Correction du TD ipt^2 n° 7: Dictionnaires 2: Programmation dynamique

Exercice n°1: Recherche du chemin optimal dans une pyramide de nombres

Listing 1:

❶
```python
def max_nb(x,y):
    if y < x :
        return x
    else :
        return y
```

Listing 2:

❷
```python
def gen_tab(n):
    T = np.zeros([n,n], dtype=int)
    for i in range(n) :
        for j in range(i+1) :
            T[i,j] = rd.randint(1,10)
    return T
```

❸ **a·** L'information sur la condition de terminaison (cas de base) est dans l'énoncé: les récursions s'arrêtent dès que la base de la pyramide est atteinte. Donc si i représente l'indice de parcours des lignes de la pyramide et n la hauteur totale de celle-ci, il y a arrêt lorsque $i = n$.

b· D'après la représentation matricielle de la pyramide, les sous-matrices g et d correspondant aux sous-pyramides g et d sont les suivantes:

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ \mathbf{1} & 8 & 0 & 0 & 0 \\ \mathbf{3} & \mathbf{1} & 2 & 0 & 0 \\ \mathbf{2} & \mathbf{10} & \mathbf{1} & 3 & 0 \\ \mathbf{3} & \mathbf{1} & \mathbf{2} & \mathbf{1} & 3 \end{bmatrix}$$

Sous-matrice "g"

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 1 & \mathbf{8} & 0 & 0 & 0 \\ 3 & \mathbf{1} & \mathbf{2} & 0 & 0 \\ 2 & \mathbf{10} & \mathbf{1} & \mathbf{3} & 0 \\ 3 & \mathbf{1} & \mathbf{2} & \mathbf{1} & \mathbf{3} \end{bmatrix}$$

Sous-matrice "d"

Dans l'algorithme proposé, on doit additionner la valeur du sommet t_{11} de la pyramide initiale avec la plus grande somme obtenue à partir des sous-pyramides g et d. Pour la matrice correspondante de sommet $t[i, j]$, on doit donc additionner $t[i, j]$ soit à la somme de la sous-matrice inférieure gauche $s_{i+1,j}$, soit à celle de la sous-matrice inférieure droite $s_{i+1,j+1}$ en fonction de la plus grande des deux valeurs; la relation de récurrence sur les sommes calculées est donc:

$$s_{ij} = t_{ij} + max\_nb(s_{i+1,j}\ ,\ s_{i+1,j+1})$$

c· Compte tenu de ce qui précède, on dégage très facilement la fonction récursive:

Listing 3:

```python
def s_rec(t,i,j) :
    N=t.shape[0]
    if i==N:
        return 0
    else :
        return t[i,j] + max_nb(s_rec(t,i+1,j),s_rec(t,i+1,j
    +1))
```

d· On calcule la somme totale de la pyramide (représenté par la matrice t) en lançant l'appel initial à la fonction `s_rec(t,i,j)` avec les paramètres suivants: `print s_rec(t,0,0)`

e· Chaque récursion comporte deux appels récursifs, et chaque appel récursif se fait sur une pyramide de hauteur réduite d'une unité; les récursions s'arrêtent lorsque les sous-pyramides sont de hauteur 1; la relation de récurrence sur la complexité en terme d'appels récursifs s'écrit donc:

$$C(n) = 2 \times C(n-1) = 2^2 \times C(n-2) = 2^{n-1} \times \underbrace{C(1)}_{=1}$$

$C(1)$ correspondant à la complexité du cas de base pour lequel la pyramide est de hauteur 1; le programme se contente alors de renvoyer la valeur 0 (puisqu'il n'y a plus rien à sommer), donc une opération en temps constant $C(1) = 1$.
Ainsi: $\boxed{C(n) = 2^{n-1}}$

❹ **a·** On propose la fonction suivante:

Listing 4:

```python
def s_iter(t):
    n=t.shape[0]
    sum_tab = np.copy(t)
    for i in range(n-2,-1,-1):
        for j in range(i+1) :
            sum_tab[i,j] = sum_tab[i,j] + max_nb(sum_tab[i+1,j], sum_tab[i+1,j+1])
    return sum_tab
```

b· La première boucle sur i compte à rebours de $N-2$ à 0 compris, i prendra donc $N-1$ valeurs. La seconde boucle compte de 0 à i compris. Le nombre d'itérations est donc:

$$C(n) = (n-1)+(n-2)+\ldots+1 = \frac{1}{2}(n-1)(1+n-1) \implies \boxed{C(n) = \frac{1}{2}n(n-1)}$$

Conclusion: la complexité de cette méthode est en $O(n^2)$, ainsi cet algorithme de type programmation dynamique est bien plus efficace que son homologue récursif!!!

c· Pour reconstruire le chemin qui aboutit à la somme maximale dans la pyramide, on reprend le **tableau des sommes cumulées** renvoyé par la fonction `s_iter(t)`. On part du sommet (somme maximale), et l'on détermine **le plus grand des deux éléments adjacents sur la ligne d'en dessous** puisque le chemin de collecte maximale passe nécessairement par lui: si c'est celui dans la même colonne, alors on enregistre le déplacement élémentaire alors effectué dans une chaine de caractère avec "B" pour "Bas", et si c'est celui dans la colonne immédiatement à droite alors on enregistre dans la chaine "D" pour Diagonal. Une proposition de code est la suivante:

Listing 5:

```python
def recons_chemin(t):
    sumtab=s_iter(t)
    n,m=sumtab.shape
    i,j=0,0 # on part du sommet de la pyramide
    chemin=""
    while i<n-1:
        if sumtab[i+1,j]>sumtab[i+1,j+1]: #on est descendu juste en dessous
            chemin=chemin+"B"
            i+=1
        else: #on est descendu juste en dessous et en diagonale
            chemin=chemin+"D"
            i+=1
            j+=1
    return sumtab,chemin #renvoie à la fois la matrice des sommes cumulées et le chemin de collecte maximale
```

Exercice n°2: Partition équilibrée d'un tableau d'entiers positifs

1. On a $S = S(L_1)+S(L_2)$ soit: $S(L_2) = S - S(L_1)$ donc:

$$|S(L_1)-S(L_2)| = |2S(L_1)-S| = |S-2S(L_1)|$$

Ainsi, rechercher le minimum de $|S(L_1)-S(L_2)|$ revient à rechercher le minimum de $|S-2S(L_1)|$, et donc le minimum de:

$$\left|\frac{S}{2}-S(L_1)\right|$$

2. Pour la recherche par force brute, on formera la sous-liste L_1 en testant pour chaque élément de la liste L **s'il doit ou non figurer dans** L_1 pour assurer $|S(L_1)-L(L_2)|$ minimale; ainsi, il y a deux possibilités pour chaque élément de la liste qui comporte n éléments ce qui entraine un total de 2^n possibilités à tester (complexité exponentielle).

3. **Approche récursive**

a· On propose la fonction récursive suivante s'appuyant sur la stratégie décrite:

Listing 6:

```python
def partition_rec(L): #on évite l'apparition du paramètre S12 dans l'appel initial
    def part(L,S12):
        if len(L)==1: # la liste n'a qu'un seul élément (cas de base)
            return L
        p1,p2 = part(L[1:],S12-L[0]), part(L[1:],S12) # on forme récursivement la liste L1 suivant les deux possibilités
        s1=L[0]+sum(p1) #on forme la somme s1 des éléments de L1 en intégrant l'élément L[0]
```

```python
            s2=sum(p2) #on forme la somme s2 des éléments de L1 sans intégrer L[0]
            if abs(s1-S12)<abs(s2-S12): #on teste laquelle des deux sommes est la plus proche de la 1/2 somme des éléments de L
                return [L[0]]+p1 # si c'est la première, alors on concatène L[0] avec p1
            return p2 # sinon on renvoie p2
    return part(L,sum(L)/2)
```

