TP1

August 25, 2022

0.1 TP 1 : addition, multiplication, écriture en base b

0.1.1 Exo 1 (écriture binaire)

[1]:
```
def binaire(n):
    L=[]
    while n!=0:
        q,r=n//2, n%2
        # on ajoute le bit calculé en fin de liste
        L.insert(0,r)
        # puis on passe au suivant
        n=q
    return(L)
```

[2]:
```
binaire(214)
```

[2]:
```
[1, 1, 0, 1, 0, 1, 1, 0]
```

[3]:
```
2**7+2**6+2**4+2**2+2
```

[3]:
```
214
```

[4]:
```
bin(214)
```

[4]:
```
'0b11010110'
```

0.1.2 Exo 2 (exponentiation rapide)

[5]:
```
def puiss1(x,n):
    r=1
    b=binaire(n)
    for i in range(len(b)-1,-1,-1):
        if b[i]==1:
            r=r*x
        x=x**2
    return r
```

```
def puiss2(x,n):
    r = 1
    while n > 0 :
        if n % 2 == 1 :
            r = r*x
        x = x**2
        n = n//2
    return r

def puiss3(x,n):
    if n==0:
        return 1
    if n%2==0:
        return puiss3(x**2,n//2)
    return x*puiss3(x**2,(n-1)//2)

print(puiss1(3,13))
print(puiss2(3,13))
print(puiss3(3,13))
```

```
1594323
1594323
1594323
```

0.1.3 Exo 3 (opérations binaires)

[6]:
```
# addition a+1 en binaire

def add1(a):
    r=a # initialisation du resultat
    c=1 # c est la retenue
    for i in range(len(r)-1,-1,-1):
        if c==1:
            if r[i]==0 :
                r[i]=1
                c=0
            else :
                r[i]=0
    if c==1:
        r.insert(0,1)
    return(r)
```

[7]:
```
r=add1([1,1,1,1])
print(r)
```

```
[1, 0, 0, 0, 0]
```

[8]:
```
# addition a + b en binaire

def add(a,b):
    # on commence par mettre a et b de la meme taille en completant avec des 0
    la=len(a); lb=len(b)
    if la<lb :
        a=[0 for _ in range(lb-la)]+a
    if lb<la :
        b=[0 for _ in range(la-lb)]+b
    l=max(la,lb)
    r=[0 for _ in range(l)]   #initialisation du resultat
    c=0  # c est la retenue
    for i in range(l-1,-1,-1):
        if c==0:
            if a[i]==1 and b[i]==1:
                r[i]=0 ; c=1
            elif a[i]==0 and b[i]==0:
                r[i]=0; c=0
            else:
                r[i]=1; c=0
        else :
            if a[i]==1 and b[i]==1:
                r[i]=1; c=1
            elif a[i]==0 and b[i]==0 :
                r[i]=1; c=0
            else :
                r[i]=0; c=1
    if c==1: # si retenue finale vaut 1, on ajoute un 1 en début
        r.insert(0,1)
    return(r)
```

[9]:
```
add([1,1,0,0,1],[1,1,0,1])
```

[9]:
```
[1, 0, 0, 1, 1, 0]
```

[10]:
```
bin(0b11001+0b1101)
```

[10]:
```
'0b100110'
```

[11]:
```
def mul2(a):
    # pour multiplier par 2, il suffit de rajouter un 0 a droite, sauf si a=0
    if a!=[0]:
        a.append(0)
    return(a)


def mul(a,b):
```

```
    lb=len(b)
    r=[0] # initialisation du resultat
    for i in range(lb-1,-1,-1):
        if b[i]==1:
            r=add(r,a)
        a.append(0)  # a = 2*a
    return(r)

mul([1,1,0,1],[1,0,1])
```

[11]: [1, 0, 0, 0, 0, 0, 1]

0.1.4 Exo 4 (écriture b-adique)

[12]:
```
def Badic(A,B):
    L=[]; Q=A
    while Q != 0 :
        # division eucidienne de A par B
        (Q,R)= Q.quo_rem(B)
        # on ajoute le reste calculé en début de liste
        L.insert(0,R)
    return(L)

R=PolynomialRing(QQ,'X')
X=R.gen()
A=X**5+3*X**4+5*X**3-X**2+1
B=X**2-3*X+1
Badic(A,B)
```

[12]: [X + 9, 48*X + 50, 155*X - 58]

[13]:
```
def BadicRapide(A,B):
    a=A.degree(); b=B.degree()
    if a<b :
        return([A])
    # Cherche plus petite puissance de 2 tq 2^p deg(B) > deg(A)
    k=1
    while k*b <= a:
        k=2*k
    Q,R=A.quo_rem(puiss1(B,k//2))
    LQ=BadicRapide(Q,B)
    LR=BadicRapide(R,B)
    # on a deg(R)< (k//2)deg(B), donc LR de taille <= k//2. On complete LR avec␣
 ↪des zéros au début
    # pour qu'elle soit de taille exactement k//2 avant de la consaténer à LQ␣
 ↪(sinon y a un souci de taille totale de liste).
```

```
    for i in range(k//2-len(LR)):
        LR.insert(0,0)
    LQ.extend(LR)
    return(LQ)
```

