COMPLEX - Projet

DE MATOS Kevin LEWANDOWSKI Basile

11 décembre 2020

Exercice 1

a. Calcul du pgcd selon l'algorithme d'Euclide : Pour chaque itération, on effectue "a mod b"
- Si le résultat est différent de 0, "a" prend la valeur "b" et "b" la valeur "a mod b"
- Si le résultat est égal à 0, on retourne la valeur de "b" correspondant au pgcd(a,b)

```python
def my_gcd(a,b):
     while(a%b != 0) :
        a, b = b, a%b
     return b
```

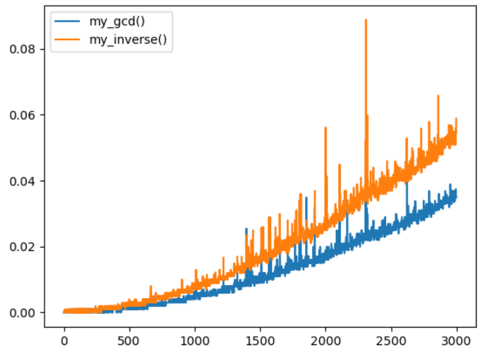
b. Calcul de l'inverse du modulo N de a selon l'algorithme d'Euclide étendu :
Utilisation de l'identité de Bézout avec $ab + Nc = PGCD(a, N)$
Initialisation avec une matrice identité pour les valeurs de b,b1,c,c1
Retourne un entier b tel que $a*b \equiv 1 \bmod N$ ssi $pgcd(a,N) = 1$

```python
def my_inverse(a,N):
    b,b1 = 1,0
    c,c1 = 0,1
    while N!=0:
        temp_quotient, temp_reste = a//N, a%N
        a, N = N, temp_reste
        b, b1 = b1, (b - temp_quotient*b1)
        c, c1 = c1, (c - temp_quotient*c1)

    # a => pgcd(a,N)
    # b => a*b ≡ pgcd(a,N) mod N
    # c => N*c ≡ pgcd(a,N) mod a

    if a!=1:
        raise ValueError("a n'est pas inversible modulo N!")
    else:
        return b
```

c.



d.

```python
def my_expo_mod(g,n,N):
    l = n.bit_length()
    h = 1
    for i in range(l,-1,-1):
        h = (h*h)%N
        if((n & (2**i)) == 2**i):
            h = (h*g)%N
    return h
```

Exercice 2

a.

```python
    def first_test(n):
        for a in range(2, int(ma.sqrt(n) + 1)):
            if n % a == 0:
                return False
        return True
```

b. En considérant que la divison euclidienne a une complexité arithmétique en $O(1)$, le test naïf de primalité à une complexité en $O(\sqrt{n})$.

c. On compte ainsi 9680 nombres premiers inférieurs à 10^5.

d.

```python
def gen_carmichael(t):
    res = []
    for x in range(3, int(t), 2):  # les nombres de Carmichael sont impairs
        valid = False
        for y in range(2, x):
            if ma.gcd(x, y) == 1:
                if pow(y, x-1, x) != 1:
                    valid = False
                    break
            else:
                valid = True
        if valid:
            res.append(x)
    return res
```

e.

```
def gen_carmichael_3(k):
    prime = []
    for n in range(3, 2 ** k, 2):
        if first_test(n):
            prime.append(n)
    res = []
    for i_a, a in enumerate(prime):
        for i_b, b in enumerate(prime[:i_a]):
            ab = a * b
                for c in prime[:i_b]:
                # on  a obtenu 3 premiers, on teste si leur produit est Carmichael
                tst = ab * c - 1
                if ma.ceil(ma.log2(tst)) != k:
                    continue
                if tst % 2 == 0 and tst % (a - 1) == 0 and tst % (b - 1) == 0
                  and tst % (c - 1) == 0 and a % (b * c) != 0:
                    res.append(tst + 1)
    return sorted(res)
```

f. On compare le nombre de Carmichael maximal obtenu par plusieurs algorithmes :

Algorithme naïf :	63973 en 287s
Algorithme naïf multithreadé :	115 921 en 317s
Algorithme basé sur le critère de Korselt :	1 196 508 858 409 en 302s

L'algorithme `gen_carmichael_3` est donc significativement plus efficace. Cependant, il ne retourne pas tous les nombres de Carmichael puisqu'il ne s'intéresse qu'à ceux qui se décomposent en 3 facteurs premiers. Dans cet exemple, il y a 8314 nombres inférieurs au maximum (la majorité) qui n'ont pas été pris en compte.

g.

