TP12019-corrigé

September 18, 2019

1 L3 Algèbre effective — TP 1 du mardi 17 septembre 2019

1.1 Pgcd, algorithme d'Euclide et relations de Bezout

1.2 I La suite de Fibonacci

On rappelle que la suite de Fibonacci est la suite linéaire récurrente définie par

$$u_0 = 1 \qquad u_1 = 1 \qquad \text{et} \qquad \forall n \geq 2,\ u_n = u_{n-1} + u_{n-2}.$$

Pour évaluer le 100ème terme de la suite, il est tentant d'implémenter la suite de manière récursive.

Faîtes le. Que remarquez-vous? Pourquoi?

[0]:
```
```

[38]:
```
def fib(n):
    if n==0:
        return 1
    elif n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

[39]:
```
fib(10)
```

[39]:
```
89
```

Le calcul de `fib(100)` est trop long: en effet, l'implémentation récursive est ici particulièrement inappropriée puisque l'on refait beaucoup de fois les mêmes calculs. Le nombre d'appels de la fonction `fib` pour calculer `fib(n)` est de l'ordre de 2^{n-2}. Pour $n = 100$, c'est trop grand!

[40]:
```
2^100
```

[40]:
```
1267650600228229401496703205376
```

On implémente donc de manière iétrative: on calcule tous les termes de la suite de u_0 jusqu'à u_n en ne gardant en mémoire que les deux derniers termes.

[41]:
```
def fib_it(n):
    if n==0:
        return 1
    elif n==1:
        return 1
    else:
        i=1
        a=1
        d=1
        while (i<n):
            i=i+1
            aux=a
            a=d
            d=d+aux
        return d
```

[42]:
```
fib_it(100), fib_it(10000)
```

[42]:
```
(573147844013817084101,
 54438373113565281338734260993750380135389184554695967026247715841208582865622349017083051547938960541173822675978026317384359584751116241439174702642959169925586334117906063048089793531476108466259072759367899150677960088306597966641965824937721800381441158841042480997984696487375337180028163763317781927941101369262750979509800713596718023814710669912644214775254478587674568963808002962265133111359929762726679441400101575800043510777465935805362502461707918059226414679005690752321895868142367849593880756423483754386342639635970733756260098962462668746112041739819404875062443709868654315626847186195620146126642232711815040367018825205314845875817193533529827837800351902529239517836689467661917953884712441028463935449484614450778762529520961887597272889220768537396475869543159172434537193611263743926337313005896167248051737986306368115003088396749587102619524631352447499505204198305187168321623238385979462724591977145462821839969578922379891219943177546970521613108109655995063829726125384824200789710905475402843814961193046506186617012298328896435273375079278606944476185352514442107792804597990456129812942380915605503303233891960916223669875992278292319189668801771857555552099465332012844650237115371514174929091310489720345557750719664542523286202201950609148358522388271101670843305116994211577515125551025165593188816404834412955703882547752111157739578011586839707260256561482495646053870028033131186148539980539703155572752969339958607985038158144627643385882852953580342485084542644647168153100153318047956743639681565332615250957112748041192819602214884914828438912417852017450730553892871785792350941774338333150689823935442198880542933244037119486721554357654856549913451927109891980266518456492782782721295764924023550759555820564756936539487331765900020637312657064350970948264971003873351747771340331902810557566793178947002411880309460403436295347199746139227479154973035641263307423082405199999610154978466734045832685296038830112076562924599813625165234709396304973404644510636530416363082366924225776146828846179184322479343440607991788336067684671185597501)
```

On peut alors aller loin dans le calcul des termes...

1.3 II -Pgcd

[0]:

a) Trouver dans l'aide la fonction Sage qui permet de calculer le pgcd de deux entiers.

Faire quelques tests: pgcd(5,6)=? pgcd(0,6)=? pgcd(0,0)=?

En Sage, la fonction `gcd` permet de calculer le pgcd de deux entiers. On obtient l'aide associée en tapant `help(gcd)` ou bien `gcd?`.

[43]:
```
gcd(5,6)
```

[43]:
```
1
```

[2]:
```
gcd?
```

[2]:
```
1
```

Un premier objectif de ce TP est de reprogrammer cette fonction vous même, en utilisant l'algorithme d'Euclide.

b) Quelle fonction permet en Sage de calculer le reste dans une division euclidienne? Et le quotient?

Quelle est la division euclidienne de 1234 par 7?

On calcule le reste avec `%` et le quotient avec `//`.

[0]:

[44]:
```
1234%7
```

[44]:
```
2
```

[45]:
```
1234//7
```

[45]:
```
176
```

[46]:
```
1234==(1234//7)*7+(1234%7)
```

[46]:
```
True
```

Pour $0 \leq m \leq n$, quelle est la division euclidienne de m par n?

Quelle relation y a-t-il entre la division euclidienne de m par n et celle de $m+n$ par n?

En utilisant les remarques ci-dessus, écrivez un programme **récursif** qui calcule la division euclidienne du premier argument par le second (s'il est non nul).

Pour $0 \leq m \leq n$, la division euclidienne de m par n est

$$m = 0 \cdot n + m.$$

Si $m = qn + r$ est la division euclidienne de m par n, alors $m + n = (q + 1) \cdot n + r$ est la division euclidienne de $m + n$ par n. On en déduit le programme récursif ci-dessous.

[47]:
```
def div(m,n):
    if (n==0):
        return "Division par zero"
    elif (m<n):
        return([0,m])
    else:
        aux=div(m-n,n)
        return(aux[0]+1,aux[1])
```

[48]:
```
div(512,3)
```

[48]: (170, 2)

c) Compléter la fonction ci-dessous pour qu'elle calcule le pgcd des arguments. Plusieurs approches sont possibles, dont:

1) Une boucle
2) Une fonction récursive

On présente les deux variantes.

[49]:
```
def pgcd(m,n):
    a,b=m,n
    while(b<>0):
        aux=b
        b=a%b
        a=aux
    return a
```

[50]:
```
def pgcdRec(m,n):
    if (n==0):
        return m
    else:
        return pgcdRec(n,m%n)
```

[53]:
```
pgcd(0,5)
```

[53]: 5

[0]:
```

```

d-1.) Pour évaluer la "complexité" du calcul du pgcd, on pourrait commencer par compter le nombre de divisions euclidiennes effectuées. Faîtes le.

On rajoute un compteur dans le programme précédent.

[55]:
```
def pgcd(m,n):
    compteur=0
    a,b=m,n
    while(b<>0):
        compteur=compteur+1
        aux=b
        b=a%b
        a=aux
    return(a, compteur)
```

[0]:
```
```

[56]:
```
pgcd(fib(10),fib(9))
```

[56]: (1, 9)

[57]:
```
pgcd(fib(10)-1,fib(9))
```

[57]: (11, 4)

[58]:
```
pgcd(fib(10)-2,fib(9))
```

[58]: (1, 7)

On remarque que le nombre d'itérations semble assez chaotique: pour des entiers très proches, cela varie, et pas du tout linéairement.

d-2) Une façon d'améliorer la complexité ci-dessus est de ne pas utiliser des divisions euclidiennes *stricto sensu*, mais des divisions avec restes possiblement négatifs. Remarquer que si $a>b>0$ sont des entiers, alors il existe une division avec reste:

$a=bq+r$, **avec** $|r| \leq b/2$

(Noter qu'ici, r est potentiellement négatif!) et qu'alors, on a $pgcd(a,b)=pgcd(b,|r|)$.

Utiliser cette remarque pour proposer un calcul du pgcd plus rapide que le précédent et comparer le nombre d'opérations.

On commence par coder le nouveau reste de la division euclidienne sophistiquée.

[67]:
```
def reste(m,n):
    if (n==0):
        return "Division par zero"
    elif abs(m)<= abs(n)//2:
        return m
    else:
```

