Algorithmique : exercices et éléments de cours

L2 informatique

Vendredi 13 février

1 Preuves de correction

Exercice 1.
Montrer rigoureusement la correction de l'algorithme de tri par insertion vu à la fin du dernier cours : on prouvera un invariant pour chaque boucle. Donner le pire et le meilleur cas pour la complexité et évaluer cette complexité dans les deux cas.

Lorsque l'on a une boucle `Pour i de 1 à n`, on sait qu'elle sera exécutée exactement n fois : au bout de n itérations, la boucle se termine. En revanche, ce n'est nullement le cas des boucles `Tant que` qui continuent tant que la condition est vérifiée. Il se peut qu'on entre dans une boucle infinie (condition toujours vraie), et dans les cas qui nous intéressent il n'est pas toujours aisé de montrer que ce genre de boucle termine. Par exemple, on ne sait pas montrer que la boucle suivante termine pour tout a (problème de Syracuse) :

```
x <- a
Tant que (x > 1) faire
  Si x est pair, x <- x/2
  Sinon x <- 3*x+1
```

La méthode standard pour montrer qu'une boucle `Tant que` termine est de montrer qu'il existe une quantité (typiquement un entier positif) qui décroît à chaque itération : dans ce cas, la boucle s'arrête dès que l'entier devient nul car il ne peut plus décroître. Par exemple, dans la boucle suivante :

```
S : tableau d'entiers
x <- a
y <- b
Tant que (x > 0 et y > 0) faire
  Si S[x] est pair alors x <- x-1
  Sinon y <- y-1
```

c'est la quantité $x+y$ qui décroît à chaque étape. Puisque la condition `x>0 et y>0` impose que la somme soit positive, on en déduit que la boucle termine (en au plus $a+b$ itérations).
Exercice 2.
On se donne un tableau S d'entiers triés par ordre croissant et un entier cible x. Proposer un algorithme linéaire pour trouver s'il existe deux éléments de S dont la somme vaut x. Prouver que la boucle termine (quelle quantité décroît à chaque itération ?) et que votre algorithme est correct (en montrant un invariant).

2 Tri fusion

On rappelle le principe du tri fusion d'un tableau A : on coupe le tableau A au milieu, de façon à avoir les deux moitiés A_1 et A_2. On trie récursivement A_1 et A_2, puis on fusionne les deux moitiés de manière à ce que le tableau A entier soit trié.

On appellera `Fusion(A1,A2,A)` la procédure qui fusionne les deux tableaux A_1 et A_2 dans A. On a donc un algorithme de la forme :

```
Tri_fusion(A) :
  Si A a un seul élément, sortir
  Recopier A[1..n/2] dans A1
  Recopier A[n/2+1..n] dans A2
  Tri_fusion(A1)
  Tri_fusion(A2)
  Fusion(A1,A2,A)
```

Exercice 3.
Proposer un algorithme linéaire pour la procédure de fusion. Prouver que cet algorithme est correct, c'est-à-dire que la fusion de deux tableaux triés donne un tableau trié dont les éléments sont tous ceux des deux tableaux de départ.

Afin d'évaluer la complexité d'un algorithme récursif, on appelle $T(n)$ le nombre d'étapes nécessaires pour trier un tableau de taille n. Puisque l'on fait deux appels à `Tri_fusion` sur des tableaux de taille $n/2$ et que la fusion a un coût linéaire, on obtient la relation de récurrence suivante :

$$\begin{cases} T(1) \in \Theta(1) \\ T(n) = 2T(n/2) + \Theta(n) \end{cases}$$

Exercice 4.
Pour simplifier, on suppose que la taille du tableau est une puissance de 2 (donc les divisions par 2 « tombent juste »), que $T(1) = 1$ et que la fusion a un coût exactement n (c'est-à-dire $T(n) = 2T(n/2) + n$). On pose $u_n = T(2^n)$. Montrer que $u_0 = 1$ et $u_n = 2u_{n-1} + 2^n$. En posant $v_n = u_n/2^n$, montrer que $v_n = v_{n-1} + 1$. En déduire la valeur de v_n, puis de u_n et de $T(2^n)$. Si $N = 2^n$, exprimer $T(N)$ en fonction de N.

3 Divers

Exercice 5. *Tours de Hanoï*
Dans le jeu des tours de Hanoï, on dispose de trois tiges A, B et C sur lesquelles peuvent venir se poser des disques de diamètres tous distincts. Au début de la partie, on dispose de n disques sur la tige A, celui de plus grand diamètre en dessous et rangés par diamètres décroissants (donc celui de plus petit diamètre au dessus). Le but du jeu est de déplacer toute la pile de disques sur la tige B. Pour cela, à chaque étape on a le droit de déplacer un disque d'une tige à une autre tant qu'on le pose sur un disque de plus grand diamètre ou sur une tige vide.
Écrire un algorithme récursif pour résoudre le problème. Évaluer sa complexité.

4 Indications pour les exercices

Exercice 2 : utiliser 2 curseurs, l'un partant du début du tableau et l'autre de la fin. On comparera la somme des deux éléments pointés par les curseurs avec la valeur cible x et on mettra à jour les curseurs en fonction.
On pourra utiliser un invariant (à formaliser) de la forme "x ne peut pas être atteint comme somme de deux éléments dont l'un d'entre eux est avant le premier curseur ou après le second curseur".

Exercice 3 : on utilise trois curseurs, un pour A_1, un pour A_2 et un pour A, qui partent du début des tableaux. On écrit dans le tableau A à la position du curseur soit l'élément de A_1 pointé par le curseur, soit celui de A_2 (selon leur valeur) puis on met à jour les curseurs.
On pourra utiliser un invariant (à formaliser) de la forme "le début du tableau A jusqu'au curseur est trié et contient tous les éléments précédant les curseurs de A_1 et A_2".

Exercice 4 : on a $v_n = n + 1$, $u_n = (n + 1)2^n$, $T(2^n) = (n + 1)2^n$ et donc $T(N) = (1 + \log N)N \in \Theta(N \log N)$.

Exercice 5 : pour déplacer une pile de taille n, on déplace d'abord les $n - 1$ premiers disques sur la tige C, puis le plus grand disque sur la tige B, puis on redéplace la pile de la tige C sur la tige B. Pour la complexité de l'algorithme, on a donc la relation de récurrence $T(n) = 2T(n - 1) + O(1)$ ce qui donne $T(n) \in \Theta(2^n)$ (méthode similaire à l'exercice 4).