ASI3 Mercredi 20 Janvier 2010

Algorithmique et Base de la programmation

Durée : *3h00*
Documents autorisés : **AUCUN**

Remarques :
- Veuillez lire attentivement les questions avant de répondre.
- Le barème donné est un barème indicatif qui pourra évoluer lors de la correction.
- Rendez une copie propre.
- Il est interdit d'écrire au crayon à papier sur votre copie.

1 Polynôme (3 points)

Soit le TAD `Polynome` suivant :

Nom:	Polynome	
Utilise:	**Naturel**,**Reel**	
Opérations:	polynome:	→ Polynome
	obtenirDegre:	Polynome → **Naturel**
	obtenirCoefficient:	Polynome × Naturel → **Reel**
	modifierCoefficient:	Polynome × **Reel** × **Naturel** → Polynome

Axiomes:
- obtenirCoefficient(polynome(),i)=0
- obtenirDegre((polynome())=0
- obtenirDegre(p)=d tel que $\forall i > d, obtenirCoefficient(p, i) = 0$
- obtenirCoefficient(modifierCoefficient(p,v,i),i)=v

1.1 Conception préliminaire

Donnez la signature des fonctions et procédures correspondant aux opérations du TAD précédent.

1.2 Utilisation du TAD

Donnez les algorithmes des fonctions suivantes :
- **fonction** addition (p1,p2 : Polynome) **:** Polynome
- **fonction** multiplication (p1,p2 : Polynome) **:** Polynome
- **fonction** derivee (p : Polynome) **:** Polynome

2 Compréhension : Codage de Shannon-Fano (3 points)

« Le codage de Shannon-Fano est un algorithme de compression de données sans perte élaboré par Robert Fano à partir d'une idée de Claude Shannon.

Il s'agit d'un codage entropique produisant un code préfixe très similaire à un code de Huffman, bien que non-optimal, contrairement à ce dernier.

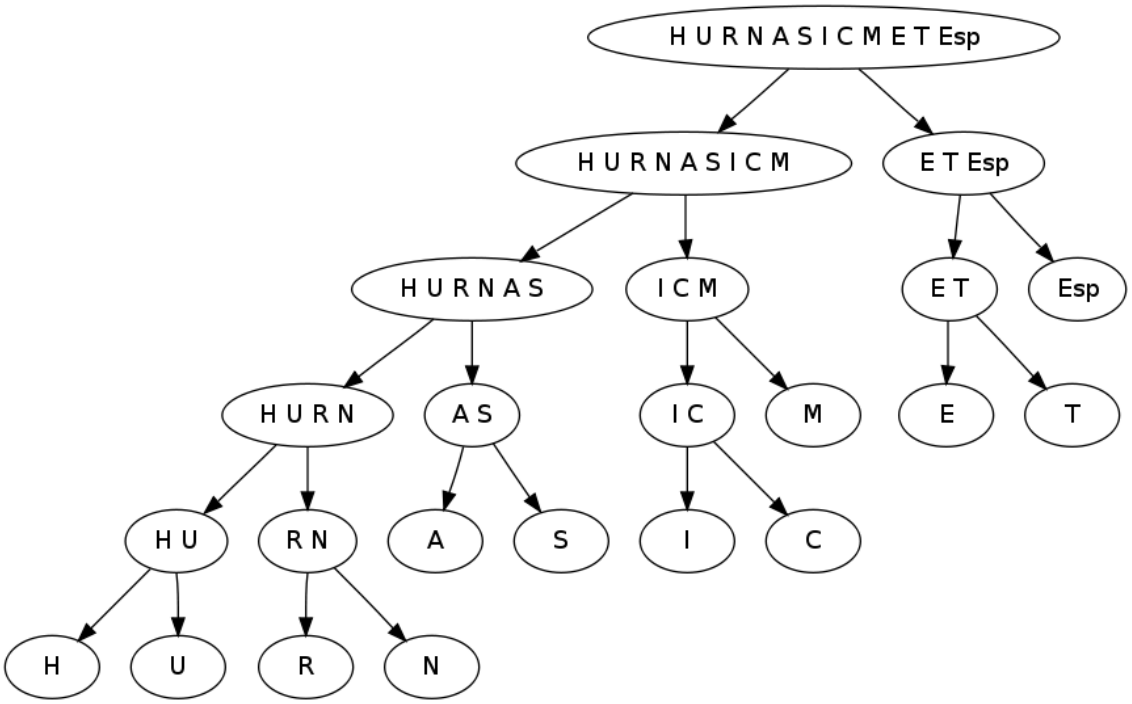
Tous les symboles à compresser sont triés selon leur probabilité, et l'ensemble trié des symboles est coupé en deux parties de telle façon que les probabilités des deux parties soient le plus proche possible de l'égalité (la probabilité d'une partie étant égale à la somme des probabilités des différents symboles de cette partie). Tous les symboles de la première partie sont codés par un 0 suivi de leur code de Shannon-Fano en ne prenant en compte que les symboles de la première partie, et tous les symboles de la seconde partie sont codés par un 1 suivi de leur code de Shannon-Fano en ne prenant en compte que les symboles de la seconde partie, récursivement. Lorsqu'une partie ne contient qu'un seul symbole, celui-ci est représenté par un code vide (de longueur nulle). » (source Wikipédia)