b· Chaque appel récursif en entraine deux avec pour chacun une liste privée d'un élément, on a donc la récurrence suivante:

$$C(n) = 2C(n-1) = 2^{n-1}C(1)$$

or $C(1) = 1$
soit:$C(n) = 2^{n-1}$
donc une complexité "en gros" $\boxed{C(n) = O(2^n)}$

c· On peut améliorer ce code en évitant d'éventuelles réévaluations de sommes en stockant ces dernières dans un dictionnaire (mémoïsation); on propose le code suivant dans lequel les valeurs de sommes sont stockées dans un dictionnaire avec comme clé les tuples $(str(L), S12)$, la conversion de la liste en chaîne étant nécessaire pour obtenir un objet "hachable":

Listing 7:

```python
def partition_rec_mem(L):
    def part(L,S12,D):
        if (str(L),S12) not in D:
            if len(L)==1:
                return L
            p1,p2 = part(L[1:],S12-L[0],D), part(L[1:],S12,D)
            s1=L[0]+sum(p1)
            s2=sum(p2)
            if abs(s1-S12)<abs(s2-S12):
                D[(str(L),S12)]=[L[0]]+p1
            else:
                D[(str(L),S12)]=p2
        return D[(str(L),S12)]
    return part(L,sum(L)/2,{})
```

4. Fonction python `tableau_somme(L)`:

Listing 8:

```python
import numpy as np
def tableau_somme(L):
    n=len(L)
    S=sum(L)
    T=np.full((n+1,S+1),False, dtype=bool)
    T[:,0]=True
    for i in range(len(L)):
        v=L[i]
        for j in range(1,S+1):
            if T[i,j]:
                T[i+1,j]=True
            elif T[i,j-v] and j>=v:
                T[i+1,j]=True
    return T
```

a· On a d'après le tableau précédent $j_m = \lfloor \dfrac{14}{2} \rfloor = 7$.

b· On peut obtenir la somme $j_m - L[i]$.

c· Stratégie de constitution de la sous-liste L_1:

- On construit le tableau T de booléens pour la somme L;
- on calcule la partie entière de la demi-somme des éléments de L soit j_m;
- En partant de la cellule de coordonnées $[n, j_m]$, on recherche, en se déplaçant sur la même ligne n et vers la gauche, la première valeur à `True`.
- on remonte ensuite dans cette colonne jusqu'à trouver la première valeur `False` à l'indice i; on retient alors l'élément $T[i]$ pour la liste L_1.
- On se déplace à l'abscisse $j_m - L[i]$ et on réitère ce processus tant que $j > 0$.

Pour le tableau correspondant à la liste $[2, 4, 5, 3]$ proposée en énoncé, le trajet de "remontée" est le suivant:

$j\rightarrow$ / $i\downarrow$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	T	F	$\mathbf{F\ (L[0]=2)}$	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	$\uparrow T$	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	$\uparrow T$	F	T	F	T	$\mathbf{F\ (L[2]=5)}$	F	F	F	F	F	F	F
3	T	F	T	F	T	T	T	$\uparrow T$	F	T	F	T	F	F	F
4	T	F	T	T	T	T	T	$\uparrow T$	T	T	T	T	T	F	T

Ainsi, la sous-liste recherchée est: $L_1 = [5, 2]$

On propose la fonction suivante:

Listing 9:

```python
def partition_dyn(L):
    L1=[]
    n=len(L)
    T=tableau_somme(L)
    jm=sum(L)//2
    while not T[n,jm]:
        jm-=1
    while jm>0:
        h=len(L)
        while T[h,jm]:
            h-=1
        v=L[h]
        L1.append(v)
        jm-=v
    return L1
```