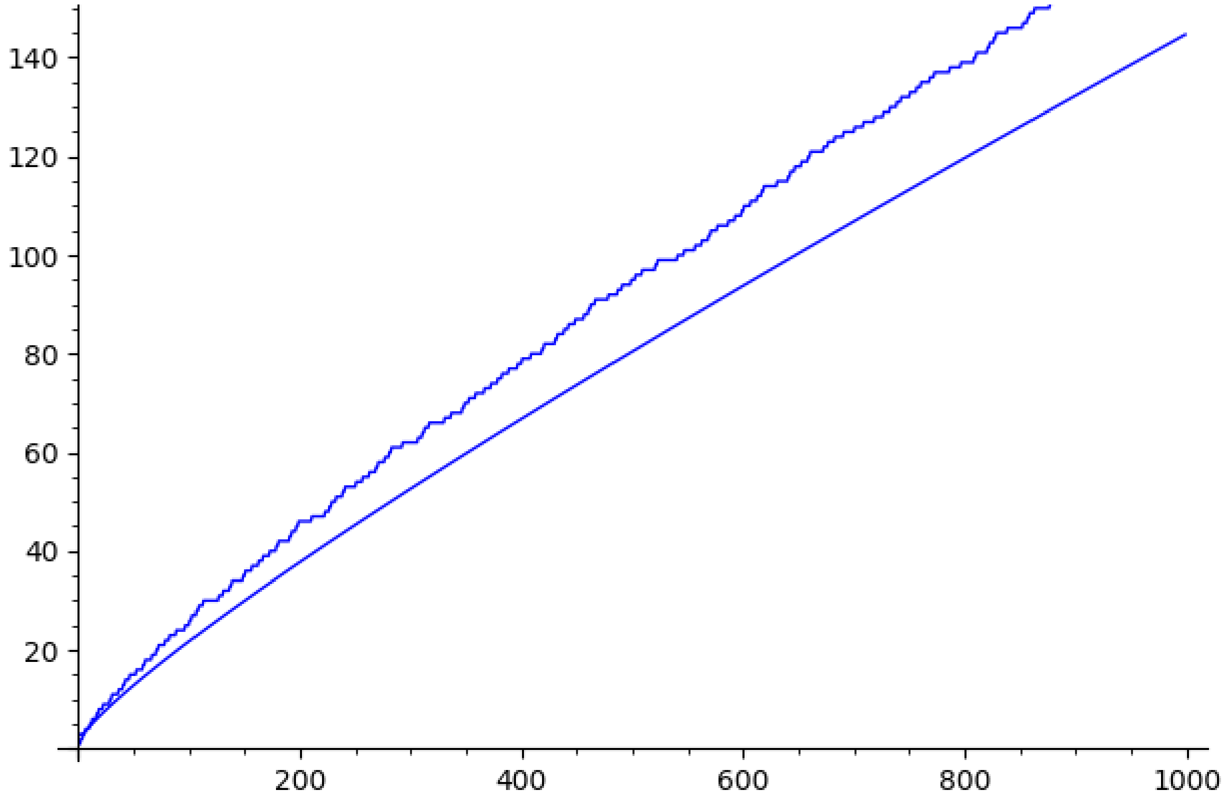
Pour tester la limite de la commande prime_pi, nous avons fait augmenter le paramètre n jusqu'à le temps d'exécution dépasse uune minute. Nous obtenons le temps d'exécution de l'ordre minute quand n >= 10^14

```
prime_pi(100000000000000)
```

```
3204941750802
```