Cela revient donc à construire un arbre de codage, comme l'arbre d'Huffman, mais en partant de la racine, alors que pour construire l'arbre d'Huffman on part des feuilles.

Proposez l'arbre de codage de Shannon-Fano pour le texte `"ASI C EST CHOUETTE COMME FORMATION"` en indiquant pour chaque nœud l'ensemble des caractères associés (l'espace est un caractère comme un autre).

Solution proposée :

A	S	I	C	E	T	H	O	U	M	R	N	Esp
2	2	2	3	4	4	1	4	1	3	1	1	5



3 k-ppv (5 points)

Soit le type `Element` avec la fonction suivante qui permet de calculer une distance entre deux éléments :

– **fonction** distance (e1,e2 : Element) **: ReelPositif**

L'algorithme du $k-ppv$, pour k plus proches voisins, est un algorithme qui permet d'identifier dans une collection d'éléments (appelée « base d'apprentissage »), les k éléments qui sont les plus proches d'un élément donné.

1. Rappelez les signatures des fonctions et procédures permettant d'utiliser le TAD Liste dans le paradigme de la programmation structurée.
2. Donnez l'algorithme de la fonction suivante :
   – **fonction** kppv (baseDApprentissage : Liste<Element>, e : Element, k : **NaturelNonNul**) **:** Liste<Element>

     ⌊**précondition(s)** longueur(baseDApprentissage)≥k
3. Quelle est la complexité dans le pire des cas de votre algorithme indépendamment des complexités des opérations sur le TAD `Liste` (justifiez) ?

Solution proposée :

1. Signatures fonctions/procédures :
   – **fonction** liste () **:** Liste
   – **fonction** estVide (uneListe : Liste) **: Booleen**
   – **procédure** inserer (**E/S** uneListe : Liste,**E** position : **Naturel**, element : Element)

     ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe) + 1$
   – **procédure** supprimer (**E/S** uneListe : Liste,**E** position : **Naturel**)

     ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
   – **fonction** obtenirElement (uneListe : Liste, position : **Naturel**) **:** Element

     ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
   – **fonction** longueur (uneListe : Liste) **: Naturel**
2. $k-ppv$

```
fonction indiceDuPlusEloigne (l : Liste<Element>,e : Element) : NaturelNonNul
   Déclaration  resultat,i : NaturelNonNul
debut
   resultat ← 1
   pour i ←2 à longueur(l) faire
      si distance(obtenirElement(l,i),e)>distance(obtenirElement(l,resultat),e) alors
         resultat ← i
      finsi
   finpour
   retourner resultat
fin
fonction kppv (baseDApprentissage : Liste<Element>, e : Element, k : NaturelNonNul)
: Liste<Element>
```

