hypercube

May 18, 2022

1 Hypercube et autres exercices

Exercices autour de tableaux en plusieurs dimensions et autres exercices.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.1 Q1 - triple récursivité

Réécrire la fonction `u` de façon à ce qu'elle ne soit plus récurrente.

[2]:
```
def u(n):
    if n <= 2:
        return 1
    else:
        return u(n-1) + u(n-2) + u(n-3)

u(5)
```

[2]:
```
9
```

Le problème de cette écriture est que la fonction est triplement récursive et que son coût est aussi grand que la fonction elle-même. Vérifions.

[3]:
```
compteur = []

def u_st(n):
    global compteur
    compteur.append(n)
    if n <= 2:
        return 1
    else:
        return u_st(n-1) + u_st(n-2) + u_st(n-3)

u_st(5), compteur
```

[3]:
```
(9, [5, 4, 3, 2, 1, 0, 2, 1, 3, 2, 1, 0, 2])
```

La seconde liste retourne tous les n pour lesquels la fonction `u_st` a été appelée.

[4]:
```
def u_non_recursif(n):
    if n <= 2:
        return 1
    u0 = 1
    u1 = 1
    u2 = 1
    i = 3
    while i <= n:
        u  = u0 + u1 + u2
        u0 = u1
        u1 = u2
        u2 = u
        i += 1
    return u

u_non_recursif(5)
```

[4]: 9

1.2 Q2 - comparaison de listes

On considère deux listes d'entiers. La première est inférieure à la seconde si l'une des deux conditions suivantes est vérifiée :

- les n premiers nombres sont égaux mais la première liste ne contient que n éléments tandis que la seconde est plus longue,
- les n premiers nombres sont égaux mais que le $n+1^{\text{ème}}$ de la première liste est inférieur au $n+1^{\text{ème}}$ de la seconde liste

Par conséquent, si l est la longueur de la liste la plus courte, comparer ces deux listes d'entiers revient à parcourir tous les indices depuis 0 jusqu'à l exclu et à s'arrêter sur la première différence qui détermine le résultat. S'il n'y pas de différence, alors la liste la plus courte est la première. Il faut écrire une fonction `compare_liste(p,q)` qui implémente cet algorithme.

[5]:
```
def compare_liste(p, q):
    i = 0
    while i < len(p) and i < len(q):
        if p [i] < q [i]:
            return -1  # on peut décider
        elif p [i] > q [i]:
            return  1  # on peut décider
        i += 1                          # on ne peut pas décider
    # fin de la boucle, il faut décider à partir des longueurs des listes
    if len (p) < len (q):
        return -1
    elif len (p) > len (q):
        return  1
    else :
        return 0

compare_liste([0, 1], [0, 1, 2])
```

[5]: -1

[6]:
```
compare_liste([0, 1, 3], [0, 1, 2])
```

[6]: 1

[7]:
```
compare_liste([0, 1, 2], [0, 1, 2])
```

[7]: 0

[8]:
```
compare_liste([0, 1, 2, 4], [0, 1, 2])
```

[8]: 1

1.3 Q3 - précision des calculs

On cherche à calculer la somme des termes d'une suite géométriques de raison~$\frac{1}{2}$. On définit $r = \frac{1}{2}$, on cherche donc à calculer $\sum_{i=0}^{\infty} r^i$ qui une somme convergente mais infinie. Le programme suivant permet d'en calculer une valeur approchée. Il retourne, outre le résultat, le nombre d'itérations qui ont permis d'estimer le résultat.

[9]:
```
def suite_geometrique_1(r):
    x = 1.0
    y = 0.0
    n = 0
    while x > 0:
        y += x
        x *= r
        n += 1
    return y, n

print(suite_geometrique_1(0.5))
```

```
(2.0, 1075)
```

Un informaticien plus expérimenté a écrit le programme suivant qui retourne le même résultat mais avec un nombre d'itérations beaucoup plus petit.

