

## Corrigé du TP n° 4

Avant toute chose, on importe les fonctions nécessaires à ce TP et on définit trois listes qui nous serviront de référence pour la suite :

```
from time import time
from random import randint
lst1 = [randint(-10, 10) for k in range(1000)]
lst2 = [randint(-10, 10) for k in range(10000)]
lst3 = [randint(-10, 10) for k in range(100000)]
```

### Exercice 1. L'algorithme naïf

Il consiste à calculer les sommes de toutes les coupes de  $a$  et à en prendre le minimum :

```
def somme(a, i, j):
    s = 0
    for x in a[i:j]:
        s += x
    return(s)

def coupe_min1(a):
    m, n = a[-1], len(a)
    for i in range(n - 1):
        for j in range(i + 1, n + 1):
            m = min(m, somme(a, i, j))
    return(m)
```

Notons qu'une fonction `sum` existe en PYTHON et permet de remplacer `somme(a, i, j)` par `sum(a[i:j])`.

Cette fonction réalise essentiellement des additions et des appels à la fonction `min`, il est donc judicieux de compter ceux-ci pour mesurer la complexité de cet algorithme.

L'appel `somme(a, i, j)` utilise  $j - i + 1$  additions, donc le nombre total d'additions utilisées par la fonction `coupe_min1` est :

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^n (j - i + 1) = \frac{1}{6}(n-1)(n+3)(n+4) \sim \frac{n^3}{6} = \Theta(n^3)$$

et le nombre d'appels à la fonction `min` :

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^n 1 = \frac{1}{2}(n-1)(n+2) \sim \frac{n^2}{2} = \Theta(n^2).$$

La complexité totale de cet algorithme est donc un  $\Theta(n^3)$ .

Expérimentons cette fonction avec la première des trois listes :

```
debut = time()
print("la coupe minimale vaut", coupe_min1(lst1))
fin = time()
print("durée :", fin - debut)
```

On obtient une durée de l'ordre de 15s (sur mon ordinateur). Si nous utilisions cette fonction pour la liste `lst2`, il faudrait prévoir un temps multiplié par  $10^3$  c'est à dire plus de quatre heures, et pour `lst3`, plus de 173 jours.

### Exercice 2. Un algorithme de coût quadratique

Pour obtenir un algorithme de complexité quadratique, il faut diminuer le nombre d'additions. On définit une fonction calculant la valeur minimale d'une coupe de la forme  $a[i : j]$  pour  $i$  fixé, avec  $j \in \llbracket i + 1, n \rrbracket$  :

```
def mincoupe(a, i):
    m = s = a[i]
    for j in range(i+1, len(a)):
        s += a[j]
        m = min(m, s)
    return(m)
```

et on en déduit la fonction :

```
def coupe_min2(a):
    m = a[-1]
    for i in range(len(a) - 1):
        m = min(m, mincoupe(a, i))
    return(m)
```

La fonction mincoupe utilise  $n - i - 1$  additions et appels à la fonction min, donc le nombre total d'additions utilisées par coupe\_min2 vaut :

$$\sum_{i=0}^{n-2} (n - i - 1) = \frac{n(n-1)}{2} \sim \frac{n^2}{2} = \Theta(n^2)$$

et le nombre d'appels à la fonction min :

$$\sum_{i=0}^{n-2} (n - i) = \frac{(n-1)(n+2)}{2} \sim \frac{n^2}{2} = \Theta(n^2).$$

Le coût est devenu quadratique.

L'expérimentation donne cette fois 0,17s pour calculer la coupe minimale de lst1 et 18s pour lst2. Si nous utilisons cette fonction pour la liste lst3, il faudrait prévoir un temps multiplié par  $10^2$ , soit environ une demi-heure.

### Exercice 3. Un algorithme de coût linéaire

Une coupe se terminant par  $a_i$  est soit égale à  $a[i]$  soit à  $a[j : i] + a[i]$ , où  $a[i : i]$  est une coupe se terminant par  $a_{i-1}$ . On en déduit que  $c_{i+1} = \min(a_i, c_i + a_i)$ .

Par ailleurs, une coupe de  $a[0 : i + 1]$  est soit une coupe de  $a[0 : i]$ , soit une coupe se terminant par  $a[i + 1]$ , donc  $m_{i+1} = \min(m_i, c_{i+1})$ .

Ces deux relations nous permettent d'itérer ces deux suites en temps linéaire pour calculer  $m_n$ , égal à la coupe minimale de  $a$  :

```
def coupe_min3(a):
    c = m = a[0]
    for i in range(1, len(a)):
        c = min(c + a[i], a[i])
        m = min(m, c)
    return(m)
```

L'expérimentation donne cette fois 0,0007s pour lst1, 0,006s pour lst2 et 0,06s pour lst3.

### Exercice 4. Un algorithme diviser pour régner

Posons  $k = \lfloor n/2 \rfloor$ . Toute coupe de  $a$  est :

- ou bien une coupe de  $a[0 : k]$  ;
- ou bien une coupe de  $a[k : n]$  ;
- ou bien une coupe  $a[i : j]$  avec  $i < k < j$ .

Dans ce dernier cas,  $a[i : j]$  est la concaténation de  $a[i : k]$  et de  $a[k : j]$ , donc la somme est minimale dès lors que les sommes  $s[i : k]$  et  $s[k : j]$  le sont.

On en déduit la démarche « diviser pour régner » suivante :

```
def coupe_rec(a):
    n = len(a)
    k = n // 2
    if n == 1:
        return a[0]
    else:
        m = min(coupe_rec(a[:k]), coupe_rec(a[k:]))
        m1 = s = a[k-1]
        for i in range(k-2, -1, -1):
            s += a[i]
            m1 = min(m1, s)
        m2 = s = a[k]
        for i in range(k+1, n):
            s += a[i]
            m2 = min(m2, s)
        return min(m, m1+m2)
```

On peut observer que la recherche de la valeur minimale dans le troisième cas se fait en temps linéaire, ce qui justifie la relation :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

On peut dès lors montrer que  $C(n) = \Theta(n \log n)$ .

L'expérimentation donne 0.006s pour `lst1`, 0.076s pour `lst2` et 0.882S Pour `lst3`.

On comprend sur ces valeurs pourquoi une complexité en  $\Theta(n \log n)$  est qualifiée de *semi-linéaire* : les durées obtenues sont bien plus proches de celles de l'algorithme linéaire que de l'algorithme quadratique.

### Exercice 5. Gain maximal

Pour déterminer le gain maximal, on s'inspire de la fonction `coupe_min3` en cherchant à itérer la suite  $g$  définie par :

$$g_k = \max\{a_j - a_i \mid 0 \leq i < j \leq k\}.$$

Clairement,  $g_{k+1} = \max(g_k, c_{k+1})$  avec  $c_k = \max\{a_k - a_i \mid 0 \leq i < k\}$ , et  $c_{k+1} = a_{k+1} - a_k + \max(c_k, 0)$ .

On prend comme valeurs initiales  $g_1 = c_1 = a_1 - a_0$ , ce qui conduit à la fonction (clairement de complexité linéaire) :

```
Def gain_max(a):
    g = c = a[1]-a[0]
    for k in range(2, len(a)):
        c = a[k] - a[k-1] + max(c, 0)
        g = max(g, c)
    return g
```