Université de Mons–Hainaut
FS/1/5684 – Algorithmique
Examen de seconde session – Partie pratique

Le 25 août 2008

Consignes

– Pour cette partie, vous avez le droit de consulter vos notes et tout ouvrage qui vous semble utile.
– Cette partie de l'examen dure 1 heure 45 minutes.
– Veillez à bien justifier vos réponses. Une réponse mal justifiée, même correcte, ne permet pas d'obtenir le maximum des points.
– Quand vous indiquez une complexité, veillez à bien expliquer ce que sont les paramètres qui apparaissent dans le $\mathcal{O}$. Par exemple, $\mathcal{O}(n^2)$ n'a aucun sens si n n'apparaît pas dans l'algorithme ou dans la définition de la structure qui est traitée…

Question 1 – 3 points

On considère la fonction suivante, qui a pour but de calculer la somme des éléments stockés dans un vecteur d'entiers V, de taille n :

```
début
    Entier somme := 0 ;
    Entier i := 1 ;
    tant que i ≤ n faire
        somme := somme + V[i] ;
        i := i + 1 ;
fin
```

Donc, si cette fonction est correcte, l'égalité suivante doit être vraie à la fin de la boucle :

$$somme = \sum_{j=1}^{n} V[j] \quad (1)$$

1. Parmi les assertions suivantes, lesquelles constituent un invariant de la boucle ci-dessus ? Justifiez votre réponse pour les assertions qui ne sont *pas* un invariant.
   (a) $i \leq n+1$ et $somme = \sum_{j=1}^{i-1} V[j]$
   (b) $i \leq n$ et $somme = \sum_{j=1}^{i-1} V[j]$
   (c) $somme = \sum_{j=1}^{i-1} V[j]$
   (d) $somme = \sum_{j=1}^{i} V[j]$
2. Parmi les assertions qui constituent un invariant lesquelles permettent de prouver que (1) est bien correcte à la fin de la boucle ? Donnez la preuve que (1) est correcte pour chacun des invariants qui le permettent.

Rappel important : nous avons utilisé la convention du cours de numéroter les cases à partir de 1.

Correction

Seules les assertions (a) et (c) sont des invariants de la boucle. L'assertion (b) n'est pas un invariant, car, lors du dernier tour de boucle, la variable i prend la valeur $n+1$, ce que n'autorise pas l'assertion (b). Elle est donc fausse juste avant de sortir de la boucle. L'assertion (d) n'est pas un invariant car elle affirme que $somme$ contient à tout moment la somme des éléments se trouvant dans les cases de 1 à i. Or, ceci est faux dès l'entrée dans la boucle. En effet, en entrant dans la boucle, on a $i = 1$ et $somme = 0$ (on n'a encore rien ajouté dans $somme$), alors que d'après l'assertion (d) on devrait avoir $somme = \sum_{j=1}^{1} V[j] = V[1]$. Ce n'est clairement pas le cas.
Des deux invariants, seul l'assertion (a) (la plus forte) permet de prouver la correction de la boucle. On procède de façon classique, en commençant par calculer la négation de la condition de la boucle :

$$\neg B \quad \equiv \quad i \geq n+1$$

En prenant la conjonction avec l'invariant proposé (l'assertion (a)), on obtient une assertion qui est vraie à la fin de la boucle :

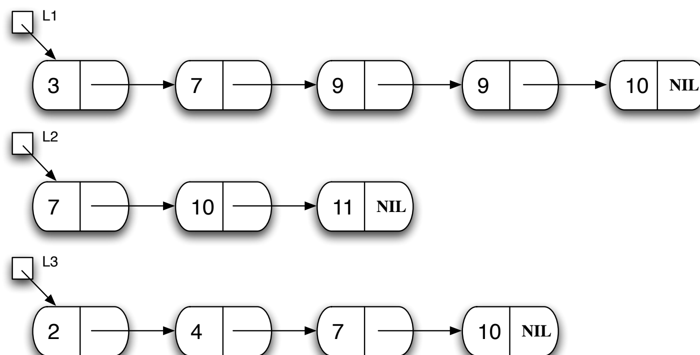
$$\begin{aligned}
\neg B \wedge Inv \quad &\equiv \quad i \geq n+1 \wedge i \leq n+1 \wedge somme = \sum_{j=1}^{i-1} V[j] \\
&\equiv \quad i = n+1 \wedge somme = \sum_{j=1}^{i-1} V[j] \\
&\equiv \quad i = n+1 \wedge somme = \sum_{j=1}^{n} V[j]
\end{aligned}$$

ce qui prouve que la boucle calcule bien la somme des n premières cases du tableau. Remarquons que l'assertion (c), tout en étant un invariant de la boucle, est trop faible, car elle ne permet pas de déterminer la valeur précise de i à la fin de la boucle.

Question 2 – 4 points

On vous demande d'écrire une algorithme *itératif* qui reçoit trois *listes* contenant des entiers positifs *triés par ordre croissant*, et qui renvoie le plus petit entier qui est présent dans les trois listes. L'algorithme renverra -1 s'il n'existe pas de valeur commune.

Par exemple, sur les listes suivantes :



La valeur retournée est 7 (remarquez que 10 est également présent dans les trois listes, mais ce n'est pas la plus petite valeur commune).

Correction

Supposons que les pointeurs donnant accès aux têtes des listes s'appellent L_1, L_2, et L_3. Comme il faudra parcourir ces trois listes, on aura besoin de trois pointeurs p_1, p_2, p_3 qui sont initialisés respectivement à L_1, L_2 et L_3. Une technique simple consisterait à considérer chaque élément de L_1, et à le rechercher dans L_2 puis dans L_3. On peut utiliser la fonction de recherche dans une liste triée vue au cours.

```
Entier RecPlsPetValCom(Elem * L1, Elem * L2, Elem * L3)
début
  | tant que L1 ≠ NIL faire
  |   | si Recherche(L2,L1.info) ≠ NIL et Recherche(L3,L1.info) ≠ NIL alors
  |   |   | retourner L1.info ;
  |   | sinon L1 := L1.next ;
  | retourner −1 ;
fin
```

Malheureusement, cela signifie qu'il faut parcourir potentiellement tout L_2 et tout L_3 pour chaque valeur dans L_1, soit une complexité en $\mathcal{O}(\ell_1 \times (\ell_2 + \ell_3))$, où ℓ_i est la longueur de L_i $(i = 1,2,3)$. Ce n'est pas la meilleure solution. On peut obtenir un algorithme plus efficace en utilisant le fait que les listes sont triées. Ayant les trois pointeurs p_1, p_2 et p_3 qui pointent respectivement vers un élément de L_1, L_2, L_3, on a deux possibilités :

1. soit $p_1.\text{info} = p_2.\text{info} = p_3.\text{info}$. On a donc trouvé une solution. Si on a pris garde de parcourir les listes depuis le début, on est sûr qu'il s'agit bien de la plus petite valeur commune, puisque les listes sont triées par ordre croissant.
2. soit $p_1.\text{info} \neq p_2.\text{info}$ ou $p_2.\text{info} \neq p_3.\text{info}$. Dans ce cas, il faut faire avancer l'un des trois pointeurs. Faire avance un pointeur revient à *éliminer* une valeur : si on fait avancer p_1 par exemple, on suppose que $p_1.\text{info}$ n'est pas la solution, puisqu'on ne revient jamais en arrière.

   Il est alors facile de voir que la valeur à éliminer est la plus petite des trois, car cette valeur ne peut plus apparaître dans les deux autres listes, puisque celles-ci sont triées par ordre croissant. Si par exemple les trois valeurs sont 15, 10 et 13, on est certain que 10 n'apparaîtra plus ni dans la première ni dans la troisième liste. Par contre, 15 pourrait encore apparaître après 13 et 10. On peut donc se contenter de faire avancer le pointeur qui pointe vers l'information 10.

On obtient donc :

```
Entier RecPlsPetValCom(Elem * L1, Elem * L2, Elem * L3)
début
  | Elem * p1 := L1 ;
  | Elem * p2 := L2 ;
  | Elem * p3 := L3 ;
  | tant que p1 ≠ NIL ∧ p2 ≠ NIL ∧ p3 ≠ NIL faire
  |   | si p1.info = p2.info = p3.info alors
  |   |   | retourner p1.info ;
  |   | sinon
  |   |   | si p1.info ≤ p2.info ∧ p1.info ≤ p3.info alors
  |   |   |   | p1 := p1.next ;
  |   |   | sinon si p2.info ≤ p1.info ∧ p2.info ≤ p3.info alors
  |   |   |   | p2 := p2.next ;
  |   |   | sinon
  |   |   |   | p3 := p3.next ;
  | retourner −1 ;
fin
```

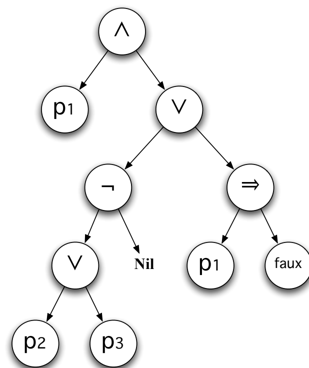
La complexité au pire cas de cette fonction est obtenue quand il est nécessaire de parcourir entièrement les trois listes, soit $\mathcal{O}(\ell_1 + \ell_2 + \ell_3)$.

Question 3 – 3 points

Pour cette question, nous allons considérer des *expression booléennes*. Une expression booléenne est constituée à l'aide de variables booléennes (elles ne peuvent valoir que *vrai* ou *faux*) et des opérateurs logiques habituels ($\vee$ pour « ou », $\wedge$ pour « et », $\Rightarrow$ pour l'implication et $\neg$ pour la négation). On peut également utiliser des parenthèses et les constantes *vrai* et *faux*. Par exemple, $p_1 \wedge (\neg(p_2 \vee p_3) \vee (p_1 \Rightarrow \mathit{faux}))$ est une expression booléenne.

En donnant à chaque variable une valeur (*vrai* ou *faux*), on obtient une valeur pour l'expression. Par exemple, si p_1 vaut *vrai*, p_2 vaut *faux* et p_3 vaut *faux*, l'expression ci-dessus vaut *vrai*. Si par contre on remplace la valeur de p_1 par *faux*, l'expression vaut *faux*.

On peut représenter une telle expression sous forme d'un arbre *binaire*. Les feuilles sont étiquettées par *vrai*, *faux* ou par une variable. Les nœuds internes sont étiquettés par $\vee$, $\wedge$, $\Rightarrow$ ou $\neg$. Dans les trois premiers cas, les deux fils sont les arbres qui représentent les deux opérandes. Dans le cas de $\neg$, le fils gauche est l'arbre représentant l'expression dont on veut prendre la négation, et le fils droit est l'arbre vide.



On vous demande d'écrire un algorithme *récursif* qui reçoit un arbre représentant une expression booléenne, ainsi qu'un tableau T de booléens tel que $T[i]$ est la valeur assignée à p_i, et qui renvoie la valeur de l'expression pour ces valeurs des variables. Par exemple, sur l'arbre ci-dessus et en prenant $T =$ | *vrai* | *faux* | *faux* |, l'algorithme doit renvoyer *vrai*.

Correction

La solution est quasi-identique à l'algorithme d'évaluation d'une expression arithmétique (voir l'algorithme 39 du syllabus, section 6.2.5).

```
Booléen EvalExBool(Nœud * A,Entier T[n])
début
    si EstVide(A) ou InfoRacine(A) = faux alors
        retourner faux ;
    sinon si InfoRacine(A) = vrai alors
        retourner vrai ;
    sinon si InfoRacine(A) = p_i alors
        retourner T[i];
    sinon
        Booléen vg := EvalExBool(A.fg, T) ;
        Booléen vd := EvalExBool(A.fd, T) ;
        si InfoRacine(A) = ¬ alors
            retourner ¬vg;
        sinon si InfoRacine(A) = ∨ alors
            retourner vg ∨ vd;
        sinon si InfoRacine(A) = ∧ alors
            retourner vg ∧ vd;
        sinon si InfoRacine(A) = ⇒ alors
            retourner vg ⇒ vd;
fin
```