[10]:
```
def suite_geometrique_2(r):
    x = 1.0
    y = 0.0
    n = 0
    yold = y + 1
    while abs (yold - y) > 0:
        yold = y
        y += x
        x *= r
        n += 1
    return y,n

print(suite_geometrique_2(0.5))
```

```
(2.0, 55)
```

Expliquez pourquoi le second programme est plus rapide tout en retournant le même résultat. Repère numérique : $2^{-55} \sim 2,8.10^{-17}$.
Tout d'abord le second programme est plus rapide car il effectue moins d'itérations, 55 au lieu de 1075. Maintenant, il s'agit de savoir pourquoi le second programme retourne le même résultat que le premier mais plus rapidement. L'ordinateur ne peut pas calculer numériquement une somme infinie, il s'agit toujours d'une valeur approchée. L'approximation dépend de la précision des calculs, environ 14 chiffres pour *python*. Dans le premier programme, on s'arrête lorsque r^n devient nul, autrement dit, on s'arrête lorsque x est si petit que *python* ne peut plus le représenter autrement que par~0, c'est-à-dire qu'il n'est pas possible de représenter un nombre dans l'intervalle $[0, 2^{-1055}]$.
Toutefois, il n'est pas indispensable d'aller aussi loin car l'ordinateur n'est de toute façon pas capable d'ajouter un nombre aussi petit à un nombre plus grand que~1. Par exemple, $1 + 10^{17} = 1,000\,000\,000\,000\,000\,01$. Comme la précision des calculs n'est que de 15 chiffres, pour *python*, $1 + 10^{17} = 1$.
Le second programme s'inspire de cette remarque : le calcul s'arrête lorsque le résultat de la somme n'évolue plus car il additionne des nombres trop petits à un nombre trop grand. L'idée est donc de comparer la somme d'une itération à l'autre et de s'arrêter lorsqu'elle n'évolue plus.
Ce raisonnement n'est pas toujours applicable. Il est valide dans ce cas car la série $s_n = \sum_{i=0}^{n} r^i$ est croissante et positive. Il est valide pour une série convergente de la forme $s_n = \sum_{i=0}^{n} u_i$ et une suite u_n de module décroissant.

1.4 Q4 - hypercube

Un chercheur cherche à vérifier qu'une suite de~0 et de~1 est aléatoire. Pour cela, il souhaite compter le nombre de séquences de n nombres successifs. Par exemple, pour la suite 01100111 et $n = 3$, les triplets sont 011, 110, 100, 001, 011, 111. Le triplet 011 apparaît deux fois, les autres une fois. Si la suite est aléatoire, les occurrences de chaque triplet sont en nombres équivalents.

[11]:
```
def hyper_cube_liste(n, m=None):
    if m is None:
        m = [0, 0]
    if n > 1 :
        m[0] = [0,0]
        m[1] = [0,0]
        m[0] = hyper_cube_liste (n-1, m[0])
        m[1] = hyper_cube_liste (n-1, m[1])
    return m

hyper_cube_liste(3)
```

[11]:
```
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
```

La seconde à base de dictionnaire (plus facile à manipuler) :

[12]:
```
def hyper_cube_dico (n) :
    r = { }
    ind = [ 0 for i in range (0,n) ]
    while ind [0] <= 1 :
        cle = tuple(ind) # conversion d'une liste en tuple
        r[cle] = 0
        ind[-1] += 1
        k = len(ind)-1
        while ind[k] == 2 and k > 0:
            ind[k] = 0
            ind[k-1] += 1
            k -= 1
    return r
```

```
hyper_cube_dico(3)
```

[12]:
```
{(0, 0, 0): 0,
 (0, 0, 1): 0,
 (0, 1, 0): 0,
 (0, 1, 1): 0,
 (1, 0, 0): 0,
 (1, 0, 1): 0,
 (1, 1, 0): 0,
 (1, 1, 1): 0}
```

Le chercheur a commencé à écrire son programme :

[13]:
```
def occurrence(l,n) :
    # d = ....... # choix d'un hyper_cube (n)
    # .....
    # return d
    pass

suite = [ 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1 ]
h = occurrence(suite, 3)
h
```

Sur quelle structure se porte votre choix (a priori celle avec dictionnaire), compléter la fonction `occurrence`.

