Lycée Louis-Le-Grand, Paris 2015/2016
MPSI 4 – Informatique commune
G. Dewaele, A. Troesch

TP n° 6 : 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.

Problème – Recherche du sous-tableau minimal

Dans ce problème, on consière des listes d'entiers relatifs $a = (a_0, \dots, a_{n-1})$, et on appelle *coupe* de a toute suite non vide $(a_i, \dots, a_{j-1})$ d'entiers consécutifs de cette liste $(0 \leqslant i < j \leqslant n)$, qu'on notera désormais $a[i : j]$.

À toute coupe $a[i : j]$, on associe la somme $s[i : j] = \sum_{k=i}^{j-1} a_k$ des éléments qui la composent. Le but de ce problème est de déterminer un algorithme efficace pour déterminer la valeur minimale des sommes des coupes de a.

On expérimentera les différentes fonctions avec trois listes aléatoires de longueur 1000, 10000 et 100000, qu'on nommera respectivement `lst1`, `lst2` et `lst3`, contenant des entiers pris au hasard entre -10 et 10. On utilisera pour créer ces listes la fonction `randint(a, b)`, du module `random`, qui retourne un entier arbitraire de l'intervalle $[\![a, b]\!]$.

Partie I – L'algorithme naïf

1. Définir une fonction `somme` prenant en paramètre une liste a et deux entiers i et j, et retournant la somme $s[i : j]$.
2. En déduire une fonction `coupe_min1` prenant en paramètre une liste a et retournant la somme minimale d'une coupe de a.
3. Montrer que la complexité de cette algorithme est en $\Theta(n^3)$.
4. Mesurer le temps d'exécution pour la liste `lst1`. Quel temps pourrait-on prévoir d'attendre pour les listes `lst2` et `lst3` ?

Partie II – Un algorithme de coût quadratique

1. Définir, sans utiliser la fonction `somme`, une fonction `mincoupe` prenant en paramètres une liste a et un entier i, et calculant la valeur minimale de la somme d'une coupe de a dont le premier élément est a_i, en parcourant une seule fois la liste `a` à partir de l'indice i.
2. En déduire une fonction `coupe_min2` permettant de déterminer la somme minimale des coupes de a, en temps quadratique. On justifiera que la complexité est précisément en $\Theta(n^2)$.
3. Mesurer le temps d'exécution de la fonction `coupe_min2` pour les listes `lst1` et `lst2`. Quel temps pourrait-on prévoir pour `lst3` ?

Partie III – Un algorithme de coût linéaire

Étant donnée une liste a, on note m_i la somme minimale d'une coupe quelconque de la liste $a[0:i]$, et c_i la somme minimale d'une coupe de $a[0:i]$ se terminant par a_{i-1}.

1. Montrer que $c_{i+1} = \min(c_i + a_i, a_i)$ et $m_{i+1} = \min(m_i, c_{i+1})$, et en déduire une fonction `coupe_min3` de coût linéaire, calculant la somme minimale d'une coupe de a.
2. Mesurer le temps d'exécution de la fonction `coupe_min3` pour les listes `lst1`, `lst2` et `lst3`.

Partie IV – Un algorithme diviser-pour-régner

Nous proposons un dernier algorithme, basé sur l'utilisation de la récursivité, consistant à définir une fonction à l'aide d'elle-même, par un appel de ladite fonction avec des paramètres de taille plus petite (ou au moins qui nous rapproche des conditions initiales). La structure d'une fonction récursive est toujours la même : on commence par l'initialisation, si les paramètres passés nous placent dans cette situation, et sinon, nous faisons un ou plusieurs appels récursifs nous permettant de nous rapprocher des conditions initiales.

Un algorithme de type diviser-pour-régner est un algorithme proposant de scinder le problème initial en plusieurs sous-problèmes de taille plus petite (par exemple deux sous-problèmes de taille deux fois plus petite que le problème initial)

1. Soit $k = \left\lfloor \frac{n}{2} \right\rfloor$. Justifier que la coupe de somme minimale de a est :
   - soit entièrement contenue dans $a[0:k]$,
   - soit entièrement contenue dans $a[k:n]$,
   - soit constituée de la concaténation de la coupe (non vide) $a[i_0:k]$ réalisant le minimum de la somme sur les coupes terminant par a_{k-1}, et de la coupe (non vide) $a[k:j_0]$, réalisant le minimum de la somme sur les coupes commençant par a_k.
2. En déduire une fonction `coupe_min4`, de type diviser-pour-régner, calculant la somme minimale d'une coupe de a, telle que sa complexité $C(n)$ vérifie :

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

3. Mesurer le temps d'exécution de cette fonction sur les trois listes définies initialement.

On pourrait montrer que la relation de récurrence trouvée pour C impose que $C(n) = \Theta(n \lg n)$ (on dit dans ce cas que la complexité est quasi-linéaire).

Partie V – Gain maximal

Étant donné un tableau a, on recherche maintenant le gain maximal, à savoir la quantité $\max_{i<j}(a_j - a_i)$. Il s'agit par exemple du gain maximal que peut faire une personne en achetant et en revendant une seule fois une action en bourse sur une période donnée, les valeurs du tableau correspondant au cours de l'action, sur la période considérée (on ne peut évidemment pas revendre avant d'avoir acheté...)

Écrire une fonction recherchant le gain maximal de a en temps linéaire.