M2 CCI – Algorithmique – Devoir surveillé

Durée 3h, sans documents **25 février 2025**

NE PAS RECOPIER les énoncés des questions. Ne pas perdre de temps à un soin excessif de la présentation. Les parties sont indépendantes. Le barème est indicatif.

1. Compactage d'une séquence de bits (séquences sous forme contiguë)

Considérons une séquence **non vide** de bits (entiers valant 0 ou 1), par exemple :

SB = [**1,1,1,1**,0,0,**1**,0,0,0,0,0,**1,1**,0,0,0,**1**,0,0,0,0,0,0,**1,1**] de longueur **L** = 26

Chacun des groupes de cette séquence peut être représenté par un couple comportant sa longueur et la valeur répétée. La séquence initiale est alors représentée par une séquence de tels couples. Dans notre exemple, on obtient ainsi : [<4,1>, <2,0>, <1,1>, <5,0>, <2,1>, <3,0>, <1,1>, <6,0>, <2,1>]

Comme il y a alternance entre les valeurs 0 et les valeurs 1 des bits, on peut encore simplifier cette représentation : connaissant la valeur répétée de la première sous-séquence, il suffit de caractériser chaque sous-séquence uniquement par sa longueur.

On définit ainsi une *forme compacte* de la séquence de bits initiale. C'est une séquence d'entiers : le premier entier est un 0 ou un 1, valeur répétée du premier groupe de bits de même valeur ; les autres sont les longueurs des groupes de bits consécutifs de même valeur de la séquence initiale. (on sait que la valeur des bits de chaque groupe alterne).

Dans notre exemple, le premier bit est un 1 : la forme compacte est ainsi :

FC = [1,4,2,1,5,2,3,1,6,2] de longueur **LC** = 10 (commence par un 1, quatre 1 puis deux 0, puis un 1, puis cinq 0, etc).

Autre exemple :

FC = [0,4,2,1,5,2,3,1,6,2] de longueur **LC** = 10 est la forme compacte de **SB** = [**0,0,0,0**,1,1,**0**,1,1,1,1,1,1,**0,0**,1,1,1,**0**,1,1,1,1,1,1,**0,0**] de longueur **L** = 26.

Q1 **[4 points]**

Les séquences sont représentées sous forme contiguë avec longueur explicite. Le lexique suivant décrit une séquence non vide de bits (représentée dans le tableau **SB** et de longueur **L**) et une forme compacte (représentée dans le tableau **FC** et de longueur **LC**) :

```
Bit : type entier sur [0,1]
Lmax : constante de type entier > 0          {longueur maximum des séquences de bits}
SB : tableau sur [1…Lmax] de Bit
L : entier sur [1…Lmax]
FC : tableau sur [1…Lmax+1] d'entier
LC : entier sur [2…Lmax+1]
```

(i) Décompactage

— Donner une **réalisation itérative** d'un algorithme qui produit la séquence de bits correspondant à une forme compacte donnée : **FC** et **LC** sont supposés donnés, l'algorithme construit les valeurs de **SB** et **L**.

(ii) Compactage

— Donner une **réalisation itérative** d'un algorithme qui produit la forme compacte d'une séquence non vide de bits donnée : **SB** et **L** sont supposés donnés, l'algorithme construit les valeurs de **FC** et **LC**.

2. Où l'on "tisse" des séquences (traitement itératif et récursif de listes chaînées)

Q2 Étude fonctionnelle [1,5 points]

On définit une fonction de *tissage* de deux séquences d'entiers, nommée **Tis** :

Tis : fonction (X, Y : séquence d'entier) → séquence de couple d'entier

{Posons X = [x_1, x_2, …, x_n] et Y = [y_1, y_2, …, y_m] et soit k le minimum de n et m.
Tis(X, Y) =[<x_1, y_1>, <x_2, y_2>, …, <x_k, y_k>]. Si X ou Y est vide, Tis(X,Y) = [].}

Exemple : si X = [3, 9, 6] et Y = [4, 8, 7, 5], alors Tis(X,Y) = [<3,4>,<9,8>,<6,7>]

— Donner des **équations de récurrence** définissant la fonction **Tis**.

Q3 Représentation chaînée [6,5 points]

Dans ce qui suit, les séquences sont représentées par des listes chaînées sous forme standard :

{listes chaînées d'entiers}
adCel_E : type pointeur de Cel_E
Cel_E : type <E : entier, Suc : adCel_E>
{listes chaînées de couples d'entiers}
adCel_C : type pointeur de Cel_C
Cel_C : type <E1, E2 : entier ; CSuc : adCel_C>

On spécifie une action nommée **Tisser** :

Tisser : action (donnée TX, TY : adCel_E ; résultat TZ : adCel_C)
{Soient X et Y les séquences d'entiers représentées par les listes d'adresses de tête TX et TY. À l'état final, la liste d'adresse de tête TZ représente la séquence de couples d'entiers de valeur Tis(X, Y).}

(i) Réalisation récursive de Tisser

— Donner une **réalisation récursive** de l'action **Tisser** :

(ii) Réalisation itérative de Tisser

— Donner une **réalisation itérative** de l'action **Tisser**.

(iii) Utilisation de l'action Tisser

Étant donnée une séquence **S** on veut construire la séquence des couples consécutifs de **S**. Par exemple, pour **S = [s_1, s_2, s_3, s_4]**, la séquence construite est **[<s_1, s_2>, <s_2, s_3>, <s_3, s_4>]**. Le résultat est la séquence vide si **S** a moins de deux éléments.

On étudie la construction d'une telle séquence de couples dans le cas d'une représentation chaînée. Pour cela, on spécifie l'action suivante :

CréerCC : action (donnée T : adCel_E ; résultat TC : adCel_C)
{Soit S la séquence représentée par la liste d'adresse de tête T. À l'état final, la liste d'adresse de tête TC représente la séquence des couples consécutifs de S.}

— Donner une **réalisation simple** de l'action **CréerCC**, fondée sur un seul appel de l'action **Tisser**.

(iv) Suppression des éléments de rang pair

On veut maintenant, étant donnée une séquence **[s_1, s_2, s_3, s_4, s_5, s_6]**, supprimer les éléments de rangs pairs de cette séquence pour obtenir **[s_1, s_3, s_5].**

On spécifie l'action suivante :

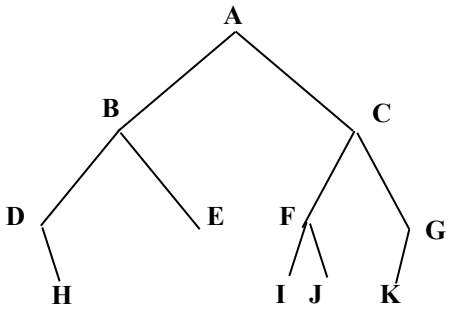
SupprUnSurDeux : action (donnée-résultat T : adCel_C)
*{**Modifie** la séquence de couples représentée par la liste d'adresse de tête T, en supprimant tous les couples de rang pair. Les cellules non utilisées sont libérées.}*

— Donner, **au choix, une réalisation récursive ou une réalisation itérative** (pas les deux) de l'action **SupprUnSurDeux**.

3. A propos de chemins dans un arbre binaire

On appelle *position* d'un nœud dans un arbre binaire, **la séquence de bits** (entiers valant 0 ou 1) décrivant le chemin d'accès à ce nœud (chemin de la racine au nœud) : à chaque arc de ce chemin correspond un bit, 0 pour un arc reliant un nœud à son fils gauche et 1 pour un arc reliant un nœud à son fils droit. La position de la racine est la séquence vide.

La figure ci-dessous illustre ce principe (les lettres A, B,… dénotent des étiquettes de nœuds) :



A →	**[ ]**	**B →**	**[0]**
C →	**[1]**	**D →**	**[0, 0]**
E →	**[0, 1]**	**F →**	**[1, 0]**
G →	**[1, 1]**	**H →**	**[0, 0, 1]**
I →	**[1, 0, 0]**	**J →**	**[1, 0, 1]**
K →	**[1, 1, 0]**		

Positions des nœuds

On fixe le lexique suivant :
Bit : type entier sur [0, 1]
Position : type séquence de Bit

Q4 [2 points] Un nœud fait-il partie de la descendance stricte d'un autre ?

On considère deux nœuds **N1** et **N2** d'un arbre binaire **A**, de positions respectives **P1** et **P2**.
On spécifie la fonction suivante :

EstDesc : fonction (P1, P2 : Position) → booléen
{vrai si et seulement si le nœud de position P2 dans un arbre binaire fait partie de la descendance stricte du nœud de position P1 dans le même arbre. P1 ne fait pas partie de sa propre descendance stricte.}

Exemple : dans l'arbre ci-dessus, I fait partie de la descendance de C donc :
EstDesc([1],[1,0,0]) = vrai

— Donner des **équations de récurrence** définissant la fonction nommée **EstDesc**.

Q5 [4 points] Arbres d'étoiles

Pour caractériser la structure d'un arbre, il suffit de donner l'ensemble des positions de ses feuilles (dans un ordre quelconque). On considère des arbres binaires **dont tous les nœuds sont étiquetés par le caractère étoile** ('*'). Étant donné l'ensemble des positions des feuilles d'un tel arbre, on veut construire sa représentation chaînée.

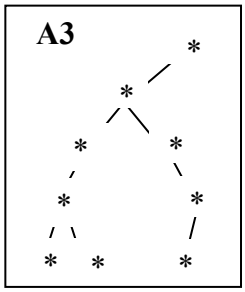
(i) Etude fonctionnelle récursive

On complète le lexique précédent et on spécifie les deux fonctions suivantes :
Etoile : caractère '*'

ArbreDétoiles : fonction (SP : séquence de Position) → arbre binaire d'étoile
{Arbre défini par la séquence SP de positions de ses feuilles (l'arbre vide si SP est vide).}

Plus : fonction (P : Position, A : arbre binaire d'étoile) → arbre binaire d'étoile
{Arbre obtenu à partir de A, en le complétant par les nœuds du chemin décrit par P et qui ne sont pas déjà dans A. Les nœuds sont tous étiquetés par une étoile.}

L'arbre **A3** dans la figure suivante est l'arbre obtenu par les appels successifs suivants :
soit A1 = Plus([0, 0, 0, 0], /\) dans
soit A2 = Plus([0, 1, 1, 0], A1) dans
Plus([0, 0, 0, 1], A2)



— Donner des **équations de récurrence** définissant **ArbreDétoiles** en utilisant **Plus**.

— Compléter les **équations de récurrence** suivantes, définissant la fonction **Plus** :
(1) Plus([], / \) = //Etoile\\ *{[] ↔ position de la racine : construction de l'arbre singleton}*
(2) Plus([], /G, Etoile, D\) = /G, Etoile, D\
(3) Plus(b_oP, / \) = • • • • • • •
(4) Plus(b_oP, /G, Etoile, D\) = • • • • • • •

(ii) Création de la représentation chaînée

Les arbres sont représentés sous forme chaînée standard.
Nœud : type <R : caractère, G, D : adNœud>
adNœud : type pointeur de Nœud

On spécifie l'action suivante :
CompléterArbre : action (donnée P : Position, donnée-résultat adR : adNœud)
{Modifie l'arbre d'adresse adR, en le complétant par les nœuds du chemin décrit par P (relativement à cet arbre) qui n'y sont pas déjà.
A l'état initial, adR est l'adresse d'un arbre de valeur A ; à l'état final, adR est l'adresse de l'arbre de valeur Plus(P, A).}

Pour les positions, on fera abstraction de la représentation des séquences. Pour les manipuler, on utilisera les opérateurs génériques définis sur le type séquence (EstVide?, premier, fin, …).

— Donner une **réalisation récursive** de l'action **CompléterArbre**.

4. À propos d'arbres n-aires (représentation chaînée)

On traite des arbres n-aires d'entiers représentés sous forme chaînée standard.
On utilise le lexique suivant :
adNœud : type pointeur de Nœud
Nœud : type <R : entier, Fils, Frère : adNœud>

Q6 [2 points] Valeur maximum dans un arbre n-aire d'entiers

On spécifie la fonction suivante :
MaxArbre : fonction (A : adNœud) → entiers
{Valeur maximum des éléments de l'arbre donné par l'adresse A de sa racine.
Pré-condition : l'arbre donné n'est pas vide, i.e. A ≠ Nil}

— Donner une **réalisation** de la fonction **MaxArbre**. On dispose d'une fonction nommée **Max2V** qui détermine le maximum de deux entiers donnés, dont on ne demande pas la réalisation.

M2 CCI - Algorithmique

Des exemples de solutions **25 février 2025**

1. Compactage d'une séquence de bits

Q1

(i) Décompactage [1,5 points]

{ parcours de la séquence de longueur donnée dans FC}

```
BC : Bit                                  {bit courant}
BC ← FC1
L ← 0
pour i allant de 2 à LC
    répéter FCi fois
        L ← L + 1
        SBL ← BC      {ajout en queue}
    BC ← 1-BC
```

(ii) Compactage, version 1 (par couples d'éléments consécutifs) [2,5 points]

```
FC1 ← SB1
LC ← 2
FCLC ← 1
pour i allant de 2 à L
    si SBi = SBi-1 alors FCLC ← FCLC + 1
    sinon LC ← LC + 1
          FCLC ← 1
```

Compactage, version 2 (SB est découpé en groupes de bits de même valeur)

```
i : entier sur [1…Lmax+1]                 {pour le parcours de SB}
nb : entier sur [1…Lmax]                  {nombre de bits de même valeur}
FC1 ← SB1
LC ← 1
i ← 1
tant que i ≠ L+1
        {i est la position dans SB du premier élément d'un groupe de bits de même valeur}
   nb ← 1
   i ← i +1
   tant que i ≠ L+1 et puis SBi = SBi-1
      nb ← nb + 1
      i ← i+1
   LC ← LC+1
   FCLC ← nb     {ajout en queue}
```

2. Où l'on tisse des séquences

Q2 Étude fonctionnelle [1,5 points]

(1) Tis([ ], [ ]) = [ ]
(2) Tis([ ], $e_o S$) = [ ]
(3) Tis($e_o S$, [ ]) = [ ]
(4) Tis($e1_o S1$, $e2_o S2$) = <e1, e2> $_o$ Tis (S1, S2)

Q3

(i) [1,5 points]

version 1

```
Tisser(TX, TY, TZ) :
    si TX = Nil ou TY = Nil alors TZ ← Nil
    sinon
        Allouer(TZ)
        TZ↑.E1 ← TX↑.E
        TZ↑.E2 ← TY↑.E
        Tisser(TX↑.Suc, TY↑.Suc, TZ↑.CSuc)
```

version 2

```
Tisser(TX, TY, TZ) :
    N : adCelC
    si TX = Nil ou TY = Nil alors TZ ← Nil
    sinon
        Tisser(TX↑.Suc, TY↑.Suc, TZ)
        Allouer(N)
        N↑ ← <TX↑.E, TY↑.E, TZ>
        TZ ← N
```

(ii) [2,5 points]

```
Tisser(TX, TY, TZ) :
    AC1, AC2 : adCel_E          {adresses courantes pour le parcours}
    Q : adCel_C                 {adresse de queue du résultat}
    N : adCel_C                 {pour création d'une nouvelle cellule}
    si TX = Nil ou TY = Nil alors TZ ← Nil
    sinon
        Allouer(TZ)
        TZ↑ ← <TX↑.E, TY↑.E, Nil>
        Q ← TZ
        AC1 ← TX↑.Suc
        AC2 ← TY↑.Suc
        tant que AC1 ≠ Nil et AC2 ≠ Nil
                Allouer(N)
                N↑ ← <AC1↑.E, AC2↑.E, Nil>
                Q↑.CSuc ← N
                Q ← N
                AC1 ← AC1↑.Suc
                AC2 ← AC2↑.Suc
```

(iii) [1 point]

CréerCC(T, TC) : si T=Nil alors TC ← Nil sinon Tisser(T, T↑.Suc, TC)

(iv) [1,5 points]

```
SupprUnSurDeux(T) :                 {Version itérative}
    AC, X : adCel_C
    AC ← T
    tant que AC ≠ Nil
            si AC↑.Suc ≠ Nil alors
                    X ← AC↑.Suc ; AC↑.Suc ← X↑.Suc ; Libérer(X)
            AC ← AC↑.Suc
```

```
SupprUnSurDeux(T) :                                   {Version récursive}
    X : adCel_C
    si T ≠ Nil alors
        si T↑.Suc ≠ Nil alors
            X ← T↑.Suc
            T↑.Suc ← X↑.Suc
            Libérer(X)
            SupprUnSurDeux(T↑.Suc)
```

3. À propos de chemins dans un arbre binaire

Q4 [2 points]

N2 fait partie de la descendance de **N1** si et seulement si **N1** se trouve sur le chemin d'accès à **N2** et par conséquent, si et seulement si **P1** est un préfixe strict de **P2.**

(1) EstDesc([], []) = faux
(2) EstDesc(b_oP, []) = faux *{on peut regrouper (1) et (2)}*
(3) EstDesc([], b_oP) = vrai
(4) EstDesc($b1_o$P1, $b2_o$P2) = (b1=b2) et puis EstDesc(P1, P2)

Q5

(i)

[1 point]

(1) ArbreDétoiles([]) = /\
(2) ArbreDétoiles(P_oSP) = Plus(P, ArbreDétoiles(SP))

[1 point]

(1) Plus([], /\) = //Etoile\\
(2) Plus([], /G, Etoile, D\) = /G, Etoile, D\
(3) Plus(b_oP, /\) = si b = 0 alors
/Plus(P, /\ ), Etoile, /\ \
sinon
/ /\, Etoile, Plus(P, /\ )\\
(4) Plus(b_oP, /G,Etoile,D\) = si b = 0 alors
/Plus(P, G), Etoile, D\
sinon
/G, Etoile, Plus(P, D)\

(ii) [2 points]

```
CompléterArbre(P, adR) : {Version 1 : traduction directe des 4 équations}
   si adR = Nil alors
        Allouer(adR)
        adR.G↑ ← Nil ; adR.R↑ ← Etoile ; adR.D↑ ← Nil
        si non EstVide?(P) alors
            si premier(P) = 0 alors
               CompléterArbre(fin(P), adR↑.G)
            sinon
               CompléterArbre(fin(P), adR↑.D)
   sinon si non EstVide?(P) alors
            si premier(P) = 0 alors
               CompléterArbre(fin(P), adR↑.G)
            sinon
               CompléterArbre(fin(P), adR↑.D)
```

```
CompléterArbre(P, adR) : {Version 2 : simplification}
    si adR = Nil alors
        Allouer(adR)
        adR.G↑ ← Nil ; adR.R↑ ← Etoile ; adR.D↑ ← Nil
    si non EstVide?(P) alors
        si premier(P) = 0 alors
            CompléterArbre(fin(P), adR↑.G)
        sinon
            CompléterArbre(fin(P), adR↑.D)
```

4. À propos d'arbres n-aires

Q6 [2 points]

```
MaxArbre(A) : {A≠Nil}
        M : entier
        AC : adNœud
        M ← A↑.R
        AC ← A↑.Fils
        tant que AC ≠ Nil
            M ← Max2v(M, MaxArbre(AC))
            AC ← AC↑.Frère
    retour : M
```

Version avec une fonction MaxForêt pour une forêt non vide :

```
MaxArbre(A) :
        retour : Max2V(A↑.R, MaxForêt (A↑.Fils))

MaxForêt (F) :
        retour :
            si A↑.Fils = Nil alors
                A↑.R
            sinon
                soit M = Max2V(A↑.R, MaxForêt (A↑.Fils)) dans
                    si A↑.Frère ≠Nil alors
                        Max2V(M, MaxForêt (A↑.Frère)
                    sinon
                        M
```