```
        return reste(abs(m)-abs(n),n)
```

[68]:
```
reste(5,-1)
```

[68]:
```
0
```

On peut alors reprendre le même programme qu'avant, en utilisant juste la nouvelle fonction reste.

[88]:
```
def pgcd2(m,n):
    compteur=0
    a,b=m,n
    while(b<>0):
        compteur=compteur+1
        aux=b
        b=reste(a,b)
        a=aux
    return(a, compteur)
```

[76]:
```
pgcd2(fib(10),fib(9)),pgcd2(fib(10)-1,fib(9)),pgcd2(fib(10)-2,fib(9))
```

[76]:
```
((-1, 5), (11, 3), (1, 5))
```

La fonction pgcd2 a bien l'air meilleure. Pour tester plus, on tire au hasard quelques entiers et on compare les deux fonctions.

[93]:
```
for i in range(100):
    a=randint(1,10000)
    b=randint(1,10000)
    p1=pgcd(a,b)
    p2=pgcd2(a,b)
    print(p1[1]/(p2[1]+0.))
```

```
1.00000000000000
1.25000000000000
1.12500000000000
1.42857142857143
1.60000000000000
1.40000000000000
1.83333333333333
1.50000000000000
1.00000000000000
1.42857142857143
1.42857142857143
1.60000000000000
1.50000000000000
1.40000000000000
1.00000000000000
```

1.33333333333333
1.37500000000000
1.00000000000000
1.25000000000000
1.50000000000000
1.14285714285714
1.50000000000000
1.14285714285714
1.44444444444444
1.33333333333333
1.66666666666667
1.16666666666667
1.75000000000000
1.22222222222222
1.33333333333333
1.00000000000000
1.60000000000000
1.00000000000000
1.60000000000000
1.33333333333333
1.20000000000000
1.00000000000000
1.80000000000000
1.50000000000000
1.14285714285714
1.40000000000000
1.20000000000000
1.25000000000000
1.25000000000000
1.25000000000000
1.20000000000000
1.20000000000000
1.25000000000000
1.50000000000000
1.28571428571429
1.16666666666667
1.66666666666667
1.00000000000000
1.28571428571429
1.50000000000000
1.20000000000000
1.33333333333333
1.42857142857143
1.14285714285714
1.71428571428571
1.57142857142857
1.40000000000000
1.42857142857143