⌊**précondition(s)** longueur(baseDApprentissage)≥k

Déclaration resultat : Liste<Element>
i,j : **NaturelNonNul**
trouvePlusGrand : **Booleen**

debut
resultat ← liste()
pour i ←1 **à** k **faire**
inserer(resultat,i,obtenirElement(baseDApprentissage,i))
finpour
si longueur(baseDApprentissage)>k **alors**
pour i ←k+1 **à** longueur(baseDApprentissage) **faire**
j ← indiceDuPlusEloigne(resultat,e)
si distance(obtenirElement(resultat,j),e)>distance(obtenirElement(baseDApprentissage,i)
,e) **alors**
supprimer(resultat,j)
inserer(resultat,j,obtenirElement(baseDApprentissage,i))
finsi
finpour
finsi
retourner resultat
fin

3. Complexité :
   - Ce sont k et $longueur(baseDApprentissage)$ qui détermine la taille du problème
   - Le premier pour est en complexité de $O(k)$
   - Le deuxième pour est en $O(k * longueur(baseDApprentissage))$
   - Donc l'algorithme de la fonction `kppv` est en $O(k * longueur(baseDApprentissage))$

4 Sudoku (9 points)

Le jeu du Sudoku est composé d'une grille carrée de 9 cases de côté. Ce jeu consiste « à compléter toute la grille avec des chiffres allant de 1 à 9. Chaque chiffre ne doit être utilisé qu'une seule fois par ligne, par colonne et par carré de neuf cases »[1].

On suppose que l'on numérote les lignes, les colonnes et les carrés d'une grille de Sudoku de 1 à 9.

La grille présentée par la figure 1 présente une grille de Sudoku à compléter.

Soit les TAD *Coordonnee* et *GrilleSudoku* suivants :

Nom:	Coordonnee	
Utilise:	**Naturel**	
Opérations:	coordonnee:	$1..9 \times 1..9 \rightarrow$ Coordonnee
	obtenirLigne:	Coordonnee $\rightarrow 1..9$
	obtenirColonne:	Coordonnee $\rightarrow 1..9$
	obtenirCarre:	Coordonnee $\rightarrow 1..9$

[1] Définition donnée par le journal le Monde.

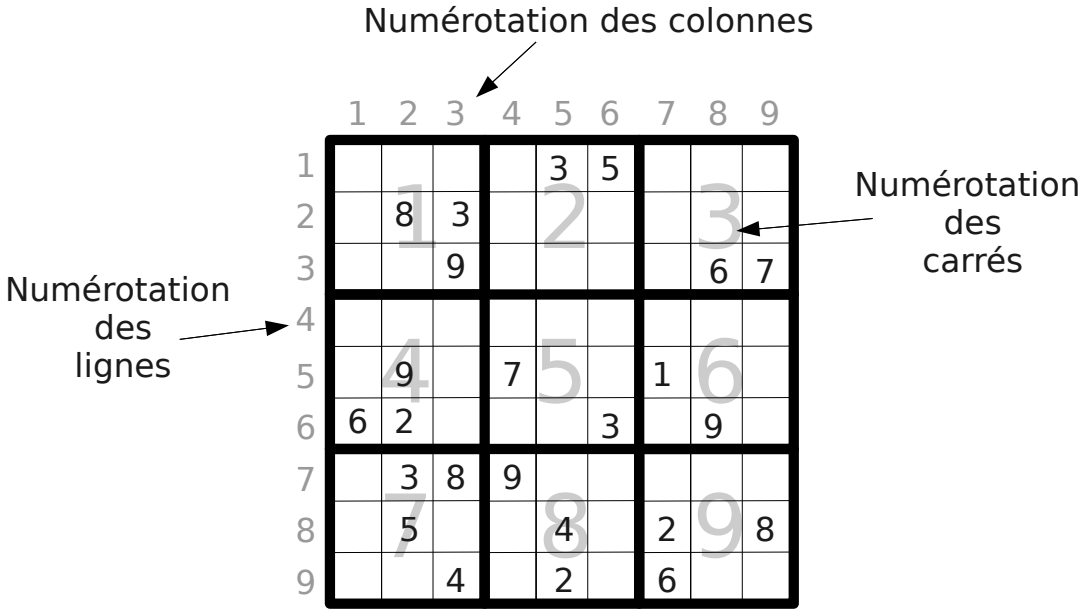


FIG. 1 – Exemple de grille de Sudoku

Axiomes: - obtenirColonne(coordonnee(c,l))=c
- obtenirLigne(coordonnee(c,l))=l

Nom: GrilleSudoku

Utilise: **Naturel**, Coordonnee, **Booleen**

Opérations:

grilleSudoku:	→ GrilleSudoku
caseVide:	GrilleSudoku × Coordonnee → **Booleen**
obtenirChiffre:	GrilleSudoku × Coordonnee ↛ 1..9
fixerChiffre:	GrilleSudoku × Coordonnee × 1..9 ↛ GrilleSudoku
viderCase:	GrilleSudoku × Coordonnee ↛ GrilleSudoku

Sémantiques:

grilleSudoku:	permet de créer une grille de Sudoku vide
caseVide:	permet de savoir si une case d'une grille de Sudoku vide
obtenirChiffre:	permet d'obtenir le chiffre d'une case non vide
fixerChiffre:	permet de fixer un chiffre d'une case vide
viderCase:	permet d'enlever le chiffre d'une case non vide

Préconditions:

obtenirChiffre(g,c):	non caseVide(g,c)
fixerChiffre(g,c,v):	caseVide(g,c)
viderCase(g,c):	non caseVide(g,c)

4.1 Conception préliminaire

Donnez la signature des fonctions et procédures correspondant aux deux TAD précédents.

Solution proposée :

- **fonction** coordonnee (c,l : 1..9) **:** Coordonnee
- **fonction** obtenirLigne (c : Coordonnee) **:** 1..9

- **fonction** obtenirColonne (c : Coordonnee) **:** 1..9
- **fonction** obtenirCarre (c : Coordonnee) **:** 1..9
- **fonction** grilleSudoku () **:** GrilleSudoku
- **fonction** caseVide (g : GrilleSudoku, c : Coordonnee) **: Booleen**
- **fonction** obtenirChiffre (g : GrilleSudoku, c : Coordonnee) **:** 1..9
  ⌊**précondition(s)** non caseVide(g,c)
- **procédure** fixerChiffre (**E/S** g : GrilleSudoku,**E** c : Coordonnee, v : 1..9)
  ⌊**précondition(s)** caseVide(g,c)
- **procédure** viderCase (**E/S** g : GrilleSudoku,**E** c : Coordonnee)
  ⌊**précondition(s)** non caseVide(g,c)

4.2 Conception détaillée

On se propose de concevoir le TAD *Coordonnee* de la façon suivante :

Type Coordonnee = **Structure**
ligne : 1..9
colonne : 1..9
finstructure

Donnez les algorithmes des fonctions correspondant aux opérations de ce TAD.

Solution proposée :

fonction coordonnee (c,l : 1..9) **:** Coordonnee
Déclaration resultat : Coordonnee
debut
resultat.colonne ← c
resultat.ligne ← l
retourner retourner
fin
fonction obtenirLigne (c : Coordonnee) **:** 1..9
debut
retourner c.ligne
fin
fonction obtenirColonne (c : Coordonnee) **:** 1..9
debut
retourner c.colonne
fin
fonction obtenirCarre (c : Coordonnee) **:** 1..9
debut
retourner 3*((c.ligne-1) div 3)+(c.colonne -1) div 3+1
fin

4.3 Fonctions métiers

On se propose d'écrire des fonctions et procédures permettant de vérifier ou d'aider à la résolution manuelle d'une grille de Sudoku.

1. Donnez l'algorithme de la fonction suivante qui permet de savoir si une grille de Sudoku est totalement remplie (sans vérifier sa validité) :
   – **fonction** estRemplie (g : GrilleSudoku) **: Booleen**
2. On suppose que l'on possède les fonctions suivantes qui permettent d'obtenir l'ensemble des chiffres déjà fixés d'une colonne, d'une ligne ou d'un carré :
   – **fonction** obtenirChiffresDUneLigne (g : GrilleSudoku, ligne : 1..9) **:** Ensemble< 1..9 >
   – **fonction** obtenirChiffresDUneColonne (g : GrilleSudoku, colonne : 1..9) **:** Ensemble< 1..9 >
   – **fonction** obtenirChiffresDUnCarre (g : GrilleSudoku, carre : 1..9) **:** Ensemble< 1..9 >

   Donnez l'algorithme de la fonction suivante qui permet de savoir si on peut mettre un chiffre dans une case vide sans contredire la règle donnée en introduction :
   – **fonction** estChiffreValable (g : GrilleSudoku, chiffre : 1..9, case : Coordonnee) **: Booleen**

     ⌊**précondition(s)** caseVide(g,case)
3. En utilisant les fonctions et procédures permettant d'utiliser le TAD Liste (avec les signatures que vous avez explicitées dans la 1ère question de l'exercice 3), donnez l'algorithme la fonction suivante qui donne la liste des solutions possibles pour une case vide :
   – **fonction** obtenirSolutionsPossibles (g : GrilleSudoku, case : Coordonnee) **:** Liste< 1..9 >

     ⌊**précondition(s)** caseVide(g,case)
4. Donnez l'algorithme de la procédure suivante qui cherche la solution *sol* d'une grille de sudoku *g* (si il y a une solution, *trouve* vaut alors *vrai*, *faux* sinon) :
   – **procédure** chercherSolution (**E** g : GrilleSudoku,**S** trouve : **Booleen**, sol : GrilleSudoku)

Solution proposée :

1.

```
fonction estRemplie (g : GrilleSudoku) : Booleen
  Déclaration  i,j : 1..9
               c : Coordonnee
               resultat : Booleen
debut
  resultat ← VRAI
  i ← 1
  j ← 1
  tant que resultat et j<4 faire
    c ← coordonnee(i,j)
    si non estVide(g,c) alors
      resultat ← FAUX
    sinon
      i ← i+1
      si i=4 alors
        i ← 1
        j ← j+1
      finsi
```

```
        finsi
    fintantque
    retourner resultat
fin
```

2.

```
fonction estChiffreValable (g : GrilleSudoku, chiffre : 1..9, case : Coordonnee) : Booleen
    ⌊précondition(s)  caseVide(g,case)
    Déclaration  e1,e2,e3 : Ensemble< 1..9 >
debut
    e1 ← obtenirChiffresDUneLigne(g,obtenirLigne(c))
    e2 ← obtenirChiffresDUneColonne(g,obtenirColonne(c))
    e3 ← obtenirChiffresDUnCarre(g,obtenirCarre(c))
    retourner non estPresent(e1,chiffre) et non estPresent(e2,chiffre) et non estPresent(e3,chiffre)

fin
```

3.

```
fonction obtenirSolutionsPossibles (g : GrilleSudoku, case : Coordonnee) : Liste< 1..9 >
    ⌊précondition(s)  caseVide(g,case)
    Déclaration  resultat : Liste< 1..9 >
                 i : 1..9
debut
    resultat ← liste()
    pour i ←1 à 9 faire
        si estChiffreValable(g,i,case) alors
            inserer(resultat,1,i)
        finsi
    finpour
    retourner resultat
fin
```

4.

```
procédure trouverSolution (E g : GrilleSudoku,S trouve : Booleen, sol : GrilleSudoku)
    Déclaration  temp : GrilleSudoku
                 i,j,k : 1..9
                 l : Liste<1..9>
debut
    si estRemplie(g) alors
        trouve ← Vrai
        sol ← g
    sinon
        trouve ← Faux
        i ← 1
        tant que non trouve et i≤9 faire
            j ← 1
```

```
        tant que non trouve et j≤9 faire
          si caseVide(g,coordonnee(i,j)) alors
            l ← obtenirSolutionsPossibles(g,coordonnee(i,j))
            si longueur(l)>0 alors
              k ← 1
              tant que non trouve et j≤longueur(l) faire
                temp ← g
                fixerChiffre(temp,coordonnee(i,j),k)
                trouverSolution(temp,trouve,sol)
              fintantque
            finsi
          finsi
        fintantque
      fintantque
    finsi
fin
```