Algorithmique et structures de données : examen de première session

Ensimag 1A

Année scolaire 2010–2011

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 deux 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 : Pavage par des triominos *(7 pts)*

Définition 1 (triomino) *Un triomino est un objet de forme L comportant* 3 *carrés* 1×1, *voir Figure 1 (a).*

Définition 2 (échiquier à défaut) *Un échiquier* $n \times n$ *avec un carré manquant quelque part sera appelé échiquier à défaut. Pour un exemple avec* $n = 4$, *voir Figure 1 (b) et (c).*

On considère le problème de déterminer si un échiquier à défaut peut être recouvert par des triominos, sachant que les triominos peuvent être tournés dans tous les sens.

Définition 3 (pavage) *Un recouvrement, une partition des cases de l'échiquier à défaut en triominos s'appelle pavage.*

Algorithme "Diviser pour régner"

Question 1 (0,5 pts) Donner un pavage des échiquiers 4×4 à défaut de la Figure 1 (b) et (c) par des triominos.

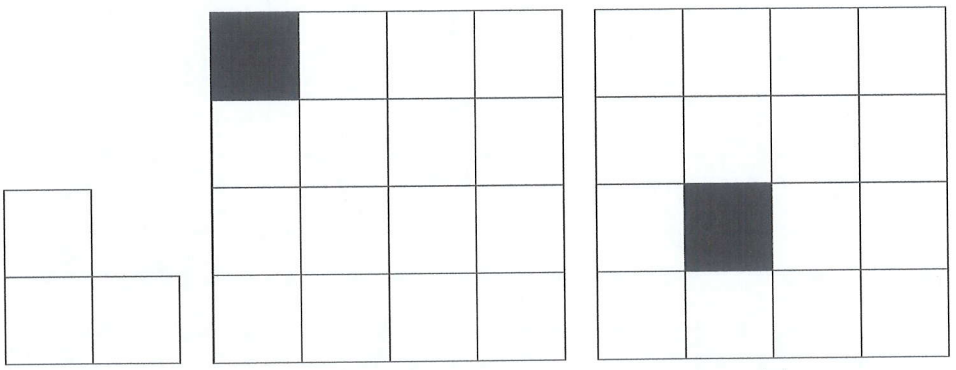


FIGURE 1 – (a) Triomino, (b) un échiquier 4×4 à défaut, (c) un autre échiquier 4×4 à défaut.

Question 2 (1 pts) Donner un échiquier 5×5 à défaut pour lequel il n'existe pas de pavage par triominos. Bien justifier pourquoi un tel pavage n'existe pas.

Question 3 (2,5 pts) On suppose que $n = 2^k, k \in \mathbb{N}$.

1. Décrire un algorithme de type *"diviser pour régner"* qui prend en entrée l'entier n et la position du carré manquant, et produit en sortie un pavage par des triominos de l'échiquier n^2 à défaut correspondant. On pourra couper l'échiquier à défaut en sous-échiquiers, tous de même taille, et placer un triomino de manière à créer un carré manquant sur chacun des sous-échiquiers. On se contentera ici d'une description de l'algorithme en français accompagnée de dessins. Par contre, il faut bien expliquer le cas de base et le traitement récursif.
2. Analyser le coût de l'algorithme proposé.

Représentation du pavage

Définition 4 (quadtree) *Un quadtree est une structure d'arbre général, où tout nœud possède exactement 4 fils, et permettant de partitionner un espace bidimensionnel. La racine du quadtree représente l'espace entier, et chaque fils d'un nœud N représente un quart de l'espace représenté par N.*

```
type NoeudQT ;
type Quadtree is access NoeudQT ;
type NoeudQT is record
   ...
   FilsNO, FilsNE, FilsSO, FilsSE: Quadtree ;
end record ;
```

Question 4 (1,5 pts) Expliquer comment on peut utiliser un quadtree pour représenter un pavage d'un échiquier à défaut par des triominos, selon l'algorithme "Diviser pour Régner" proposé à la question précédente. Expliquer notamment :

1. quelle est la relation entre les triominos et les nœuds du quadtree ;
2. ce que représente une feuille du quadtree ;
3. quel(s) champ(s) doi(ven)t être ajouté(s) à la structure `NoeudQT`.

