Lycée Louis-Le-Grand, Paris 2018/2019
MPSI 4 – Informatique commune
N. Carré, A. Troesch

TP n° 4 : Problèmes de complexité

Nous étudions dans ce TP plusieurs algorithmes dont le but est de répondre à la même question, en vue de comparer leur vitesse d'exécution. Pour mesurer le temps d'exécution d'un programme, nous utiliserons la fonction `time`, qu'on importera du module `time`. La fonction `time()` renvoie une durée exprimée en secondes depuis une date de référence. Ainsi, en définissant en tout début de programme une variable `debut = time()` et en toute fin de programme une variable `fin = time()`, la quantité `fin - début` mesure le temps qui s'est écoulé entre le début et la fin de l'exécution du programme.

Correction du problème –

Partie I – L'algorithme naïf

Il consiste à énuérer tous les couples (i, j) tels que $i < j$ et à calculer la somme sur la coupe correspondante.

```
import random
import time

lst1 = [random.randint(-10,10) for i in range(1000)]
lst2 = [random.randint(-10,10) for i in range(10000)]
lst3 = [random.randint(-10,10) for i in range(100000)]

####### ALGORITHME NAÏF #######

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

def coupe_min1(a):
    m = a[0]
    for i in range(len(a)):
        for j in range(i+1,len(a)+1):
            s = somme(a,i,j)
            if s < m:
                m = s
    return(m)
```

L'algorithme est en $\Theta(n^3)$. En effet, en notant $C_1(n)$ ne nombre d'opérations à effectuer pour un tableau de taille n, on a, pour chaque valeur de i, début de la coupe, et chaque valeur de $j > i$, un nombre d'opérations à faire égal à $j - i$

(calcul de la somme, en partant de la somme vide), plus 1 ou 2 opérations (un test, et éventuellement une affectation). On obtient alors

$$C_1(n) \geqslant \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} (j-i+1) = \sum_{i=0}^{n-1} \sum_{j=1}^{n-i-1} j+1 = \sum_{i=0}^{n-1} \left(\frac{(n-i-1)(n-i)}{2} + 1 \right).$$

Pour simplifier cette somme, on fait un changement d'indice $i' = n-i-1$:

$$C_1(n) \geqslant \left(\sum_{i=0}^{n-1} \frac{i(i+1)}{2} \right) + n = \frac{(n-1)n(2n-1)}{12} - \frac{n(n-1)}{12} + n \underset{+\infty}{\sim} \frac{n^3}{6}.$$

Le même calcul donne un majorant similaire, le seul facteur qui change étant le terme n du bout, remplacé par $2n$. Ainsi, on obtient $\boxed{C_1(n) \sim_{+\infty} \frac{n^3}{6}}$

Remarquez la logique intuitive du coefficient obtenu : en moyenne la longueur de l'intervalle $[i, j]$ est à peu près $\frac{n}{3}$ (les 2 valeurs i et j vont en moyenne séparer l'intervalle total en trois parts de même longueur). Il faut donc en moyenne à peu près $\frac{n}{3}$ opérations pour calculer la somme sur une coupe, et les coupes sont déterminés par les couples (i, j) tels que $i < j$, donc par les points entiers d'un demi-carré sans sa diagonale. Lorsque n devient grand, il y a donc à peu près $\frac{n^2}{2}$ coupes. On obtient bien un ordre de grandeur égal à $\frac{n^3}{6}$ opérations.

Le temps d'exécution (mesure donnée après la partie IV) pour $n = 1000$ est déjà long (une vingtaine de seconde). Suivant les implémentations, on peut faire mieux, mais cela reste de l'ordre de plusieurs secondes. L'algorithme étant en n^3, multiplier les données par 10 multiplie le temps de calcul par $10^3 = 1000$. Ainsi, sur un tableau de taille 10000, on peut s'attendre à un temps de réponse de 20000 secondes, soit plus de 5 heures. Pour un tableau de taille 100000, on peut s'attendre à un temps de réponse de 231 jours.