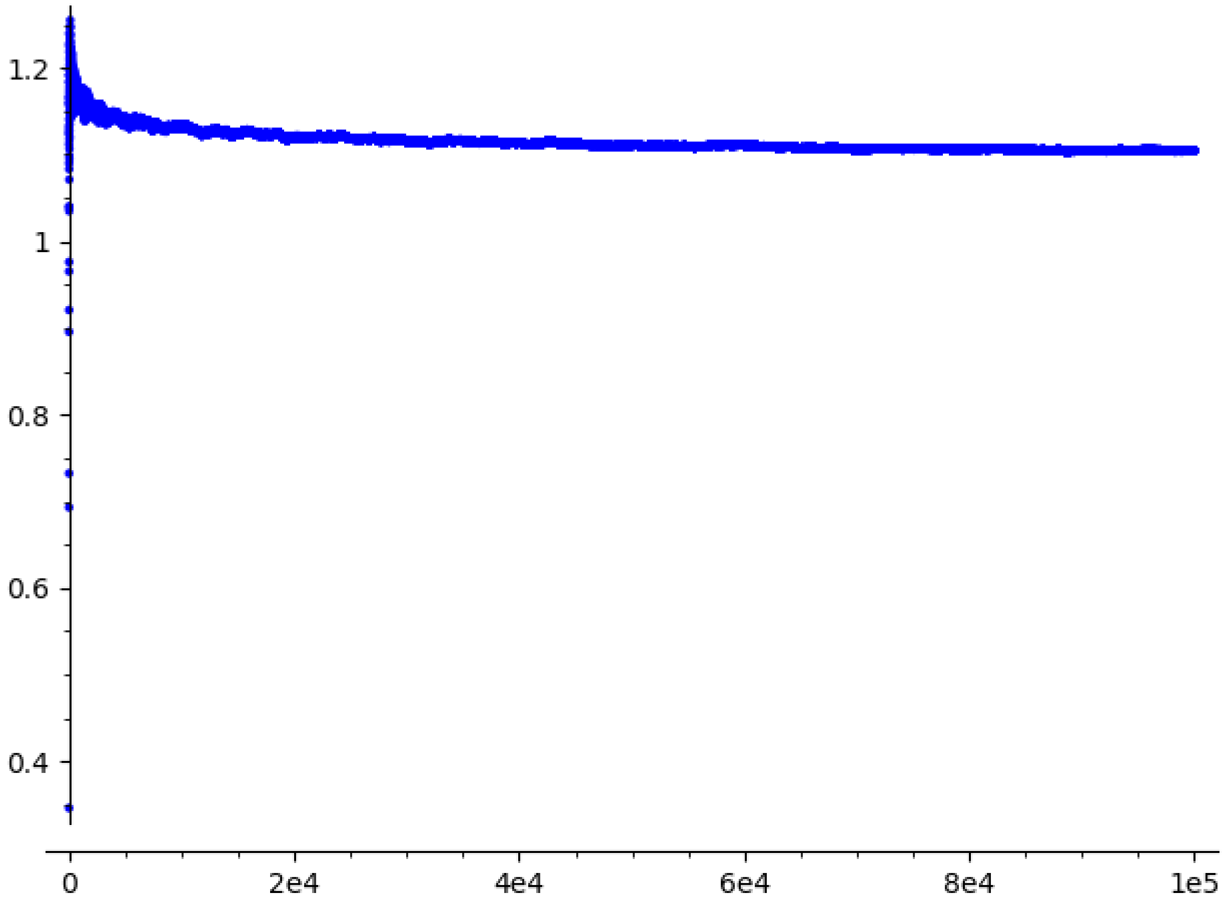
2. Tracer sur un meme graphe n → π(n) et n → n/ln(n)

```
vectorPrimePi =[(n, prime_pi(n)) for n in range(2,1000)]
vectorN_lnN = [(n,n/ln(n)) for n in range(2,1000)]
line(vectorPrimePi) + line(vectorN_lnN)
```



3. Tracer sur un graphe u_n en fonction de n.

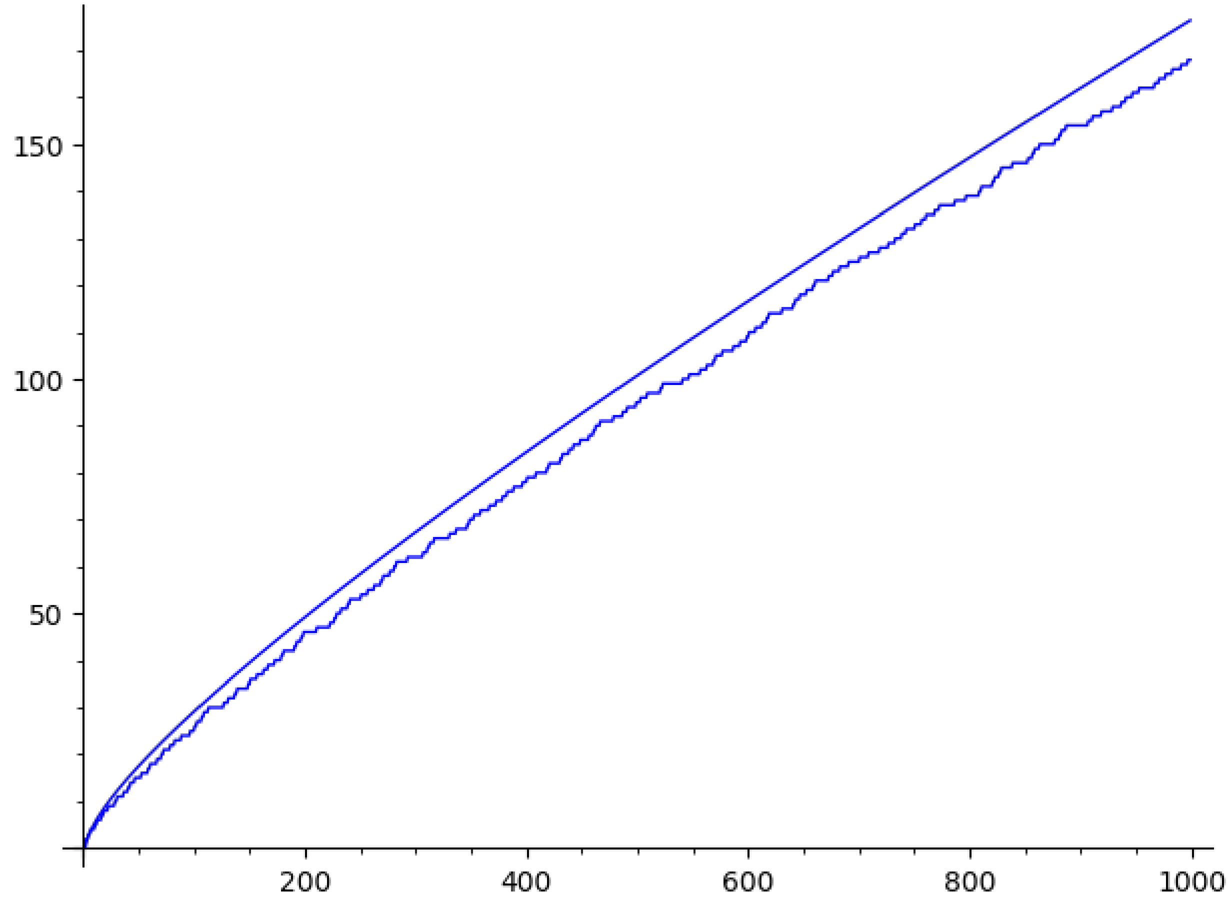
```
vectorUn=[(n, prime_pi(n)*ln(n)/n) for n in range(2,100000)]
point(vectorUn)
```



On remarque que la suite est convergente vers 1 comme la théorème des nombres premiers

4. Comparer π(n) à la fonction d'écart logarithmique intégrale Li(n)

```
vectorLi_n = [(n, integrate(1/ln(x), x, 2, n)) for n in
range(2,1000)]
line(vectorPrimePi) + line(vectorLi_n)
```



2.2 Nombres de Fermat

On a la formule de Fermat

```
var('x')
formuleFermat(x) = 2^(2^x) + 1
```

Afin de montrer qu'il avait tort, il est souffit de cherche un nombre dans la suite qui n'est pas un nombre de premier.

La fonction suivant cherche ce nombre:

```
def chercheNonPremier():
    n = 1
    while(1):
        if not is_prime(formuleFermat(n)):
            return [n, formuleFermat(n)]
        n += 1
```

```
chercheNonPremier()
```

```
[5, 4294967297]
```

Donc avec n = 5, la formule Fermat donne 4294967297, qui n'est pas un nombre permier

2.3 Nombres de Mersenne

On a la forme de nombre de Mersenne est Mp = 2^p − 1. Soit qu'il exist r,s tel que p = r.s.

On a Mp = 2^(r.s) - 1 = (2^r - 1)(2^(r.(s-1)) + 2^(r.(s-2)) + ... + 1). Si Mp est premier alors 2^r - 1 = 1 => r=1

Donc Mp est premier si p est premier.

1. Les nombres premiers inférieurs ou égaux à 257

La liste des nombres premiers inférieurs ou égaux à 257:

```
prime_range(258)
```

```
[2,
 3,
 5,
 7,
 11,
 13,
 17,
 19,
 23,
 29,
 31,
 37,
 41,
 43,
 47,
 53,
 59,
 61,
 67,
 71,
 73,
 79,
 83,
 89,
 97,
 101,
 103,
 107,
 109,
 113,
 127,
 131,
 137,
```

```
139,
149,
151,
157,
163,
167,
173,
179,
181,
191,
193,
197,
199,
211,
223,
227,
229,
233,
239,
241,
251,
257]
```

Donc le nombre de nombre premier inférieur à 257 est:

```
len(prime_range(258))
```

```
55
```

La liste de nombre de Mersenne correspondants:

```
var('p')
Mp(p) = 2^p - 1
```

```
listeMersenne257 = [Mp(p) for p in prime_range(258)]
```

2. Fonction de cherche les nombre Mersenne qui est premier:

```
def mersennePremier(n):
    liste =[]
    for p in prime_range(n):
        if is_prime(Mp(p)):
            liste.append(p)
    return liste
```

```
mersennePremier(258)
```

```
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127]
```

Et donc les nombres de Mersenne premiers correspondants:

```
[Mp(p) for p in mersennePremier(258)]
```

```
[3,
```

```
7,
31,
127,
8191,
131071,
524287,
2147483647,
2305843009213693951,
618970019642690137449562111,
162259276829213363391578010288127,
170141183460469231731687303715884105727]
```

Alors l'affirmation de Mersenne est correcte jusqu'à 31

3. Décomposer M41 et M47 en facteurs premiers

```
factor(2^41 - 1)
```

```
13367 * 164511353
```

```
factor(2^47 - 1)
```

```
2351 * 4513 * 13264529
```

4. Tester nombre parfait

```
def testerParfait(n):
    for p in mersennePremier(n):
        nombreParfait = 2^(p-1) * (2^p - 1)
        if 2*nombreParfait != sum(divisors(nombreParfait)):
            return false
    return true
```

```
testerParfait(258)
```

```
True
```

2.4 Un test de primalité pour les nombres de Mersenne

La fonction testLucas(s) va tester si n = 2^s - 1 est un nombre Mersenne premier. Si n est premier, elle va retourner n, sinon elle retourne 0

```
def testLucas(s):
    L = 4
    n = 2^s - 1
    for i in range(2,s):
        L = L^2 - 2
    if L%n == 0:
        return n
    else:
        return 0
```

Le nombre premier plus grand que nous trouvé est:

```
testLucas(31)
```

    2147483647

- Comparer avec is_prime(), la performance de testLucas() est beaucoup moins bien

```
is_prime(2147483647)
```

    True

3 Algorithmes d’exponentiation

3.1 Naif itératif et naif récursif

- Fonction itérative

```
def exponentIteratif(x, n):
    if n==0:
        return 1
    result = x
    for i in range(1,n):
        result = result * x
    return result
```

- Fonction recursive

```
def exponentRecursif(x,n):
    if (n==0):
        return 1
    else:
        return x * exponentRecursif(x, n-1)
```

Les 2 fonctions ont la meme complexité

3.2 Dichotomique itératif et dichotomique récursif

- Fonction itérative

```
def dichotomieIteratif(x, n):
    if n==0:
        return 1
    y = 1
    while (n >= 1):
        if (n%2 == 0):
            x = x * x
            n = n/2
        else:
```

```
            y = x * y
            x = x * x
            n = (n-1)/2
    return y
```

- Fonction recursive

```
def dichotomieRecursif(x, n):
    if n == 0:
        return 1
    if (n%2 == 0):
        return dichotomieRecursif(x*x, n/2)
    else:
        return x*dichotomieRecursif(x*x, (n-1)/2)
```

Les 2 fonctions ont la meme complexité

3.3 Algorithme d’exponentiation modulaire

```
def powerModulo(x, n, N):
    return dichotomieRecursif(x, n)%N
```

```
powerModulo(5,10000,6)
```

    1

```
power_mod(5,10000,6)
```

    1

La performance de 2 fonctions est semble le meme

4 Cryptosystème RSA

4.2 L’ensemble des messages, codage, décodage

- Fonction numerise: pour numériser une suite binaire en des entiers entre 0 et N-1, donc il faut découper la suite en plussieur segment, chaque segment est de longeur L tel que 2^L - 1 < N-1.

```
def numerise(original, N):
    S3 = BinaryStrings()
    suiteBinaire = S3.encoding(original)
    L = floor(log(N, 2))
    suiteInteger = []
    pointer = 0
    numberSegment = len(suiteBinaire)/L
    if len(suiteBinaire)%L != 0:
        longeurDernierSegment = len(suiteBinaire)%L
```

```
        numberSegment += 1
    else:
        longeurDernierSegment = L
    for i in range(numberSegment):
        suiteInteger.append(int(str(suiteBinaire[pointer:(i+1)*L]),
base = 2))
        pointer = (i+1)*L
    #ajoute le longeur de dernier segment à la fin de la suite pour
l'aide le déchiffre
    suiteInteger.append(longeurDernierSegment)
    return suiteInteger
```

- Fonction alphabetise:

```
from sage.crypto.util import bin_to_ascii
def alphabetise(numero, N):
    L = floor(log(N, 2))
    longeurDernierSegment = numero[-1]
    suiteBinaire = ''
    for i in range(len(numero)-2):
        suiteBinaire += format(numero[i], '0'+str(L)+'b')
    suiteBinaire += format(numero[-2],
'0'+str(longeurDernierSegment)+'b')
    return bin_to_ascii(suiteBinaire)
```

```
original="Que j'aime a faire connaitre un nombre utile aux sages ..."
numerise(original, 13213)
alphabetise(numerise(original, 13213), 13213)
```

```
    "Que j'aime a faire connaitre un nombre utile aux sages ..."
```

4.3 La génération de clés RSA

- La recommandation de la société RSA est le paire clefs doivent etre grand pour diffifcile à factoriser, p et q doivent etre choisit au hassard, N doit etre avoir au moins 30 chiffres.

- Le nombre de Mersenne doit etre évité parce que il peut se factoriser facilement.

- e doit etre grand pour qu'il est difficile à trouvé.

Pour definit la fonction cleRSA, on définit d'abord la fonction euclidEtendu pour cherche les coefficient de Bézout

```
def euclidEtendu(n , m ):
    if n == 0:
        return (m, 0, 1) if m >= 0 else (-m, 0, -1)
    else :
        g ,x ,y = euclidEtendu(m%n, n)
        return (g, y - (m//n ) * x, x)
```

```
def cleRSA(m):
    length = floor(m*random())
    p = floor(random()*10^length) + 10^length
```

```
    while(not is_prime(p)):
        p += 1
    q = floor(random()*10^(m+1-length)) + 10^(m+1-length)
    while(not is_prime(q)):
        q += 1
    N = p*q
    e = floor((p-1)*(q-1)*random()) -1
    while (gcd([(p-1)*(q-1), e]) != 1):
        e -= 1
    u = euclidEtendu(e, (p-1)*(q-1))[1]
    d = u + floor(random()*100)*(p-1)*(q-1)
    return [N, e, d]
```

```
[N, e, d] = cleRSA(5)
[N, e, d]
```

```
[3722242, 62071, 116806471]
```

```
power_mod(power_mod(100, e, N), d, N)
```

```
100
```

4.4 Fonctions de chiffrement et déchiffrement RSA

- Fonction chiffrer

```
def chiffrer(N,e, message): return [power_mod(n,e,N) for n in
message]
```

- Fonction dechiffrer

```
def dechiffrer(N,d, code): return [power_mod(n,d,N) for n in code]
```

On a donc les 2 fonctions chiffrer et dechiffrer faitent la meme chose

4.5 Signature avec RSA

1. Protocole 1 : message+signature

```
def protocole1(m,s,N1,cle1,N2,cle2): return [chiffrer(N2,cle2,m),
chiffrer(N1,cle1,s)]
```

```
[N1, e1, d1] = cleRSA(10)
[N2, e2, d2] = cleRSA(20)
message = "Que j'aime a faire connaitre un nombre utile aux sages
..."
signature = "Alice"
```

```
numeriseMessage = numerise(message, N2)
numeriseSignature = numerise(signature, N1)
encrypted = protocole1(numeriseMessage,numeriseSignature,N1,d1,N2,e2)
decrypted = protocole1(encrypted[0] , encrypted[1] ,N1,e1,N2,d2)
[alphabetise(decrypted[0], N2), alphabetise(decrypted[1], N1)]
```

```
["Que j'aime a faire connaitre un nombre utile aux sages ...",
'Alice']
```

2. Protocole 2 : message signé

```
def protocole2(m,N1,d1,N2,e2):
    if(N1>N2):
        return chiffrer(N1,d1, chiffrer(N2,e2,m))
    else:
        return chiffrer(N2,e2, chiffrer(N1,d1,m))
```

```
if (N1>N2) :
    numeriseMessage = numerise(message, N2)
else:
    numeriseMessage = numerise(message, N1)
encrypted = protocole2(numeriseMessage,N1,d1,N2,e2)
decrypted = protocole2(encrypted,N2,d2,N1,e1)
if (N1>N2) :
    m = alphabetise(decrypted, N2)
else:
    m = alphabetise(decrypted, N1)
```

On consiedere 2 cas pour la protocole 2 parce que pour effectuer 2 chiffrer sur Z/N_AZ puis Z/N_BZ on a besoin Z/N_AZ est inclut dans Z/N_BZ ou verso. On a pas considere le cas $N_A = N_B$ parce que c'est le cas quand on effectuer l'operation exponentiel modulo sur un meme anneau.