Question 5 (0,5 pts) Expliquer comment construire le quadtree dans l'algorithme "Diviser pour Régner" de pavage proposé à la question 3.

Question 6 (1 pts) On cherche à savoir par quel triomino est recouverte une case donnée de l'échiquier à défaut. Décrire l'algorithme de parcours du quadtree renvoyant cette information. Quelle est sa complexité en pire cas ?

Exercice 2 : Tas appariants (pairing heaps) *(16 pts)*

Cet exercice étudie l'implémentation d'une file de priorités par un "*tas appariant*" (traduction de "*pairing heap*"), une structure de données alternative aux tas binaires classiques[1], proposée en 1986 par Fredman, Sedgewick, Sleator et Tarjan. Si le coût dans le pire cas *amorti* des opérations sur un tas appariant est asympotiquement identique à celui sur un tas binaire, le coût dans le meilleur cas est bien plus intéressant.
Ainsi, une file de priorités est implémentée ci-dessous par un tas appariant de type `Arbre` (cf. définition 7), et on considére les opérations `Inserer` et `DetruireMax` spécifiées ci-dessous, où `Element` est le type des éléments. On assimile ici le type `Element` à celui des priorités : on néglige donc le problème des données attachées aux éléments, et le type `Element` est supposé muni d'un ordre total. Par ailleurs, on rappelle qu'une file peut contenir plusieurs fois un même élément.

```
procedure Inserer(A: in out Arbre; X: in Element) ;
-- insère X dans la file A

procedure DetruireMax(A: in out Arbre; Max: out Element) ;
-- requiert A non vide
-- garantit Max est un élément de priorité maximale qui a été
--          retiré de A
```

Définition 5 (coût amorti) *On considère une file donnée* `A`*. On appelle* coût amorti *le nombre "*$T(m)/m$*", où* $T(m)$ *est le nombre de comparaisons dans le* pire cas *effectuées lors d'une séquence* quelconque *de* m *appels* successifs *d'opérations de* `Insere` *et* `DetruireMax` *sur la file* `A` *depuis un état initial où* `A` *est vide.*

La structure de "*tas appariant*" se base sur une structure d'arbres binaires. On introduit donc les définitions de type ci-dessous. Dans la suite, si A est un arbre, on note $|A|$ sa taille (son nombre de nœuds).

```
type Noeud ;
type Arbre is access Noeud ;
type Noeud is record
   Etiq: Element ;
   FilsG, FilsD: Arbre ;
end record ;
```

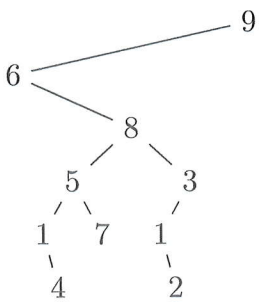
Définition 6 (arbre binaire demi-ordonné) *On dit qu'un arbre binaire* `A` *est* demi-ordonné *ssi lorsque* `A` *est non vide (e.g.* `A /= null`*), alors ses deux fils sont eux-mêmes* demi-ordonnés *et si pour tout sous-arbre* `B` *non vide de* `A.FilsG`*, on a* `A.Etiq >= B.Etiq`*.*

Ainsi, intuitivement, `A.FilsG` contient les éléments dont *on sait* qu'ils sont *inférieurs ou égaux* à `A.Etiq`. Alors que pour `A.FilsD`, on ne sait pas encore.

1. "tas binaires classiques" = arbres binaires parfaits partiellement ordonnés.

Définition 7 (Tas appariant) *Un* tas appariant *est défini comme un arbre binaire demi-ordonné sans fils droit.*

Autrement dit, un tas appariant est un arbre `A` demi-ordonné tel que `A = null` ou `A.FilsD = null`. Ainsi, si `A /= null`, alors `A.Etiq` contient un élément maximum. A titre d'exemple, l'arbre ci-contre est un tas appariant.



Question 7 (2 pts) Écrire une fonction `VerifTas` qui, sous la précondition que `A` a bien une structure d'arbre binaire, retourne `True` ssi `A` est un *tas appariant*. On pourra introduire une fonction récursive auxiliaire.

```
function VerifTas(A: Arbre) return Boolean ;
```

Indiquer le nombre de comparaisons effectuées par votre algorithme en fonction du nombre de nœuds $|A|$.

Opérations sur les tas appariants (avec tassage naïf)

Dans toute la suite, on ne compare les éléments des arbres binaires demi-ordonnés que par l'intermédiaire de la procédure `Compare` ci-dessous. Comme cette procédure fait exactement une comparaison entre 2 éléments, compter le nombre de comparaisons d'éléments effectuées par les opérations reviendra donc à compter le nombre d'appels à `Compare`.
Sous la précondition que `A` et `B` sont des arbres demi-ordonnés non vides, `Compare(A,B)` détache les fils droits de `A` et `B`, pour faire de `A` et `B` deux tas appariants, dont elle calcule ensuite l'union multiple (e.g. avec répétition des éléments communs) pour obtenir un tas appariant qu'elle place dans `A`.

```
1  procedure Compare(A: in out Arbre; B: in Arbre) is
2     Minimum: Arbre ;
3  begin
4     if A.Etiq >= B.Etiq then
5        Minimum := B ;
6     else
7        Minimum := A ;
8        A := B ;
9     end if ;
10    Minimum.FilsD := A.FilsG ;
11    A.FilsG := Minimum ;
12    A.FilsD := null ;
13 end ;
```

Question 8 (1 pts) En utilisant au plus un appel à `Compare` (et aucune autre comparaison d'éléments), écrire la procédure `Inserer` qui ajoute `X` comme un nouvel élément du tas appariant `A`.

```
procedure Inserer(A: in out Arbre; X: in Element) ;
```

Pour coder l'opération `DetruireMax`, on introduit l'opération `Tasser` suivante. Dans un premier temps, on l'implémente via un algo naïf qu'on va optimiser par la suite.

```
1  procedure Tasser(A: in out Arbre) is
2     -- requiert A /= null
3  begin
4     if A.FilsD = null then
5        return ;
6     end if ;
7     Tasser(A.FilsD) ;
8     Compare(A,A.FilsD) ;
9  end ;
```

Question 9 (1,5 pts) Si `A` est initialement un arbre non vide *demi-ordonné* quelconque, que fait `Tasser` ? En déduire comment utiliser `Tasser` pour implémenter `DetruireMax` sous la précondition que `A` est un tas appariant non vide. On désallouera le nœud devenu inutile.

```
procedure DetruireMax(A: in out Arbre; Max: out Element) ;
```

On considère maintenant le code suivant qui utilise les procédures précédentes pour trier un tableau par ordre croissant :

```
1  type Tab is array(Positive range <>) of Element ;
2  procedure Trier(T: in out Tab) is
3     A: Arbre := null ;
4  begin
5     for I in T'Range loop
6        Inserer(A,T(I)) ;
7     end loop ;
8     for I in reverse T'Range loop
9        DetruireMax(A,T(I)) ;
10    end loop ;
11 end ;
```

Question 10 (3 pts) On s'intéresse au *nombre de comparaisons entre éléments* effectuées par la procédure `Trier` en fonction de la taille n du tableau d'entrée `T`.

1. Calculer en fonction de n le nombre *exact* de comparaisons effectuées lorsque le tableau est strictement croissant initialement.
2. Même chose lorsque le tableau est strictement décroissant initialement.
3. En déduire des bornes Θ en fonction de n pour les coûts dans le meilleur cas et le pire cas de la procédure `Trier`.
4. En déduire une borne Θ pour le coût amorti en fonction de m (le nombre d'opérations de la déf 5).

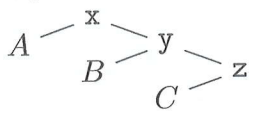
On prendra soin de bien justifier les réponses.

Tassage par paires

Fredman & co ont proposé d'effectuer `Tasser` en appliquant l'algorithme décrit ci-dessous, et qui explique le nom de la structure de données.

Définition 8 (algorithme du tassage par paires) *Comme dans la version naïve, on compare les sous-arbres de la branche complètement à droite en commençant par le nœud sans fils droit. Ce nœud* `Q` *est donc un tas appariant. Si* `Q` *a un père* `P` *et un grand-père* `G`*, on applique la séquence "*`Compare(G,P); Compare(G,Q)`*". On effectue ensuite* `Q:=G` *et on recommence tant que* `Q` *a un père et un grand-père. À la fin, soit le tas* `Q` *est la racine de l'arbre initial (auquel cas, il n'y a plus rien à faire), soit son père est cette racine. Dans ce dernier cas, on effectue un dernier* `Compare` *entre* `Q` *et son père.*

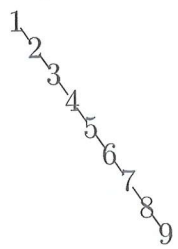
Question 11 (1 pts) On considère la situation décrite définition 8 où Q est un tas appariant, avec un père P (dont Q est le fils droit) et un grand-père G (dont P est le fils droit). Ici, on note A, B et C, les fils gauches respectifs de G, P, et Q. Et on note `x`, `y` et `z` les étiquettes respectives. L'état initial de G est donc représenté par :



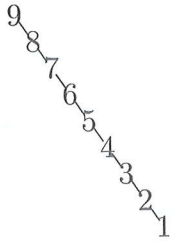
Dessiner les 4 états finaux possibles de G en fonction des résultats au 2 comparaisons successives effectuées dans "`Compare(G,P); Compare(G,Q)`".

Question 12 (1 pts) Indiquer les états obtenus si on emploie cet algorithme alternatif pour effectuer `Tasser(A)` :

1. quand `A` vaut initialement



2. puis, quand `A` vaut initialement



Question 13 (1,5 pts) Écrire une nouvelle implémentation de `Tasser` en employant cet algorithme. On introduira au préalable une procédure récursive intermédiaire appelée `TasserParPairesRec` qui s'occupe d'appliquer l'algorithme en dehors du cas particulier de la racine. Le cas particulier de la racine sera ainsi traité dans `Tasser` lui-même.

```
procedure TasserParPairesRec(A: in out Arbre) ;
-- requiert A /= null
-- garantit A réorganisé tel que  A.FilsD = null
--                            ou  A.FilsD.FilsD = null.
```

Analyse amortie pour le tassage par paires

On va faire ici les calculs de coût amorti via la méthode du potentiel dont on rappelle le principe ci-dessous. On commence par définir le potentiel utilisé ici.

Définition 9 (rang d'un arbre binaire) *Le* rang $r(A)$ *d'un arbre A est le réel positif ou nul défini par :*
- $r(A) = 0$ *si* $A = \texttt{null}$.
- *et,* $r(A) = \log_2 |A|$ *si* $A \neq \texttt{null}$.

Définition 10 (potentiel d'un arbre binaire) *Le potentiel $\phi(A)$ d'un arbre A est un réel positif ou nul défini par récurrence sur les arbres via :*
- $\phi(A) = 0$ *si* $A = \texttt{null}$.
- *et,* $\phi(A) = r(A) + \phi(B) + \phi(C)$ *si A est un arbre non vide dont les deux fils sont B et C (l'ordre des fils n'a ici aucune importance).* [2]

Définition 11 (coût compensé d'une instruction) *Le coût compensé $cc(S)$ d'une instruction S modifiant l'arbre A est défini par*

$$cc(S) = cr(S) + \phi(A) - \phi_0(A)$$

où $cr(S)$ est le coût réel de S (ici le nombre de comparaisons), $\phi(A)$ le potentiel de A dans l'état final de S et $\phi_0(A)$ son potentiel dans l'état initial.

L'intérêt du coût compensé est qu'il vérifie les propriétés suivantes :

1. Pour une séquence d'instructions "$S; S'$" on a

   $$cc(S; S') = cc(S) + cc(S')$$

   En effet, en notant $\phi_1(A)$ le potentiel dans l'état final de S, on a
   $cc(S) = cr(S) + \phi_1(A) - \phi_0(A)$ et $cc(S') = cr(S') + \phi(A) - \phi_1(A)$.
   Par ailleurs, $cr(S; S') = cr(S) + cr(S')$.
2. Si dans l'état initial de S, A est l'arbre vide, alors

   $$cr(S) \leq cc(S)$$

   En effet, $cr(S) = cc(S) + \phi_0(A) - \phi(A)$ avec $\phi_0(A) = 0$ et $\phi(A) \geq 0$.

Concrètement, **on va montrer** ici que le coût compensé des opérations sur le tas appariant A est **majoré par** "$1+3.r(A)$" où $r(A)$ représente le rang de A après l'appel de l'opération. Comme $|A| \leq m$, il s'en suit que le coût amorti est en $\mathcal{O}(\log_2\ m)$.

Question 14 (2 pts) Dans l'opération `Inserer(A,X)`, on note $\phi(A)$ le potentiel de A après l'appel et $\phi_0(A)$ celui avant appel.

1. Montrer que $\phi(A) - \phi_0(A) = r(A)$, où $r(A)$ est la valeur après l'appel.
2. En déduire que le coût compensé de `Inserer` est bien majoré par $1 + 3.r(A)$.

Les questions 15 et 16 suivantes ont pour but de montrer par récurrence sur A que le nombre de comparaisons compensé de `TasserParPairesRec(A)` est majoré par $3.r(A)$.

2. D'après ce qui précède, on a donc ici $r(A) = \log_2(1 + |B| + |C|)$.

Question 15 (1 pts) Justifier que pour montrer cette propriété par récurrence, il suffit de montrer que, dans la transformation effectuée à la question 11, on a :

$$2 + \phi(G) - \phi_1(G) + 3.r_1(Q) \leq 3.r(G)$$

avec
- $\phi(G)$ et $r(G)$ qui représentent les valeurs après la transformation ;
- $\phi_1(G)$ et $r_1(Q)$ qui représente les valeurs avant la transformation.

On justifiera soigneusement en utilisant les notations ϕ_0 et r_0 pour les valeurs avant l'appel à `TasserParPairesRec`.

Question 16 (1 pts) On montre maintenant l'inégalité de la question 15.

1. Expliquer pourquoi on peut se contenter de montrer cette inégalité pour un seul des 4 cas.
2. Dans la suite, on considère le cas $\mathtt{x} < \mathtt{y}$ et $\mathtt{y} < \mathtt{z}$. Expliquer pourquoi il suffit de montrer que
$$2 + r_1(\mathtt{z}) + r(\mathtt{x}) \leq 2.r(\mathtt{z})$$
(ici, on utilise les étiquettes pour dénoter les nœuds).
3. Conclure en remarquant que pour tous réels a et b, $4.a.b \leq (a+b)^2$.

Question 17 (1 pts) Conclure :

1. Terminer l'analyse amortie en majorant le coût compensé de `DetruireMax`.
2. En déduire les bornes en Θ pour le pire cas et le meilleur cas de `Trier` (cf. question 10) lorsqu'on utilise le tassage par paires.