Partie II – Un algorithme quadratique

On peut remarquer qu'on a des redondances dans les calculs. Lorsqu'on compare les sommes de toutes les coupes commençant par l'indice i, on recalcule les sommes successives (obtenues en faisant varier j) depui le début, alors qu'on pourrait se contenter de rajouter un terme à la somme précédente. Cette amélioration toute simple va nous permettre une amélioration nette de la complexité. Pour cela, on commence par calculer la somme minimale d'une coupe commençant en i, en n'effectuant qu'un calcul de somme.

```
######## ALGORITHME QUADRATIQUE ########

def mincoupe(a,i):
    m = a[i]
    s = a[i]
    for k in range(i+1,len(a)):
        s+=a[k]
        if s<m:
            m = s
    return m

def coupe_min2(a):
    mi = a[0]
    for i in range(len(a)):
        m = mincoupe(a,i)
        if m < mi:
            mi=m
    return(mi)
```

Cette fois, pour chaque valeur de i, on effectue une seule somme, correspondant à la somme de tous les termes d'indice i à $n-1$, soit $n-i$ termes. On pourrait procéder par encadrement pour gérer le test, ou alors considérer tout le bloc (une opération de somme, un test, une éventuelle affectation) comme une seule opération ayant un coût variable selon deux types de situations, donc correspondant à deux opérations plus complexes, de coût différent, mais chacune ayant un coût constant. Notre cours nous dit que la complexité peut se calculer en comptant ces opérations. Ainsi, on obtient :

$$C_2(n) = \sum_{i=0}^{n-1}(n-i) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2};$$

On obtient bien $C_2(n) = \Theta(n^2)$.
Cette amélioration de la complexité est considérable : en multipliant la taille d'une donnée par 10, on multiplie le temps de calcul par 100 et non 1000. Par exemple, le temps de calcul pour $n = 1000$ étant de l'ordre de 7 centièmes de secondes, on peut espérer un temps de calcul de l'ordre de 7 secondes pour $n = 10000$, ce qu'on observe effectivement. Pour $n = 100000$, il faudrait compter 700 secondes, soit entre 11 et 12 minutes. Cela reste accessible, mais peu commode.

Partie III – III Un algorithme linéaire

On calcul le minimum de façon dynamique, en augmentant petit à petit la taille du tableau considéré : on considère, pour tout i, la valeur minimale m_i d'une coupe du tableau partiel $a[0:i]$, ainsi que c_i la somme minimale d'une coupe (non vide) se terminant par l'indice $i-1$. Il se trouve que (m_i) et (c_i) vérifient des relations très simples. En effet, si m_i et c_i sont connus :

- Les coupes non vides terminant en a_i sont obtenues par concaténation d'une liste non vide terminant en a_{i-1} et de la liste contenant l'unique terme a_i, sauf la liste $[a_{i+1}]$. Parmi les listes obtenues de la première façon, celles qui fournissent une somme minimale sont celles donc la première parite fournissait déjà une somme minimale au rang précédent, somme valant c_i par définition. Ainsi, la somme minimale des listes du premier type est $c_i + a_i$. Il suffit alors de comparer cette somme à celle de la liste supplémentaire $[a_i]$.
  On obtient donc $c_{i+1} = \min(c_i, a_i)$.
- une coupe minimale de $a[0:i+1]$ était soit déjà une coupe minimale de $a[0:i]$, soit une coupe se terminant en $i+1$, dont la somme est minimale dans l'ensemble de toutes les coupes, donc aussi dans l'ensemble des coupes finissant en a_i. Ainsi, en confrontant ces deux possibilités, on obtient $m_{i+1} = \min(m_i, c_{i+1})$.

L'initialisation se fait avec une liste de longueur 1 : $m_0 = a[0] = c_0$. On obtient alors l'algorithme très simple suivant :

```
######## ALGORITHME LINÉAIRE ########

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

Dans chaque passage dans la boucle, on a 3 opérations à effectuer. Ainsi la complexité est maintenant $C_3(n) = 3n = \Theta(n)$. On a donc un algorithme linéaire.
Multiplier la taille des données par 10 a alors le même effet sur le temps d'exécution. On observe bien ce phénomène sur les essais.
Le temps obtenu pour un tableau de taille 100000 laisse envisager un temps de réponse de 5 secondes pour un tableau de taille 1000000. C'est raisonnable !

Partie IV – Algorithme récursif

On coupe maintenant le tableau en deux parts à peu près égales. La coupe minimale sera alors entièrement dans l'un des deux tableaux (on calcule donc le minimum récursivement pour chaque tableau), ou alors à cheval sur les deux. Dans ce dernier cas, si on coupe entre a_{k-1} et a_k, une coupe à cheval est obtenue par concaténation d'une coupe terminant en a_{k-1} et d'une coupe commençant en a_k. Obtenir sous cette forme une coupe de somme minimale se fait en minimisant la somme de la coupe terminant en a_{k-1} (parmi l'ensemble des coupes terminant en a_{k-1}), et en minimisant de même la somme de la coupe commençant en a_k.
On est donc ramené à la recherche d'une coupe minimale commençant en a_k (sur le même principe que dans la partie II), et d'une coupe minimale terminant en a_{k-1} (ce qu'on fait de même, dans l'autre sens).

```
######## ALGORITHME RÉCURSIF #######

def coupe_min_intermediaire(a,k):
    m1 = a[k-1]
    s = a[k-1]
    for i in range(1,k):
        s += a[k-1-i]
        if s < m1:
            m1 = s
    m2 = a[k]
    s = a[k]
    for i in range(k+1,len(a)):
        s += a[i]
        if s < m2:
            m2 = s
    return m1 + m2

def coupe_min4(a):
    if len(a)==1:
        return a[0]
    else:
        k = len(a) // 2
        m1 = coupe_min4(a[:k])
        m2 = coupe_min4(a[k:])
        m3 = coupe_min_intermediaire(a,k)
        return min([m1,m2,m3])
```

Soit $C_4(n)$ le nombre d'opérations effectuées pour un tableau de taille n. On fait un appel récursif sur deux tableaux de taille à peut près $\frac{n}{2}$ (exactement si n est pair, et $\frac{n-1}{2}$ et $\frac{n+1}{2}$ si n est impair), et une recherche d'un minimum à cheval qui se fait, comme en II, en $\Theta(n)$. Ainsi, on obtient la relation de récurrence :

$$C(n) = \begin{cases} 2C\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n \text{ pair} \\ C\left(\frac{n-1}{2}\right) + C\left(\frac{n+1}{2}\right) + \Theta(n) & \text{sinon.} \end{cases}$$

Nous avons fait les essais avec les listes définies plus haut, en nous limitant à ce qui est raisonnable :

```
######## ESSAIS ########

i=1
print("\nPour␣n␣=␣1000:")
```

```
for coupe in [coupe_min1, coupe_min2, coupe_min3, coupe_min4]:
    debut = time.time()
    m=coupe(lst1)
    fin = time.time()
    print("Résultat avec l'algorithme {} : {}".format(i,m))
    print("Temps d'execution : {}".format(fin-debut))
    i+=1

i=2
print("\nPour n = 10000:")
for coupe in [coupe_min2, coupe_min3, coupe_min4]:
    debut = time.time()
    m=coupe(lst2)
    fin = time.time()
    print("Résultat avec l'algorithme {} : {}".format(i,m))
    print("Temps d'execution : {}".format(fin-debut))
    i+=1

i=3
print("\nPour n = 100000:")
for coupe in [coupe_min3, coupe_min4]:
    debut = time.time()
    m=coupe(lst3)
    fin = time.time()
    print("Résultat avec l'algorithme {} : {}".format(i,m))
    print("Temps d'execution : {}".format(fin-debut))
    i+=1
```

Les valeurs obtenues sont :

```
Pour n = 1000:
Résultat avec l'algorithme 1 : -271
Temps d'execution : 18.197659969329834
Résultat avec l'algorithme 2 : -271
Temps d'execution : 0.07236099243164062
Résultat avec l'algorithme 3 : -271
Temps d'execution : 0.0006670951843261719
Résultat avec l'algorithme 4 : -271
Temps d'execution : 0.004403114318847656

Pour n = 10000:
Résultat avec l'algorithme 2 : -359
Temps d'execution : 6.954966068267822
Résultat avec l'algorithme 3 : -359
Temps d'execution : 0.005347013473510742
Résultat avec l'algorithme 4 : -359
Temps d'execution : 0.04796099662780762
```

```
Pour n = 100000:
Résultat avec l'algorithme 3 : -6618
Temps d'execution : 0.05915021896362305
Résultat avec l'algorithme 4 : -6618
Temps d'execution : 0.5492720603942871
```

Partie V – Gain maximal

Il est inutile de refaire quelque chose de neuf : on peut se servir de ce qui précède. On peut ramener le gain à un calcul de somme (somme telescopique des gains consécutifs). Ainsi, on peut former le tableau des gains consécutifs, et rechercher la somme maximale dans ce tableau. On peut même se ramener à la recherche de la somme minimale en inversant tous les signes.

```python
def gainmax(a):
    b = [a[i]-a[i+1] for i in range(len(a)-1)]
    return -coupe_min3(b)
```