[14]:
```
## Tests sur polynôme aléatoires de F_7[x]

F=GF(7)['X']
X=F.gen()
A=F.random_element(degree=10002)
B=X^3+5*X-1

%time L1=Badic(A,B)
%time L2=BadicRapide(A,B)
print(L1==L2)

## Remarque : pour utiliser deux fois %time dans une même cellule,
## il faut écrire les instructions en ligne, à la suite de %time.
## Sinon, utiliser deux cellules différentes.
```

```
CPU times: user 191 ms, sys: 0 ns, total: 191 ms
Wall time: 190 ms
CPU times: user 30.3 ms, sys: 0 ns, total: 30.3 ms
Wall time: 30 ms
True
```

[15]:
```
def Developpe(L,B):
    l=len(L)
    A=sum(L[l-i-1]*B**i for i in range(l))
    return (A)

def DeveloppeRapide(L,B):
    l=len(L)
    if l<=1 :
        return(L[0])
    # Cherche plus petite puissance de 2 tq 2^p > l
    k=1
    while k < l :
        k=2*k
    # Coupe la liste en deux : on garde les l-k/2 derniers elements pour le reste
    Q=DeveloppeRapide(L[:k//2],B)
    R=DeveloppeRapide(L[k//2:],B)
    Res=Q*puiss2(B,l-k//2)+R
    return(Res)


A=F.random_element(degree=10000)
```

```
B=X^3+5*X-1
L=Badic(A,B)
%time A1=Developpe(L,B)
%time A2=DeveloppeRapide(L,B)
print(A==A1)
print(A==A2)
```

```
CPU times: user 607 ms, sys: 0 ns, total: 607 ms
Wall time: 607 ms
CPU times: user 16 ms, sys: 2.6 ms, total: 18.6 ms
Wall time: 18.4 ms
True
True
```

0.1.5 Exo 5 (multiplication de Karatsuba)

[16]:
```
def Naif(A,B):
    LA=A.list()
    LB=B.list()
    n=len(LA)
    m=len(LB)
    R=0
    for k in range(m+n-1):
        s=0
        for i in range(k+1):
            if i<n and k-i < m:
                s=s+LA[i]*LB[k-i]
        R=R+s*X^k
    return R
```

[17]:
```
def Kar(A,B):
    n=max(A.degree(),B.degree())+1
    if n<=1 :
        return A*B
    # Cherche plus petite puissance de 2 tq 2^k > max(deg(B),deg(A)), soit  2^k>= n
    p=1
    while p < n:
        p=2*p
    A1,A0=A.quo_rem(X^(p//2))    # Gratuit en pratique
    B1,B0=B.quo_rem(X^(p//2))
    C1=Kar(A0,B0)
    C2=Kar(A1,B1)
    C3=Kar(A0+A1,B0+B1)-C1-C2
    return(C1+C3*X^(p//2)+C2*X^p)
```

[18]:
```
A=F.random_element(degree=10)
B=F.random_element(degree=10)
print(Naif(A,B))
print(Kar(A,B))
```

```
4*X^20 + 6*X^19 + 2*X^18 + 5*X^17 + 6*X^16 + 3*X^15 + 5*X^14 + 4*X^13 + 5*X^12 +
6*X^11 + 3*X^10 + 6*X^9 + X^8 + 6*X^7 + 5*X^6 + 4*X^5 + 6*X^4 + X^3 + 3*X^2 +
5*X + 4
4*X^20 + 6*X^19 + 2*X^18 + 5*X^17 + 6*X^16 + 3*X^15 + 5*X^14 + 4*X^13 + 5*X^12 +
6*X^11 + 3*X^10 + 6*X^9 + X^8 + 6*X^7 + 5*X^6 + 4*X^5 + 6*X^4 + X^3 + 3*X^2 +
5*X + 4
```

Comparaisons des temps

[19]:
```
import time

def tempsKar(N):
    A=F.random_element(degree=N)
    B=F.random_element(degree=N)
    t0=time.time()
    Kar(A,B)
    t1=time.time()
    return(t1-t0)

def tempsNaif(N):
    A=F.random_element(degree=N)
    B=F.random_element(degree=N)
    t0=time.time()
    Naif(A,B)
    t1=time.time()
    return(t1-t0)

def tempsSage(N):
    A=F.random_element(degree=N)
    B=F.random_element(degree=N)
    t0=time.time()
    C=A*B
    t1=time.time()
    return(t1-t0)
```

[20]:
```
S=[tempsNaif(100),tempsNaif(1000),tempsNaif(10000)]
T=[tempsKar(100),tempsKar(1000),tempsKar(10000)]
U=[tempsKar(100),tempsKar(1000),tempsKar(10000)]
print(S,T,U)
```

```
[0.003617525100708008, 0.2376995086669922, 25.714162826538086]
[0.005543708801269531, 0.1648542881011963, 9.461389064788818]
[0.005448341369628906, 0.16524696350097656, 9.460150241851807]
```

On constate que, comme prévu, Karatsuba devient plus efficace que l'algo naïf pour des grandes valeurs de n.

On note au passage que la complexité de Sage est très comparable à celle de l'algorithme de Karatsuba (c'est d'ailleurs probablement la méthode utilisée).

Comparons la croissance de complexité de Karatsuba de N=1000 à N=10000 avec la complexité théorique $N^{log_2(3)}$:

[21]:
```
c=float(log(3,2))
print(T[2]/T[1],10^c)
```

```
57.39243530614694 38.45585757936911
```

C'est assez convaincant du point de vue des ordres de grandeurs.