

Interrogation d'informatique n°3. Corrigé

Problème. Partitions équilibrées d'entiers

```
1) def somme_liste(E) :  
  
    s = 0  
  
    for x in E : s = s + x  
  
    return s
```

2) Méthode gloutonne

```
def glouton(E) :  
  
    s = 0 ; t = 0 ; A = []      # t vaut la somme des éléments de A  
  
    for x in E :  
  
        s = s + x  
  
        if abs(s-2*(t+x)) < abs(s-2*t) :  
  
            A.append(x) ; t = t + x  
  
    return A
```

3) Force brute

a) Si on n'effectuait pas une copie profonde de L , les premiers éléments de L seraient modifiés par **append**.

Ainsi, **genere(E)** renverrait une liste contenant 2^n fois la liste E .

En fait, il y aurait aussi un problème si on utilisait une boucle **for A in L**, car L étant modifiée au sein de la boucle, la boucle **for** (qui est codée par un **while**) ne terminerait jamais.

Remarque : Une autre programmation possible est :

```
01. def genere(E) :  
  
02.     n = len(E) ; L = [[]]  
  
03.     for x in E :  
  
04.         M = L.copy()  
  
05.         for A in M :  
  
06.             L.append(A+[x])  
  
07.     return L  
  
b) def equilibre(E) :  
  
    s = somme_liste(E)  
  
    delta = s ; A = []      # s est la valeur associée à la partie vide  
  
    for B in genere(E) :      # B décrit toutes les parties possibles  
  
        i = somme_liste(B)
```

```

    d = abs(2*i-s)

    if d < delta :    delta = d ; A = B

return A

```

c) La complexité de `genere(E)` et de `equilibre(E)` est en $O(n2^n)$.

En effet, la complexité de `genere(E)` est en $O(\sum_{k=1}^n k2^k)$, mais $\sum_{k=1}^n k2^k \leq \sum_{k=1}^n n2^k \leq n2^{n+1}$.

4) Programmation dynamique de bas en haut

a) `def sommes(E) :`

```

    n = len(E) ; s = somme_totale(E)

    P = [ [False]*(s+1) for k in range(n+1) ] ; P[0][0] = True

    for k in range(1,n+1) :

        for i in range(s+1) :    P[k,i] = P[k-1,i] or (E[k-1]<=i and P[k-1,i-E[k-1]])

    return P

```

b)

```

def equilibre(E) :

    n = len(E) ; P = sommes(E) ; s = len(P[n])-1

    j = 0

    for i in range(1,s+1) :

        if P[n][i] and abs(2*i-s)<abs(2*j-s) :    j = i

    return abs(2*j-s)

```

Remarque : On pourrait aussi utiliser une boucle `while` à condition de partir de la position médiane m et de parcourir les valeurs possibles selon la distance à cette position médiane (avec un test pour les valeurs de i de la forme $m - j$ et $m + j$ jusqu'à obtenir une valeur de P valant `True`).

c) La construction de P demande $O(ns)$ opérations.

d)

```

def extraction(P,E,i) :

    n = len(P) ; s = len(P[n])

    if not P[n][i] :    return None

    A = [] ; j = i

    for k in range(n,0,-1) :    # k décrit les entiers de n à 1

        x = E[k-1]

        if not P[k-1][j] :    A.append(x) ; j = j-x

    # variante : if x<=j and P[k-1][j-x] :    A.append(x) ; j = j-x

```

```
return A
```

La valeur `True` ou `False` de $P(k-1, j-x)$ permet de savoir s'il convient d'utiliser ou non $E[k-1]$ dans la liste optimale A .

5) Programmation dynamique de haut en bas

```
def sommes(E) :  
    d = {}  
    def traite(k,i) :  
        if not (k,i) in d :  
            if (k,i) == 0: d[(0,0)] = []  
            elif k == 0 : d[(0,i)] = None  
            else :  
                x = E[k-1]  
                traite(k-1,i)  
                if (x<=i) : traite(k-1,i-x)  
                if (k-1,i) in dico and d[(k-1,i)] != None :  
                    d[(k,i)] = d[(k-1,i)]  
                elif (x<=i) and ((k-1,i-x) in dico) and d[(k-1,i-x)] != None :  
                    d[(k,i)] = d[(k-1,i-x)]+[x]  
                else : d[(k,i)] = None  
    return d
```

Deux exercices indépendants

1)

```
01. def somme(L) :  
02.     if isinstance(x,int) : return x    ### test d'arrêt  
03.     s = 0  
04.     for M in L : s = s + somme(M)  
05.     return s
```

2) a) La commande `assert` permet de s'assurer que p est un entier naturel non nul.

Les lignes 04 à 09 consistent à effectuer un tri par insertions sur le sous-tableau des termes d'indice congru à r modulo p , c'est-à-dire le sous-tableau $[x_r, x_{r+p}, x_{r+2p}, x_{r+3p}, \dots]$.

Ce tri est effectué pour toute valeur de $r \in \{0, 1, \dots, p-1\}$.

b) En particulier, $\mathbf{tri}(\mathbf{A},1)$ est le tri par insertions classique (dans ce cas, r prend 0 comme unique valeur).

Le nombre d'échanges effectués est $N = \sum_{i=0}^{n-1} m(i) \leq \sum_{i=0}^{n-1} i = \frac{1}{2}n(n-1)$, donc de l'ordre de $\frac{1}{2}n^2$.

Cette valeur est atteinte lorsque les éléments de A sont classés par ordre décroissant.

c) Le nombre d'échanges effectués par $\mathbf{tri}(\mathbf{A},2)$ est donc au plus

$$\frac{1}{2}n_0(n_0-1) + \frac{1}{2}n_1(n_1-1), \text{ où } n_0 = \left\lfloor \frac{n}{2} \right\rfloor \text{ et } n_1 = \left\lceil \frac{n}{2} \right\rceil, \text{ donc de l'ordre de } \frac{1}{4}n^2$$

Une fois les deux sous-tableaux $[x_0, x_2, x_4, \dots]$ et $[x_1, x_3, x_5, \dots]$ triés, on obtient un tableau A vérifiant :

$$m(i) \leq \left\lceil \frac{i}{2} \right\rceil.$$

En effet, après $\mathbf{tri}(\mathbf{A},2)$, tous les x_j , où $j < i$ est de même parité que i , vérifient $x_j \leq x_i$.

Donc le nombre d'échanges effectués ensuite par $\mathbf{tri}(\mathbf{A},1)$ est donc au plus

$$\sum_{i=0}^{n-1} \left\lceil \frac{i}{2} \right\rceil, \text{ qui est de l'ordre de } \frac{1}{2} \left(\frac{n}{2} \right) n = \frac{1}{4}n^2$$

Ainsi, au total, $\mathbf{tri}(\mathbf{A},1)$; $\mathbf{tri}(\mathbf{A},2)$ effectue environ $\frac{n^2}{2}$ échanges, comme dans $\frac{n^2}{2}$.

La méthode n'est pas plus efficace dans le pire cas.

Mais on peut montrer qu'elle améliore la complexité moyenne (sur les $n!$ configurations possibles) car les éléments qui doivent être avancés dans la liste progressent plus rapidement vers leur position si on utilise d'abord des pas de 2 (au lieu de pas de 1).

En effectuant $\mathbf{tri}(\mathbf{A},p)$ successivement pour tous les entiers p de la forme $2^a 3^b$ pris dans l'ordre décroissant, on obtient une complexité moyenne en $O(n(\log n)^2)$, qui est la meilleure complexité connue dans un tri Shell.