université de BORDEAUX

Année Universitaire 2014/2015
DST de Printemps

Parcours : Licence LIMI201 & LIMI211
Code UE : J1MI2013
Épreuve : Algorithmes et Programmes
Date : Lundi 8 juin 2015 **Heure :** 11 heures
Durée : 2 heures
Documents : non autorisés
Épreuve de M. Alain Griffault

Collège Sciences et Technologies

SUJET + CORRIGE

Avertissement

- **À chaque question, vous pouvez répondre par un algorithme, ou bien par un programme python.**
- Les indentations des fonctions écrites en `Python` doivent être respectées.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Question	Points	Score
Approches algorithmiques	2	
Évaluation de prédicats	3	
L'algorithme de partitionnement	3	
L'évaluation d'expressions	12	
Total:	20	

Exercice 1 : Approches algorithmiques (2 points)

(a) (1 point) Donner les trois étapes principales de l'approche dite "incrémentale".

Solution: Dans une telle approche, la résolution d'un problème va se faire par une itération de modifications élémentaires (ou incrémentales). Chacune diminuant la *distance* au résultat final. Trois étapes :

1. **Formaliser** avec des variables une situation intermédiaire dans laquelle le problème est partiellement traité.
2. **Décrire** une modification élémentaire.
3. **Trouver** d'une part les conditions initiales sur les variables ; d'autre part les valeurs *terminales* des variables qui permettront de déterminer le prédicat de *sortie* de l'itération.

(b) (1 point) Donner les trois étapes principales de l'approche dite "diviser pour règner" .

Solution:

1. **Diviser** le problème de taille n en plusieurs sous-problèmes de tailles plus petites.
2. **Résoudre** les sous-problèmes (généralement de façons récursives).
3. **Combiner** les solutions aux sous-problèmes pour obtenir une solution au problème initial.

Exercice 2 : Évaluation de prédicats (3 points)

Soient `P(n)` une fonction prédicat sur les entiers et `T` un tableau d'entiers.
Vous avez vu en cours l'algorithme suivant qui retourne s'il existe l'indice du plus petit entier du tableau `T` vérifiant une fonction prédicat `P`, sinon un message indiquant la non existence d'un tel entier dans le tableau.

Algorithme 1: MinXverifiantP(T)

Données : Un tableau T d'entiers et P une fonction prédicat sur les entiers
Résultat : S'il existe, l'indice du plus petit entier de T vérifiant P

```
1  Complexité : Θ(n)
2  n ← longueur(T);
3  auMoinsUnP ← Faux;
4  pour i=0 à n-1 faire
5  |  si P(T[i]) and ((non auMoinsUnP) or T[i]<T[iMin]) alors
6  |  |  auMoinsUnP ← Vrai;
7  |  |  iMin ← i;
8  si auMoinsUnP alors
9  |  retourner iMin;
10 sinon
11 |  retourner "Le tableau ne contient pas d'entier verifiant P";
12
```

(a) (1 point) En simulant l'algorithme pour un tableau non vide et pour la valeur `i=0`, expliquer pourquoi cet algorithme est incorrect si l'on ne fait pas l'hypothèse que l'évaluation de la condition ligne 5 se fait de gauche à droite et sans évaluer les termes inutiles.
Rappel : Ce type d'évaluation a été qualifiée de "paresseuse" dans le cours, elle est dénommée également "court-circuit" dans la littérature.

Solution: Soit T un tableau non vide. Lors de l'évaluation de `P(T[i]) and ((non auMoinsUnP) or T[i]<T[iMin])` pour $i = 0$, `P(T[i])` sera défini, `(non auMoinsUnP)` également mais pas `T[i]<T[iMin]` car `iMin` n'est pas initialisé. L'évaluation provoquera donc un "bug".

(b) (2 points) Écrire un programme `Python` ou un algorithme `MinXverifiantPV2(T)` correct qui n'utilise pas l'évaluaton "paresseuse", c'est-à-dire que tous les termes d'une condition sont évalués pour en décider du résultat.

Solution:

Algorithme 2: MinXverifiantPV2(T)

Données : Un tableau T d'entiers et P une fonction prédicat sur les entiers
Résultat : S'il existe, l'indice du plus petit entier de T vérifiant P

```
n ← longueur(T);
auMoinsUnP ← Faux;
pour i=0 à n-1 faire
|  si P(T[i]) alors
|  |  si non auMoinsUnP alors
|  |  |  auMoinsUnP ← Vrai;
|  |  |  iMin ← i;
|  |  sinon si T[i]<T[iMin] alors
|  |  |  iMin ← i;
si auMoinsUnP alors
|  retourner iMin;
sinon
|  retourner "Le tableau ne contient pas d'entier verifiant P";
```

Exercice 3 : L'algorithme de partitionnement (3 points)

(a) (1 point) Vous avez vu en cours l'équivalence suivante entre les structures "répéter" et "tant que" :

Algorithme 3: TantQue(P)

Données : Un predicat

tant que P **faire**
  <ListeInstructions>;

≡

Algorithme 4: RepeterEquivTantQue(P)

Données : Un prédicat

si P **alors**
  répéter
    <ListeInstructions>;
  jusqu'à *not* P;

Complèter l'équivalence suivante :

Algorithme 5: Repeter(P)

Données : Un predicat

répéter
  <ListeInstructions>;
jusqu'à P;

≡

Solution:

Algorithme 6: TantQueEquivRepeter(P)

Données : Un prédicat

<ListeInstructions>;
tant que *not* P **faire**
  <ListeInstructions>;

(b) (2 points) Vous avez vu en cours l'algorithme suivant qui partitionne un tableau en deux sous-tableaux de telle sorte que tous les éléments du premier soient inférieurs ou égaux à tous ceux du second.

Algorithme 7: Partitionner(T,gauche,droite)

Données : Un tableau T d'entiers, et deux indices gauche et droite
Résultat : Un indice indPivot tel que T[gauche..indPivot] ≤ T[indPivot+1..droite]

```
pivot ← T[gauche];
i ← gauche-1;
j ← droite+1;
tant que Vrai faire
    répéter
        i ← i+1;
    jusqu'à T[i] ≥ pivot;
    répéter
        j ← j-1;
    jusqu'à T[j] ≤ pivot;
    si i < j alors
        Echanger(T,i,j);
    sinon
        retourner j ;                    /* indPivot est égal à j */
```

Écrire un programme `Python` ou un algorithme `PartitionnerTQ(T,gauche,droite)` équivalent dans lequel les structures `répéter` sont remplacées par des structures `tant que`.

Solution:

Algorithme 8: PartitionnerTQ(T,gauche,droite)

Données : Un tableau T d'entiers, et deux indices gauche et droite
Résultat : Un indice indPivot tel que T[gauche..indPivot] $\leq$ T[indPivot+1..droite]

```
pivot ← T[gauche];
i ← gauche-1;
j ← droite+1;
tant que Vrai faire
    i ← i+1;
    tant que non (T[i] ≥ pivot) faire
        i ← i+1;
    j ← j-1;
    tant que non (T[j] ≤ pivot) faire
        j ← j-1;
    si i < j alors
        Echanger(T,i,j);
    sinon
        retourner j ;                                   /* indPivot est égal à j */
```

Exercice 4 : L'évaluation d'expressions (12 points)

L'objectif est l'écriture d'un algorithme qui évalue une expression arithmétique bien parenthésée. Informellement, les expressions arithmétiques considérées n'utilisent que l'addition, notée $+$, et la multiplication, notée $\times$.

Dans la première partie, le but est de tester si une expression arithmétique est bien parenthésée, et dans la seconde, le but est l'évaluation d'une telle expression. Par exemple :

Expression	Bien parenthésée	Évaluation	Explication
356	Vrai	356	
$(5+15)$	Vrai	20	
$(((5+15)\times(-3+6))+5)$	Vrai	65	
(356)	Faux	None	Trop parenthésée
$5+15$	Faux	None	Pas suffisamment parenthésée
$(5+15+10)$	Faux	None	Pas suffisamment parenthésée
$(5+15\times 10)$	Faux	None	Pas suffisamment parenthésée
$(5+c)$	Faux	None	c n'est pas un nombre
$(5+15($	Faux	None	Mauvais parenthésage

Les expressions seront stockées dans des tableaux. Ainsi $(((5+15)\times(-3+6))+5)$ sera représentée par :

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
valeur	(	(	(	5	+	15	)	×	(	-3	+	6	)	)	+	5	)

Pour cet exercice, vous pourrez utiliser les algorithmes ou les fonctions `Python` suivantes, de complexité $\Theta(1)$:

```
EstNombre(T,i){
  // Retourne Vrai si T[i] est un nombre,
  // Faux sinon
}
EstParentheseOuvrante(T,i){
  retourne T[i]='(';
}
EstParentheseFermante(T,i){
  retourne T[i]=')';
}
EstAddition(T,i){
  retourne T[i]='+';
}
EstMultiplication(T,i){
  retourne T[i]='x';
}
```

```
def estNombre(T,i):
    return isinstance(T[i], numbers.Number)

def estParentheseOuvrante(T,i):
    return T[i]=='('

def estParentheseFermante(T,i):
    return T[i]==')'

def estAddition(T,i):
    return T[i]=='+'

def estMultiplication(T,i):
    return T[i]=='x'
```

(a) Algorithme de décision d'une expression arithmétique bien parenthésée.

i. (1 point) **Définition :** Les expressions arithmétiques considérées n'utilisent que des additions ou des multiplications de nombres.

Écrire un programme `Python` ou un algorithme `estOperateur(T,i)` qui retourne `Vrai` si `T[i]` est une addition ou bien une multiplication, et `Faux` sinon.

Les complexités en temps et en espace doivent être en $\Theta(1)$.

Solution:

```
EstOperateur(T,i){
  retourne EstAddition(T,i)
    ou EstMultiplication(T,i);
}
```

```
def estOperateur(T,i):
    return estAddition(T,i) \
        or estMultiplication(T,i)
```

ii. (3 points) **Définition :** Une expression arithmétique E est **bien parenthésée** si et seulement si :

$$E = \begin{cases} \text{Un nombre} \\ \text{ou bien} \\ (E1 \text{ op } E2) \end{cases}$$

avec $E1$ et $E2$ deux expressions bien parenthésées et "op" un opérateur arithmétique binaire.

Cette définition est récursive. Une approche "diviser pour règner" doit permettre d'écrire les algorithmes souhaités. Pour "diviser" le problème en sous-problèmes, c'est à dire séparer l'expression en deux sous-expressions, il faut savoir localiser la dernière opération effectuée lors de l'évaluation d'une expression.

Propriété (que l'on admet) : Dans une expression arithmétique bien parenthésée, le dernier opérateur évalué est celui dont la position est caractérisée par : *la différence entre le nombre de parenthèses ouvrantes et le nombre de parenthèses fermantes qui le précèdent est égal à 1.*

Par exemple, pour l'expression suivante :

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	(	(	(	5	+	15	)	×	(	-3	+	6	)	)	+	5	)
différence					3			2			3				1		

Cela signifie que pour calculer $(((5+15)\times(-3+6))+5)$, il faut calculer $((5+15)\times(-3+6))$, puis calculer 5, puis faire l'addition des deux résultats. Remarquer que l'opérateur '+' en position 14 est bien le dernier effectué, et la différence du nombre de parenthèses ouvrantes et fermantes qui sont avant lui vaut bien 1.