```
1.00000000000000
1.42857142857143
1.71428571428571
1.33333333333333
1.00000000000000
1.00000000000000
1.00000000000000
1.20000000000000
1.33333333333333
1.33333333333333
1.25000000000000
1.00000000000000
1.28571428571429
1.25000000000000
1.60000000000000
1.20000000000000
1.33333333333333
1.14285714285714
1.57142857142857
1.66666666666667
1.12500000000000
1.00000000000000
1.00000000000000
1.44444444444444
1.80000000000000
1.55555555555556
1.00000000000000
1.00000000000000
1.14285714285714
1.42857142857143
1.50000000000000
1.33333333333333
1.25000000000000
1.60000000000000
1.40000000000000
1.33333333333333
1.25000000000000
```

La fonction pgcd2 semble bien meilleure.

[0]:

[0]:

e) Implémenter une fonction *pgcdbin* utilisant l'algorithme binaire (faire une recherche si nécessaire).

[2]:
```
def pgcdbin(m,n):
    if (m==n):
        return m
    elif (m%2==0) and (n%2==0):
        return 2*pgcdbin(m/2,n/2)
    elif (m%2==0): # n est alors impair
        return pgcdbin(m/2,n)
    elif (n%2==0):
         return pgcdbin(m,n/2)
    elif (m>n):
        return pgcdbin((m-n)/2,n)
    else:
        return pgcdbin((n-m)/2,m)
```

[3]:
```
pgcdbin(45,15)
```

[3]:
```
15
```

[0]:
```

```

f) Quelle fonction en sage permet de trouver les coefficients d'une relation de Bézout?

Donner une relation de Bézout entre 779 et 123.

Les coefficients de Bezout se calculent en utilisant la fonction `xgcd`.

[0]:
```

```

[0]:
```
xgcd?
```

[12]:
```
B=xgcd(779,123)
```

[14]:
```
B[0]==779*B[1]+123*B[2]
```

[14]:
```
True
```

Compléter la fonction suivante pour qu'elle calcule une relation de Bézout entre les arguments.

Plusieurs approches sont possibles, dont 1) Calquer la méthode "humaine" qui consiste à remonter l'algorithme d'Euclide. Pour cela, il faut créer des listes en sage pour stocker les valeurs des quotients successifs dans l'algorithme d'Euclide 2) Si r est le reste de la division euclidienne de m par n, trouver un lien entre Bezout(m,n) et Bezout(n,r). En déduire une variante récursive.

Pour la première méthode, il faut créer la liste des quotients dans l'algorithme d'Euclide. Les listes sont codées par des suites entre crochets, par exemple `[1,2,3]`. La fonction `+` permet de concaténer deux listes. On fait alors la remontée en changeant les coefficients au fur et à mesure.

[0]:
```

```

[4]:
```
def Bezout(m,n):
    L=[]
    while(n<>0):
        aux=n
        n=m%n
        L=L+[m//aux]
        m=aux
    i=len(L)-2
    u=1
    v=-L[i]
    while(i>0):
        i=i-1
        aux=v
        v=u-L[i]*v
        u=aux
    return([u,v])
```

[5]:
```
Bezout(779,123)
```

[5]:
```
[1, -6]
```

Pour la méthode récursive, on remarque que si l'on injecte $m = nq + r \iff r = m - nq$ dans une relation de Bézout pour n et r, disons $un + vr = d$, on obtient une relation de Bezout pour m et n de la forme $d = un + v(m - nq) = vm + (u - nv)n$. D'où le programme récursif suivant:

[6]:
```
def BezRec(m,n):
    if (n==0):
         return [1,0]
    else:
        aux=BezRec(n,m%n)
        return [aux[1],aux[0]-m//n*aux[1]]
```

Testez votre fonction.

[0]:
```

```

[8]:
```
BezRec(779,123)
```

[8]:
```
[1, -6]
```

1.4 III - Ecriture triadique (=en base 3)

Ecrire une fonction *triad* qui renvoie (sous forme de liste) l'écriture d'un entier en base *3*.

[10]:
```
def triad(n):
    if (n<3):
```

```
        return [n]
    else:
        u=n%3
        return triad((n-u)/3)+[u]
```

[12]:
```
triad(23)
```

[12]:
```
[2, 1, 2]
```

[0]:

[0]: