Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
G. Dewaele, A. Troesch

TP n° 4 : Complément au TP 3

Avant d'aborder les exercices suivants, on traitera la fin du TP précédent.

Correction de l'exercice 1 – Les différents calculs ne posent pas de difficulté. Le calcul par les fractions continues nécessite d'être commencé par l'intérieur (ce qu'il y a tout au fond). Pour l'algorithme de Brown, on trouve le plus petit diviseur de α suivant a_{k-1} en lui ajoutant le reste de la division de $-a_{k-1}$ par α.

```
import math

def leibniz(n):
    si = -1
    den = -1
    S =0
    for k in range(n+1):
        den += 2
        si *= -1
        S += si / den
    return(4*S)

def leibniz2(n):
    S = 0
    for k in range(n+1):
        S += 1/((4*k+1)*(4*k+3))
    return(8*S)

def euler(n):
    S = 0
    for k in range(1,n+1):
        S += 1/(k**2)
    return math.sqrt(6*S)

def riemann(n):
    S = 0
    for k in range(1,n+1):
        S += n / (n**2+k**2)
    return(4*S)

def machin(n):
    S = 0
    u = 5
    v = 239
    si = -1
    for k in range(n+1):
        u /= 5**2
        v /= 239**2
        si *= -1
        S += si * (4 * u -v)/ (2*k+1)
    return(4*S)

def archimede(n):
```

```
    a = math.sqrt(1/4 - 1/16)
    p = 3
    for k in range(1,n+1):
        a = math.sqrt(a/4 + 1/8)
        p = p / (2*a)
    return(p)

def woon(n):
    # Valeur maximale n=512
    if n <= 512:
        a = 1
        S = 1
        for n in range(1,n+1):
            a = math.sqrt(1+S**2)
            S += a
        return(2**(n+1) / a)

def FractionsContinues(n):
    u = 1
    for k in range(n):
        u = 1 / (1/(n+1-k) +u)
    return(2+ 2/(1+u))

def brown(n):
    a = n
    for k in range(1,n):
        a += (-a) % (n-k)
    return(n**2/a)

for n in [10,100,1000,10000,100000]:
    print("\nPour␣n␣=␣{}:".format(n))
    print('Par␣Leibniz:␣{}'.format(leibniz(n)))
    print('Par␣Leibniz2:␣{}'.format(leibniz2(n)))
    print('Par␣Euler:␣{}'.format(euler(n)))
    print('Par␣Riemann:␣{}'.format(riemann(n)))
    print('Par␣Machin:␣{}'.format(machin(n)))
    print('Par␣Archimède:␣{}'.format(archimede(n)))
    print('Par␣Woon:␣{}'.format(woon(n)))
    print('Par␣les␣fractions␣continues:␣{}'.format(FractionsContinues(n)))
    print('Par␣Brown:␣{}'.format(brown(n)))
```

Voici les résultats obtenus :

```
Pour n = 10:
Par Leibniz: 3.232315809405594
Par Leibniz2: 3.0961615264636406
Par Euler: 3.04936163598207
Par Riemann: 3.0399259889071595
Par Machin: 3.1415926535897944
Par Archimède: 3.141592516692157
Par Woon: 3.1415914215112
Par les fractions continues: 3.1386384979458573
Par Brown: 2.9411764705882355

Pour n = 100:
Par Leibniz: 3.1514934010709914
Par Leibniz2: 3.136642188870299
```

```
Par Euler: 3.1320765318091053
Par Riemann: 3.131575986923127
Par Machin: 3.1415926535897944
Par Archimède: 3.141592653589793
Par Woon: 3.1415926535897927
Par les fractions continues: 3.1415545383352894
Par Brown: 3.0921459492888066

Pour n = 1000:
Par Leibniz: 3.1425916543395442
Par Leibniz2: 3.141093153121444
Par Euler: 3.1406380562059946
Par Riemann: 3.1405924869231225
Par Machin: 3.1415926535897944
Par Archimède: 3.141592653589793
Par Woon: None
Par les fractions continues: 3.141592262066432
Par Brown: 3.1390275292714316

Pour n = 10000:
Par Leibniz: 3.1416926435905346
Par Leibniz2: 3.1415426585893202
Par Euler: 3.1414971639472147
Par Riemann: 3.1414926519231177
Par Machin: 3.1415926535897944
Par Archimède: 3.141592653589793
Par Woon: None
Par les fractions continues: 3.1415926496639806
Par Brown: 3.141331981304049

Pour n = 100000:
Par Leibniz: 3.1416026534897203
Par Leibniz2: 3.1415876536398177
Par Euler: 3.141583104326456
Par Riemann: 3.1415826535731552
Par Machin: 3.1415926535897944
Par Archimède: 3.141592653589793
Par Woon: None
Par les fractions continues: 3.1415926535505245
Par Brown: 3.141526183050601
```

On remarquera la grande efficacité de la méthode de Machin. La méthode d'Archimède est aussi honorable, ainsi que la méthode de Woon, mais celle-ci ne peut pas être poursuivie au-delà de $n = 512$, car elle fait alors intervenir des quantités trop grandes.

Correction de l'exercice 2 –

1. La façon la plus naturelle de faire est de parcourir tous les entiers strictement positifs et strictement inférieurs à n, et de tester s'ils divisent n. Si oui, on les ajoute à la liste des diviseurs propres :

```python
def divpropre(n):
    """ Renvoie la liste des diviseurs propres de n """
    diviseurs = []
    for i in range(1,n):
        if n % i == 0:
            diviseurs.append(i)
    return diviseurs
```

Voici une version plus compact du même algorithme, utilisant la possibilité de définir une liste par compréhension :

```
def divpropre2(n):
    """ Renvoie la liste des diviseurs propres de n """
    return [i for i in range(1,n) if n % i == 0]
```

Remarquez l'utilisation des raw string pour décrire la fonction. Placé à cet endroit, le commentaire sera affiché lors de l'appel à l'aide (fonction `help`) pour la fonction définie. Ainsi, `help(divpropre)` affiche la page suivante :

```
Help on function divpropre in module __main__:

divpropre(n)
    Renvoie la liste des diviseurs propres de n
(END)
```

2. On recherche les nombres parfaits en calculant pour tout i la somme des diviseurs propres de i, et en la comparant à i. Si le nombre est parfait, on calcule ensuite la somme des inverses des diviseurs de i, en n'oubliant pas i parmi ces diviseurs.

```
def parfaits(N):
    """ Renvoie la liste des nombres parfaits inférieurs à N,
        et de la somme des inverses de leur diviseurs """
    parfaits = []
    for i in range(1,N+1):
        S = 0
        divi = divpropre(i)
        for d in divi:       # calcul de la somme des diviseurs propres
            S += d
        if S == i:           # cas où i est parfait
            T = 1 / i        # calcul de la somme des 1/d (y compris d=i)
            for d in divi:
                T += 1 / d
            parfaits.append((i,T))
    return parfaits
```

Remarquez que j'ai pris soin à ne pas calculer 2 fois la liste des diviseurs. Cela n'a en fait pas tellement d'importance ici, vu le faible nombre de nombres premiers, mais dans certaines situations, calculer plusieurs fois la même chose peut beaucoup ralentir le programme.

Pour $N = 10000$, on obtient :

```
[(6, 2.0), (28, 2.0), (496, 2.0), (8128, 2.0)]
```

3. Pour trouver les couples amicaux, pour toute première coordonnée m, on calcule la somme n de ses diviseurs propes. Le couple (m, n) est amical si la somme des diviseurs propres de n est alors égal à m.

```
def amicaux(N):
    """ Renvoie les couples amicaux (m,n) pour m <= N """
    ami = []
    for m in range(1,N+1):
        n = somme(divpropre(m))
        if somme(divpropre(n)) == m:
            ami.append((m,n))
    return ami
```

Pour $N = 10000$, on trouve :

```
[(6, 6), (28, 28), (220, 284), (284, 220), (496, 496), (1184, 1210), (1210, 1184),
(2620, 2924), (2924, 2620), (5020, 5564), (5564, 5020), (6232, 6368), (6368, 6232),
(8128, 8128)]
```

Il est normal de retrouver les nombres parfaits. Il est aussi normal d'observer une symétrie.

4. On se rend assez vite compte que la façon d'effectuer le calcule de la somme des diviseurs n'est pas assez efficace tel qu'on l'a écrit. On peut remarquer que si l'on connaît la décomposition primaire de $n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$, alors les diviseurs de n sont exactement les $p_1^{\beta_1} \cdots p_k^{\beta_k}$ avec $\beta_i \leqslant \alpha_i$. Ainsi, on trouve la somme de tous les diviseurs de la sorte :

$$\begin{aligned} S(n) &= \sum_{\beta_1=0}^{\alpha_1} \sum_{\beta_2=0}^{\alpha_2} \cdots \sum_{\beta_k=0}^{\alpha_k} p_1^{\beta_1} \cdots p_k^{\beta_k} \\ &= \left(\sum_{\beta_1=0}^{\alpha_1} p_1^{\alpha_1} \right) \cdots \left(\sum_{\beta_k=0}^{\alpha_k} p_k^{\alpha_k} \right) \\ &= \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1} - 1}{p_i - 1} \end{aligned}$$

On commence donc par calculer la décomposition primaire :

```
def decomposition(n):
    prem = [ ]      # liste des premiers divisant n
    valuation = [ ] # valuations correspondantes
    if n == 1:          # gestion des cas particuliers
        return prem, valuation
    if n == 2 or n == 3:
        return [n], [1]
    v = 0
    p = 2
    (m,r)= divmod(n,p) # gestion à part de la divisibilité par 2
                       # afin de pouvoir progesser ensuite de 2 en 2
                       # les autres nombres premiers étant impairs
    while r == 0 and 4 <= n:
        v += 1
        n = m
        (m,r) = divmod(n,p)
    if v > 0:
        prem.append(p)      # Si nécessaire, ajout de p
        valuation.append(v) # et de sa valuation dans les listes
    q = 2
    p = 3
    while p**2 <= n:        # Rebelotte avec les autres premiers
        if is_premier(p):   # en progressant de 2 en 2
            v = 0
            (m,r)= divmod(n,p)
            while (r == 0) and p*p <= n:
                v += 1
                n = m
                (m,r) = divmod(n,p)
            if v > 0:
                prem.append(p)
                valuation.append(v)
        q = p
        p += 2
    if n == q:
        valuation[-1] += 1
    else:
        prem.append(n)
        valuation.append(1)
    return prem, valuation
```

On obtient alors la somme des diviseurs propres (on passe la décomposition primaire en paramètre) :

```
def somme_diviseurs_propres(n):
    if n == 0:
        return 0
    else:
        prod = 1
        prem,val = decomposition(n)
        for i in range(len(prem)):
            p = prem[i]
            prod *= (p ** (val[i]+1)-1)//(p-1)
        return prod - n
```

On teste ensuite si un nombre est sociable faisant intervenir dans sa suite aliquote que des nombres inférieurs à 1000000. On calcule les sommes des diviseurs propres successives, jusqu'à ce qu'on obtienne une valeur déjà trouvée (si c'est celle de laquelle on est parti, c'est bon, sinon non) ou qu'on obtienne une valeur trop grande. Le principe des tiroirs nous assure la terminaison de cet algorithme.

L'ensemble pris en paramètre est l'ensemble de toutes les valeurs possibles auquelles on a enlevé les valeurs qu'on sait ne pas être dans une suite aliquote, ou être dans une suite aliquote déjà trouvée (donc les valeurs trouvées lors des précédents calculs) ; si on tombe sur une telle valeur, il est inutile de continuer.

```
def is_sociable(n, ens):
    """ Teste si la chaîne commencant par n est sociable et ne
    fait intervenir que des nombres <= 1000000 ;
    retourne True/False, ainsi que la chaîne obtenue"""
    li = [n]
    a = n
    while True:
        s = somme_diviseurs_propres(a)
        if s == n:
            return True, li, 0
        elif s <= 15000 and s not in ens:
            return False, li, -1
        elif s > 1000000:
            return False, li, -1
        elif (s in li) and (s != n):
            return False, li, li.index(s)
        else:
            li.append(s)
            a = s
```

On traite alors les éléments les uns après les autres. Pour cela, on a créé un ensemble des différentes valeurs à tester, et on retire à chaque étape non seulement la valeur testée, mais aussi toutes les valeurs trouvées dans la suite aliquote, afin de diminuer le nombre de suites aliquotes calculées. Si une chaîne se referme sur un élément autre que son premier élément, on la repère tout de suite : l'élément sur lequel la chaîne se referme est une suite aliquote. On récupère donc une chaîne aliquote en tronquant le début de la chaîne (cas du test de contrôle strictement positif).

```
def sociable(N):
    """ Renvoie toutes les suites sociables commencant par un nombre
    inférieur à N """
    atraiter = {i for i in range (N+1)}
    sociaux = [ ]
    while len(atraiter)>0:
        vrai, chaine, controle = is_sociable(atraiter.pop(), atraiter)
        if vrai:
            sociaux.append(chaine)
        if controle > 0 and chaine[controle] in atraiter:
```

```
            sociaux.append(chaine[controle:])
        atraiter = atraiter.difference(set(chaine))
    return sociaux
```

On trouve, pour N=15000

```
[0]
[6]
[28]
[220, 284]
[496]
[1184, 1210]
[2924, 2620]
[5020, 5564]
[14316, 19116, 31704, 47616, 83328, 177792, 295488, 629072, 589786, 294896, 358336,
  418904, 366556, 274924, 275444, 243760, 376736, 381028, 285778, 152990, 122410,
  97946, 48976, 45946, 22976, 22744, 19916, 17716]
[6232, 6368]
[8128]
[12496, 14288, 15472, 14536, 14264]
[10744, 10856]
[12285, 14595]
```

Le désordre apparent de ces résultats provient du passage par la structure `set`, qui fait perdre l'ordre naturel, mais est plus efficace pour les tests d'appartenance, et pour les suppressions de termes. Les listes de longueur 1 correspondent aux nombres parfaits, les listes de longueur 2 aux nombres amicaux. On observe qu'il existe des listes de nombres sociaux plus longues : on en trouve une de 5, et une de 28, ce qui constitue à l'heure actuelle la plus longue suite aliquote connue. Il est remarquable qu'elle ait été découverte au début du 20-ième siècle, à une époque où les ordinateurs n'existaient pas (par Paul Poulet).

Correction de l'exercice 3 – On commence par former la liste des nombres de Fibonacci non nuls et distincts inférieurs à un certain n :

```
def fibo(n):
    """ Création de la liste des nombres de fibonacci inférieurs ou
    égaux à n, sans la redondance du début """
    if n == 0:
        return([])
    if n == 1:
        return([1])
    fi = [1,2]
    suivant = fi[-1] + fi[-2]
    while suivant <= n:
        fi.append(suivant)
        suivant = fi[-1] + fi[-2]
    return(fi)
```

La décomposition de Zeckendorf se trouve en considérant la plus grand nombre de Fibonacci inférieur à n, puis en itérant en formant la différence, ou alors, ce qui revient au même, en gardant les nombres de Fibonacci inférieurs à n, en les sommant en commençant par les plus gros, en ne gardant dans la somme que ceux qui ne font pas grossir la somme au-delà de n.

```
def zeckendorf(n):
    zeck = []
    fi = fibo(n)
    fi.reverse()
    S = 0
    for F in fi:
```

```
        if S+F <= n:
            zeck.append(F)
            S += F
    zeck.reverse()
    return(zeck)
```

La stratégie gagnante, ainsi qu'on l'a vu en DM, consiste à tirer le plus petit nombre de Fibonacci de la décomposition de Zeckendorf. Ceci est possible initialement si ce nombre ne représente pas n tout entier (donc si n n'est pas un nombre de Fibonacci). Dans le cas contraire, on tire au hasard parmi les valeurs autorisées, en espérant que le joueur 2 n'adopte pas la stratégie gagnante (dans ce cas, l'ordinateur la récupère).

```
import random

def JeuAllumettes(n):
    reste = n
    tirermax = n-1
    while reste > 0:
        z = zeckendorf(reste)
        if z[0] <= tirermax:
            tirer = z[0]
        else:
            tirer = random.randint(1,tirermax)
        print('Il␣reste␣{}␣allumettes.␣Le␣joueur␣1␣peut␣en␣tirer␣de␣1␣à ␣{}'.format(reste,tirermax))
        print('Joueur␣1␣(ordinateur):␣Tire␣{}␣allumettes.'.format(tirer))
        reste -= tirer
        tirermax = 2*tirer
        if reste==0:
            print('Le␣joueur␣1␣gagne!')
            return
        print('Il␣reste␣{}␣allumettes.␣Le␣joueur␣2␣peut␣en␣tirer␣de␣1␣à ␣{}'.format(reste,tirermax))
        test = True
        while test:
            tirer2 = eval(input("Joueur␣2,␣combien␣tirez-vous␣d'allumettes?␣"))
            if (tirer2 <= 2* tirer) and (tirer2 >= 1):
                if tirer2 > reste:
                    print('Tirage␣impossible!')
                else:
                    test = False
            else:
                print('Tricheur!')
        reste -= tirer2
        tirermax = 2*tirer2
        if reste == 0:
            print('le␣joueur␣2␣gagne!')
            return
```