La récursivité s'applique aux sous-tableaux, ainsi pour l'expression comprise entre les indices 1 et 13 inclus :

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	(	(	(	5	+	15	)	×	(	-3	+	6	)	)	+	5	)
différence	-				2			1			2				-	-	-

Cela signifie que pour calculer $((5+15)\times(-3+6))$, il faut calculer $(5+15)$, puis calculer $(-3+6)$, puis faire la multiplication des deux résultats. Remarquer que l'opérateur '×' en position 7 est bien le dernier effectué, et la différence du nombre de parenthèses ouvrantes et fermantes qui sont avant lui vaut bien 1.

En utilisant la propriété précédente sur le dernier opérateur évalué, écrire un programme `Python` ou un algorithme `dernierOperateurEvalue(T,gauche,droite)` qui retourne, s'il existe, l'indice du dernier opérateur évalué dans le calcul de l'expression bien parenthésée comprise entre les indices `gauche` et `droite` inclus du tableau `T`. L'algorithme retourne "None" sinon.
Les complexités en temps doivent être en $\Omega(1), \mathcal{O}(droite - gauche)$, et en espace en $\Theta(1)$.

Solution:

```
def dernierOperateurEvalue(T,gauche,droite):
    nb = 0
    for i in range(gauche,droite+1):
        if estParentheseOuvrante(T,i):
            nb += 1
        elif estParentheseFermante(T,i):
            nb -= 1
        elif estOperateur(T,i) and nb==1:
            return i
    return None
```

Algorithme 9: DernierOperateurEvalue(T,gauche,droite)

Données : Un tableau T d'entiers, et deux indices gauche et droite
Résultat : L'indice doe tel que T[gauche+1..doe-1] "op" T[doe+1..droite-1]

```
nb ← 0;
pour i=gauche à droite faire
    si EstParentheseOuvrante(T,i) alors
        nb ← nb + 1;
    sinon si EstParentheseFermante(T,i) alors
        nb ← nb - 1;
    sinon si EstOperateur(T,i) et (nb = 1) alors
        retourner i;
retourner nil;
```

iii. (1 point) Écrire un programme `Python` ou un algorithme `aParenthesesExtremites(T,gauche,droite)` qui retourne `Vrai` si l'expression comprise entre les indices `gauche` et `droite` inclus du tableau `T`, a une parenthèse ouvrante et une fermante à chacune de ses extremités, `Faux` sinon.

Solution:

```
def aParenthesesExtremites(T,gauche,droite):
    return estParentheseOuvrante(T,gauche) \
        and estParentheseFermante(T,droite) \
```

Algorithme 10: AParenthesesExtremites(T,gauche,droite)

Données : Un tableau T contenant une expression et deux indices
Résultat : Vrai si les indices forme un couple de parentheses, Faux sinon

```
retourner EstParentheseOuvrante(T,gauche) et EstParentheseOuvrante(T,droite);
```

iv. (2 points) En utilisant les algorithmes donnés ainsi que ceux des deux questions précédentes, écrire un programme `Python` ou un algorithme `estExpressionRec(T,gauche,droite)` qui retourne `Vrai` si l'expression comprise entre les indices `gauche` et `droite` inclus du tableau `T` est bien parenthésée, `Faux` sinon.

Solution:

```
def estExpressionRec(T,gauche,droite):
    if gauche==droite:
        return estNombre(T,gauche)
    elif aParenthesesExtremites(T,gauche,droite):
        doe = dernierOperateurEvalue(T,gauche,droite)
        if doe!=None:
            return estExpressionRec(T,gauche+1,doe-1) \
                and estExpressionRec(T,doe+1,droite-1)
    return False
```

Algorithme 11: EstExpressionRec(T,gauche,droite)

Données : Un tableau T d'entiers, et deux indices gauche et droite
Résultat : Vrai si T[gauche..droite] est une expression, Faux sinon

si *gauche = droite* **alors**
  retourner EstNombre(T,gauche);
sinon si *AParenthesesExtremites(T,gauche,droite)* **alors**
  doe ← DernierOperateurEvalue(T,gauche,droite);
  si *doe ≠ None* **alors**
    retourner EstExpressionRec(T,gauche+1,doe-1) **et**
    EstExpressionRec(T,doe+1,droite-1);
retourner Faux;

v. (1 point) Écrire un programme `Python` ou un algorithme `estExpressionParenthesee(T)` qui retourne `Vrai` si l'expression contenue dans le tableau `T` est bien parenthésée, `Faux` sinon.

