Algorithmique et structures de données : examen de première session

Ensimag 1A

Année scolaire 2009–2010

Consignes générales :
- Durée : 3 heures.
- Calculatrices, portables et tous instruments électroniques interdits. Livres interdits. Autres documents autorisés.
- Le barème est donné à titre indicatif.
- Les trois exercices sont indépendants et peuvent être traités dans le désordre.
- La syntaxe Ada ne sera pas un critère déterminant de l'évaluation des copies. En d'autres termes, les correcteurs n'enlèveront pas de point pour une syntaxe Ada inexacte mais compréhensible (pensez à bien commenter votre code !).
- **Merci d'indiquer votre numéro de groupe de TD et de rendre votre copie dans le tas correspondant à votre groupe.**

Exercice 1 : Diviser pour régner *(7 pts)*

Question 1 ***(1,5 pts)*** On suppose disposer d'un type `Matrice` qui représente des matrices carrées d'ordre 2, et d'une fonction de multiplication de ces matrices appelée `Mult`.

```
type Matrice is array(1..2,1..2) of Natural ;

function Mult(M1,M2: Matrice) return Matrice ;
```

Écrire une fonction `Power` qui calcule M^N en appliquant un principe "*diviser pour régner*". Donner une approximation du coût asymptotique de votre algorithme en fonction de N. On attend ici une solution *asymptotiquement meilleure* que l'algorithme naïf itérant $N-1$ multiplications successives de M.

```
function Power(M: Matrice; N: Positive) return Matrice ;
```

Question 2 ***(3 pts)*** On définit la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ par récurrence sur n de la manière suivante :
- $F_0 = 0$ et $F_1 = 1$.
- pour $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$.

Pour $n \geq 1$, on définit un vecteur V_n en dimension 2 par :

$$V_n = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

1. Établir une relation de récurrence multiplicative entre V_{n+1} et V_n.
2. En déduire un algorithme pour calculer F_n lorsque $n \geq 2$, basé sur la fonction `Power` de la question 1.
3. Quel est le nombre d'opérations élémentaires sur les entiers (addition et multiplications confondues) effectuées par cet algorithme ? Que pensez-vous de ce résultat ?

Question 3 ***(2,5 pts)*** En utilisant un principe "*diviser pour régner*", décrire un algorithme qui calcule simultanément le minimum et le maximum d'un ensemble non ordonné à n éléments avec $n \geq 1$, et dont le nombre de comparaisons soit majoré par

$$\left\lceil \frac{3 \times 2^{\lceil \log_2(n) \rceil}}{2} \right\rceil - 2$$

Attention, on demande ici un **majorant exact et pas une approximation asymptotique**. Justifier soigneusement le calcul de coût. Comparer ce résultat au coût de l'algorithme itératif naïf.

Exercice 2 : Complexité en moyenne *(2 pts)*

Question 4 ***(2 pts)*** On considère le problème du tri par ordre croissant d'un tableau `T` d'éléments 2 à 2 **distincts** de taille N. Il existe donc une unique permutation qui permet de trier le tableau. On considère que chaque permutation de $[1..N]$ a la même probabilité d'être la permutation du tri. Calculer exactement le nombre **moyen** d'appels à la procédure `Swap` dans l'algorithme `BubbleSort` ci-dessous.

```
type Tab is array(1..N) of Element ;

procedure Swap(X,Y: in out Element) is
   Aux: Element := X ;
begin
   X:=Y ;
   Y:=Aux ;
end ;

procedure BubbleSort(T: in out Tab) is
   Swapped: Boolean ;
   I: Natural := N ;
begin
   loop
      Swapped:=False ;
      for J in 1..I-1 loop
        if T(J) > T(J+1) then
           Swap(T(J),T(J+1)) ;
           Swapped := True ;
        end if ;
      end loop ;
      exit when not Swapped ;
      I:=I-1 ;
   end loop ;
end ;
```

Exercice 3 : Arbres binaires *(11 pts)*

Dans l'ensemble de cet exercice, si A est un arbre binaire, on note $\mathcal{N}(A)$ son nombre de nœuds et $\mathcal{H}(A)$ sa hauteur. Par convention, si A est l'arbre vide, alors $\mathcal{N}(A) = 0$ et $\mathcal{H}(A) = -1$.

1 Étude de formes d'arbres binaires

Définition 1 (arbre optimal) *Un arbre binaire A est dit* optimal *ssi :*

$\mathcal{H}(A) \geq 0$ *implique* $\mathcal{N}(A) \geq 2^{\mathcal{H}(A)}$

Question 5 ***(1 pts)*** Montrer que si A est un arbre non vide qui satisfait la définition 1 alors

$$\mathcal{H}(A) = \lfloor \log_2 \mathcal{N}(A) \rfloor$$

Expliquer la définition ci-dessus : en quel sens un arbre qui satisfait la définition 1 est optimal ?

Définition 2 (arbre quasi-complet) *Un arbre binaire A est dit* quasi-complet *ssi il satisfait la définition inductive suivante :*

- *A est l'arbre vide*
- *ou, A est non vide et ses deux fils F_1 et F_2 sont quasi-complets tels que :*

$$|\mathcal{N}(F_1) - \mathcal{N}(F_2)| \leq 1$$

Question 6 ***(1,5 pts)*** Est-ce que A quasi-complet implique A optimal ? Inversement, est-ce que A optimal implique A quasi-complet ? Justifiez vos réponses.

2 Étude du coût de l'union d'ABR

On suppose donné un type `Element` muni d'une relation d'ordre total. On définit ici un type `OrdSet` pour représenter des ensembles finis dont les éléments sont des valeurs de `Element`. Dans cet exercice, une valeur de type `OrdSet` est un couple formé d'un arbre binaire de recherche de type `ABR` (pas forcément équilibré) contenant les éléments de l'ensemble, et du nombre d'éléments dans l'ensemble. Ce type `ABR` est défini classiquement comme un type pointeur sur des cellules de type `Noeud`. L'arbre vide (*i.e.* l'ensemble vide) est représenté par le pointeur `null`.

```
type Noeud ;
type ABR is access Noeud ;

type OrdSet is record
  A: ABR ;
  Card: Natural ;
end record ;

type Noeud is record
```

Question 9 *(2 pts)* Proposer un algorithme pour programmer la fonction `Union` qui construit l'union ensembliste de `S1` et `S2` avec un nombre de comparaisons *dans le pire cas* linéaire en fonction `S1.Card+S2.Card`.

```
function Union(S1,S2: OrdSet) return OrdSet ;
```

On ne demande pas de programmer précisément cette fonction, mais de bien décrire son principe et le calcul de son coût.

Question 10 *(0,5 pts)* On veut maintenant étendre la structure de donnée `ABR` de manière à ce que les arbres construits soient des ABR équilibrés AVL. Décrire (sans les programmer) les modifications qu'il faut apporter sur la structure de données `Noeud` et aux procédures et fonctions précédentes. Est-ce que les coûts de `Import` et `Union` sont modifiés ?

Question 11 *(1,5 pts)* On considère la procédure `UnionEnPlace` ci-dessous : elle place dans `S1` l'union de `S1` et `S2`, et détruit tous les éléments de `S2`.

```
procedure UnionEnPlace(S1, S2: in out OrdSet) is
   Aux: OrdSet ;
   X: Element ;
   DejaPresent: Boolean ;
begin
   if S1.Card < S2.Card then
      -- echange de S1 et S2.
      Aux:=S1;
      S1:=S2 ;
      S2:=Aux ;
   end if ;
   -- S1.Card >= S2.Card ;
   while S2.A /= null loop
      RemoveMin(S2.A,X) ;
      Insert(S1.A,X,DejaPresent) ;
      if not DejaPresent then
         S1.Card := S1.Card+1 ;
      end if ;
   end loop ;
   S2.Card := 0 ;
end ;
```

où les procédures suivantes sont supposées fournies et implémentées de manière efficace sur des AVLs :

```
procedure RemoveMin(A: in out ABR; Min: out Element) ;
-- requiert: A AVL qcq non nul.
-- garantit: Min minimum de l'ABR initial, supprimé de A.

procedure Insert(A: in out ABR; X: Element; DP: out Boolean) ;
-- requiert: A AVL qcq.
-- garantit:
```

```
  Val: Element ;
  FilsG, FilsD: ABR ;
end record ;
```

Dans le type `Noeud` ci-dessus :
– le champ `Val` contient l'élément à la racine de l'arbre
– le champ `FilsG` représente l'ensemble des éléments strictement inférieurs à `Val`
– le champ `FilsD` représente l'ensemble des éléments strictement supérieurs à `Val`
Ainsi, la fonction `NbElems` qui retourne le nombre d'éléments que contient un ensemble de type `ABR` est définie par :

```
function NbElems(A: ABR) return Natural is
begin
   if A=null then
      return 0 ;
   else
      return NbElems(A.FilsG)+NbElems(A.FilsD)+1 ;
   end if ;
end ;
```

Finalement, une valeur de type `S: OrdSet` vérifie l'invariant suivant : `S.Card=NbElems(S.A)` (`S.A` peut être nul).

Hypothèse 1 (coût des comparaisons) *Pour simplifier les analyses de coût, on suppose que le coût d'une* comparaison sur le type `Element` *est du même ordre que le coût d'une* comparaison sur les `Integer`*, lui-même du même ordre que celui d'une* comparaison de pointeurs.

Hypothèse 2 *Dans toute la suite, on fait l'hypothèse que le passage d'une tranche de tableau en paramètre d'entrée (en mode* `in`*,* `out` *ou* `in out`*) d'une procédure ou d'une fonction Ada se fait à coût constant (indépendant de la taille du tableau ou de la tranche).*[1]

Question 7 ***(2 pts)*** Écrire en Ada une fonction `Export` qui prend un `OrdSet` `S` et retourne l'ensemble des éléments de `S` sous la forme d'un tableau strictement croissant du type `Tab`.

```
type Tab is array(Positive range <>) of Element ;
function Export(S:OrdSet) return Tab ;
```

On demande d'écrire une implémentation qui fait un nombre de comparaisons linéaire en fonction de `Card(S)`. Justifier que votre implémentation vérifie cette propriété.

Question 8 ***(2,5 pts)*** Écrire en Ada une fonction `Import` qui réalise l'inverse de `Export` : sous la précondition que le tableau `T` est trié par ordre strictement croissant, elle retourne un `OrdSet` formé de l'ensemble des éléments de `T`. Il faut implémenter cette fonction de manière à ce que l'arbre retourné soit *quasi-complet* (voir définition 2).

```
function Import(T:Tab) return OrdSet ;
-- requiert T strictement croissant
```

Justifier que l'arbre retourné est quasi-complet. Calculer le nombre de comparaisons (d'entiers) effectuées en fonction de `T'Length`.

[1] Cette hypothèse semble vérifiée par le compilateur GNAT.

```
--     DP=True => X absent initialement dans A, et ajouté dans l'AVL
--     DP=False => X présent initialement dans A, et AVL non modifié
```

En supposant que la fonction `Union` travaille sur des AVL comme à la question 10, comparer cette fonction `Union` et la procédure `UnionEnPlace` ci-dessus pour le nombre de comparaisons dans le pire cas : préciser dans quelles conditions, le coût de quel algorithme est meilleur que l'autre (asymptotiquement).