1 Soit n un nombre de Carmichael de la forme pqr avec p < q < r trois nombres premiers. D'après le critère de Korselt on a :
$\exists\, \lambda \in \mathbb{N}$ tel que :

$$\begin{aligned}
\lambda(r-1) &= n-1\\
\lambda(r-1) &= pqr-1\\
\lambda(r-1) &= pq(r-1)+pq-1\\
(r-1)(\lambda-pq) &= pq-1
\end{aligned}$$

Soit $h := \lambda - pq$ on a :
- $h \neq 1$ car cela impliquerait $pq = r$ et n'aurait un facteur carré (contraire au critère de Korselt).
- Puisque $r-1 > q$ on a $hq < pq$ donc $h < p$ et plus précisement $h \leq p-1$.
D'où : $h \in [\![2; p-1]\!]$

2 De la même manière on peut montrer : $\exists\, k \in [\![p+1; r-1]\!],\ k(q-1) = pr-1$

$$\begin{aligned}
h(r-1) &= pq-1\\
h(r-1) &= p(q-1)+p-1\\
h(pr-1+1-p) &= p(p(q-1)+p-1)\\
h(k(q-1)+1-p) &= p^2(q-1)+p(p-1)\\
hk(q-1)+h(1-p) &= p^2(q-1)p(p-1)\\
(p^2+hk)(q-1) &= (h+p)(p-1)
\end{aligned}$$

3 Ainsi, on a montré que l'on pouvait exprimer q-1 en fonction de p, h et k. On peut également observer d'après (1) qu'on a : $r \leq \frac{1}{2}(pq+1)$, ce qui nous permet de montrer que les diviseurs premiers de n sont bornés proportionellement à p.
En effet, soit $p \in \mathbb{N}$, on a : $q-1 \leq (2p-1)(p-1)$ donc $q \leq 2(p^2+1)$
D'où $r \leq p(p^2+1)+\frac{1}{2}$ ce qui montre que r est borné, c'est à dire qu'il n'existe qu'un nombre fini de nombres de Carmichael de la forme $p*q*r$ pour p fixé.

h.
Pour p = 3 on obtient : 561
Pour p = 5 on obtient : 1105, 2465, 10585

Exercice 3

a.

```
def test_fermat(n, a=0):
    if not a:
        a = rd.randint(2, 100)

    if pow(a, n - 1, n) != 1:
        return False
    return True
```

c. Probabilités empiriques d'erreurs sur `test_fermat` avec une base aléatoire pour des entiers inférieurs à 10^5 sur des échantillons de tailles 10^5 :

Nombres de Carmichael :	0.9375
Nombres Composés :	0.00161
Nombres aléatoires :	0.00132

Exercice 4

a.

```
def test_miller_rabin(n, k=3):
    h, m = 0, n - 1
    while m % 2 == 0:
        h += 1
        m //= 2
    for _ in range(k):
        a = random.randrange(2, n - 1)
        b = pow(a, m, n)
        if b == 1 or b == n - 1:
            continue
        for _ in range(h - 1):
            if b != n-1 and pow(b, 2, n) == 1:
                return False
            if b == n - 1:
                break
            b = pow(b, 2, n)
        if b != n - 1:
            return False
    return True
```

c. Probabilités empiriques d'erreurs sur `test_miller_rabin` avec une précision T= 3 pour des entiers inférieurs à 10^5 sur des échantillons de tailles 10^7 :

Nombres de Carmichael :	0.0008469
Nombres Composés :	0.0000029
Nombres aléatoires :	0.0000018

Le nombre d'erreur (18 pour 10 millions de tests en nombres aléatoires) n'est pas suffisamment significatif pour conclure quant à l'impact de la primalité des nombres testés sur l'algorithme ; on peut néanmoins remarquer qu'il est plus fiable que l'algorithme du test de Fermat de l'exercice précédent.

d.

```
def gen_rsa(t):
    res = []
    for n in range(2 ** (t - 1), 2 ** t):
        if test_miller_rabin(n):
            res.append(n)
            if len(res) == 2:
                break
    if len(res) < 2:
        raise ValueError("Intervalle trop court")
    return res[0] * res[1]
```