Solution:

```
def estExpressionParenthesee(T):
    return estExpressionRec(T,0,len(T)-1)
```

Algorithme 12: EstExpressionParenthesee(T)

Données : Un tableau T contenu une expression arithmétique
Résultat : Vrai si c'est une expression bien parenthesee, Faux sinon

retourner EstExpressionRec(T,0,longueur(T)-1);

(b) Algorithme d'évaluation d'une expression arithmétique bien parenthésée.

i. (1 point) Écrire un programme `Python` ou un algorithme `valeurNombre(T,i)` qui retourne la valeur du nombre `T[i]` si c'est un nombre, la valeur `None` sinon.

Solution:

```
def valeurNombre(T,i):
    if estNombre(T,i):
        return T[i]
    return None
```

Algorithme 13: ValeurNombre(T,i)

Données : Un tableau T d'entiers, et un indice i
Résultat : La valeur de l'entier T[i] si c'est un entier

si *EstNombre(T,i)* **alors**
  retourner T[i];
retourner nil;

ii. (2 points) En adaptant le programme `Python` ou l'algorithme `estExpressionRec(T,gauche,droite)`, écrire un programme `Python` ou un algorithme `valeurExpressionRec(T,gauche,droite)` qui retourne la valeur de l'expression comprise entre les indices `gauche` et `droite` du tableau `T`, la valeur `None` si cette expression n'est pas bien parenthésée.

Solution:

```
def valeurExpressionRec(T,gauche,droite):
    if gauche==droite:
        return valeurNombre(T,gauche)
    elif aParenthesesExtremites(T,gauche,droite):
        doe = dernierOperateurEvalue(T,gauche,droite)
        if doe!=None:
```

```
            vE1 = valeurExpressionRec(T,gauche+1,doe-1)
            if vE1!=None:
                vE2 = valeurExpressionRec(T,doe+1,droite-1)
                if vE1!=None:
                    if estAddition(T,doe):
                        return vE1+vE2
                    elif estMultiplication(T,doe):
                        return vE1*vE2
    return None
```

Algorithme 14: ValeurExpressionRec(T,gauche,droite)

Données : Un tableau T d'entiers, et deux indices gauche et droite
Résultat : Vrai si T[gauche..droite] est une expression, Faux sinon

si *gauche = droite* **alors**
  retourner ValeurNombre(T,gauche);
sinon si *AParenthesesExtremites(T,gauche,droite)* **alors**
  doe ← DernierOperateurEvalue(T,gauche,droite);
  si *doe ≠ nil* **alors**
    vE1 ← ValeurExpressionRec(T,gauche+1,doe-1);
    si *vE1 ≠ nil* **alors**
      vE2 ← ValeurExpressionRec(T,doe+1,droite-1);
      si *vE2 ≠ nil* **alors**
        si *EstAddition(T,i)* **alors**
          retourner vE1 + vE2;
        sinon si *EstMultiplication(T,i)* **alors**
          retourner vE1 * vE2;
retourner nil;

iii. (1 point) Écrire un programme `Python` ou un algorithme `valeurExpressionParenthesee(T)` qui retourne la valeur de l'expression comprise dans le tableau `T`, la valeur `None` si cette expression n'est pas bien parenthésée.

Solution:

```
def valeurExpressionParenthesee(T):
    return valeurExpressionRec(T,0,len(T)-1)
```

Algorithme 15: ValeurExpressionParenthesee(T)

Données : Un tableau T contenu une expression arithmétique
Résultat : La valeur de l'expression si elle est bien parenthesee, nil sinon

retourner ValeurExpressionRec(T,0,longueur(T)-1);