[14]:
```
def occurrence(tu, n):
    d = hyper_cube_dico(n)
    for i in range (0, len(tu)-n) :
        cle = tu[i:i+n]
        d[cle] += 1
    return d

occurrence((1, 0, 1, 1, 0, 1, 0), 3)
```

[14]:
```
{(0, 0, 0): 0,
 (0, 0, 1): 0,
 (0, 1, 0): 0,
 (0, 1, 1): 1,
 (1, 0, 0): 0,
 (1, 0, 1): 2,
 (1, 1, 0): 1,
 (1, 1, 1): 0}
```

Il est même possible de se passer de la fonction `hyper_cube_dico` :

[15]:
```
def occurrence2(tu, n):
    d = { }
    for i in range (0, len(tu)-n) :
        cle = tu[i:i+n]
        if cle not in d:
            d[cle] = 0
        d [cle] += 1
    return d

occurrence2((1, 0, 1, 1, 0, 1, 0), 3)
```

[15]:
```
{(1, 0, 1): 2, (0, 1, 1): 1, (1, 1, 0): 1}
```

La seule différence apparaît lorsqu'un n-uplet n'apparaît pas dans la liste. Avec la fonction `hyper_cube_dico`, ce n-uplet recevra la fréquence 0, sans cette fonction, ce n-uplet ne sera pas présent dans le dictionnaire `d`. Le même programme avec la structure matricielle est plus une curiosité qu'un cas utile.

[16]:
```
def occurrence3(li, n):
    d = hyper_cube_liste(n)
    for i in range (0, len(li)-n) :
        cle = li[i:i+n]
        t   = d                         #
        for k in range (0,n-1) :        # point clé de la fonction :
            t = t[cle[k]]               #                 accès à un élément
        t [cle [ n-1] ] += 1
    return d

occurrence3((1, 0, 1, 1, 0, 1, 0), 3)
```

[16]:
```
[[[0, 0], [0, 1]], [[0, 2], [1, 0]]]
```

Une autre écriture...

[17]:
```
def hyper_cube_liste2(n, m=[0, 0], m2=[0, 0]):
    if n > 1 :
        m[0] = list(m2)
        m[1] = list(m2)
        m[0] = hyper_cube_liste2(n-1, m[0])
        m[1] = hyper_cube_liste2(n-1, m[1])
    return m

def occurrence4(li, n):
    d = hyper_cube_liste2(n)          # * remarque voir plus bas
    for i in range (0, len(li)-n) :
        cle = li[i:i+n]
        t   = d                         #
        for k in range (0,n-1) :        # point clé de la fonction :
            t = t[cle[k]]               #                 accès à un élément
        t [cle [ n-1] ] += 1
    return d

occurrence4((1, 0, 1, 1, 0, 1, 0), 3)
```

[17]:
```
[[[0, 0], [0, 1]], [[0, 2], [1, 0]]]
```

Et si on remplace `list(m2)` par `m2`.

[18]:
```
def hyper_cube_liste3(n, m=[0, 0], m2=[0, 0]):
    if n > 1 :
        m[0] = m2
        m[1] = m2
        m[0] = hyper_cube_liste3(n-1, m[0])
        m[1] = hyper_cube_liste3(n-1, m[1])
    return m

def occurrence5(li, n):
    d = hyper_cube_liste3(n)          # * remarque voir plus bas
```

```
    for i in range (0, len(li)-n) :
        cle = li[i:i+n]
        t   = d                         #
        for k in range (0,n-1) :        # point clé de la fonction :
            t = t[cle[k]]               #                 accès à un élément
        t [cle [ n-1] ] += 1
    return d

try:
    occurrence5((1, 0, 1, 1, 0, 1, 0), 3)
except Exception as e:
    print(e)
```

```
'int' object is not iterable
```

Intéressant...

[19]: