Nom: Prenom: Groupe: Numero:

Université de Montpellier

IUT département informatique

Date : 2019-2020
Épreuve : Algorithmique avancée
Durée : 1 h30 (2h avec tiers temps)
Autorisations : Pas de note, pas de machine

Consignes :
- commencez par mettre, en haut de *toutes* les pages du sujet et EN LETTRES MAJUSCULES D'IMPRIMERIE, votre nom, prénom, groupe et numéro (le numéro sur la feuille qui va circuler dans la salle pendant l'épreuve)
- répondez directement sur le sujet
- il est interdit d'utiliser des boucles
- par contre, vous pouvez utiliser les méthodes des questions précédentes pour répondre à la question courante

Chaque feuille sera corrigée par un enseignant différent. Si vous n'avez pas de place pour finir un exercice, ne continuez donc pas sur une autre feuille du sujet, mais demandez plutôt une feuille supplémentaire ! Le barême est indicatif (et sur 20).

Nom: Prenom: Groupe: Numero:

Exercice sur les tableaux

Le but de l'exercice est d'écrire une méthode permettant de vérifier si une matrice carrée de $n \times n$ entiers est symétrique. On rappelle qu'une matrice est symétrique si chacune des lignes i est égale à la colonne i ($i \in \{0..(n-1)\}$). Exemple de matrice carrée symétrique :

45	55	87	95
55	47	78	19
87	78	66	34
95	19	34	51

Une ligne i et une colonne i

Ecrire une méthode statique (non récursive) `egalLigCol` qui utilise une autre fonction auxiliaire (récursive elle) à **spécifier** et à écrire :

Nom: Prenom: Groupe: Numero:

Question 1 (4 points)
/* Pré-requis : mat est carrée et $0 \leq i < mat.length$
Résultat : retourne vrai ssi les i premiers entiers de la ligne i sont les mêmes que les i premiers entiers de la colonne i (ou de façon équivalente si la sous ligne $mat[i][0..mat[i][i-1]$ est égale à la sous colonne $mat[0][i..mat[i-1][i]$)
*/

```
public static boolean egalLigCol(int[][] mat, int i) {
```

Ecrire ici votre fonction auxiliaire récursive (et sa spécification) :

Nom: Prenom: Groupe: Numero:

Test de la symétrie

Ecrire les méthodes suivantes. La méthode `estSymetriqueAux` sera récursive, et elle pourra faire appel à `egalLigCol`.

Question 2 (2 points)
/* Pré-requis : mat est carrée et $0 \leq i < mat.length$
Résultat : retourne vrai ssi la sous-matrice `mat[0..i,0..i]` est symétrique
*/
```
public static boolean estSymetriqueAux (int[][] mat, int i) {
```

Question 3 (1 point)
/* Pré-requis : mat est carrée
Résultat : retourne vrai ssi la matrice `mat` est symétrique
*/
```
public static boolean estSymetrique (int[][] mat) {
```

Problème de la sous somme

On considère la classe Liste suivante vue en cours.

```
public class Liste {

    private int val;
    private Liste suiv;

    /* Constructeur d'une liste vide */
    public Liste () { this.suiv = null; }

    public booleam estVide() {return this.suiv == null;}

}
```

Etant donné une Liste l et un entier x, on s'intéresse au problème de la sous somme consistant à déterminer si l'on peut atteindre exactement x en sommant un sous ensemble des entiers de l. Par exemple, pour $l = [-1, 3, 5, 9]$ et $x_1 = 8$ la réponse est oui (et il y a même deux solutions, utiliser les entiers $\{-1, 9\}$ ou $\{3, 5\}$), et pour la même liste et $x_2 = 10$ la réponse est non. On considérera que ne prendre aucun élément donne la somme de 0.

Ecrire (dans la classe Liste) la méthode récursive existeUne.

Question 4 (3 points)
/* Pré-requis : aucun
Résultat : si $(this, x)$ n'a pas de solution alors retourne false, autrement retourne true. Par exemple pour l ci-dessus, $l.existeUne(8)$ doit retourner true, et $l.existeUne(10)$ doit retourner false.
Indication : on ne sait pas a priori si il faut utiliser l'élément en tête de this, donc essayer les deux solutions ! */

```
public boolean existeUne (int x) {
```

Pour modéliser une solution pour une liste l de n entiers, on va utiliser une liste de n booléens, le booléen en i-ème position indiquant si il faut prendre l'élément i de la liste. Pour l'exemple (l, x_1), les solutions sont donc représentées par $[t, f, f, t]$ et $[f, t, t, f]$.

Pour modéliser une solution, on utilisera la classe suivante. Remarquez que pour éviter d'ajouter des méthodes d'accès type getteurs setteurs, les attributs de *ListeBool* sont *public*, et donc dans tout le sujet on s'autorisera à accéder aux attributs b et *suiv* d'un objet *ListeBool*.

```
public class ListeBool {

    public boolean b;
    public ListeBool suiv;

    /* Constructeur d'une liste vide */
    public ListeBool () { this.suiv = null; }

    public booleam estVide() {return this.suiv == null;}


}
```

Ecrire (dans la classe Liste) la méthode récursive verif.

Question 5 (2 points)
/* Pré-requis : *this* et *sol* ont même longueur
Résultat : retourne vrai si et seulement si *sol* est bien une solution pour $(this, x)$.
Par exemple pour l ci-dessus, et $sol = [t, f, f, t]$, $l.verif(sol, 8)$ doit retourner vrai, et $l.verif(sol, 5)$ doit retourner faux.
*/

```
public boolean verif (ListeBool sol, int x) {
```

Nous allons maintenant nous intéresser au problème de retourner toutes les solutions. Pour ce faire, nous allons utiliser la notion d'*Arbre de décision*.

Définition.
Etant donné un entier $n \geq 0$, un *Arbre de décision de hauteur n* est un arbre binaire non vide dans lequel tous les chemins de la racine vers les feuilles empruntent exactement n arêtes.

On utilisera la convention que le côté gauche signifie "prendre l'élément", et le côté droit signifie "ne pas le prendre". Prenons pour exemple la liste $[2, 1, 1, 3]$ et $x = 3$, pour lequel il y a trois solutions $[t, t, f, f]$, $[t, f, t, f]$ et $[f, f, f, t]$. Ces trois solutions sont représentées dans l'arbre de gauche de la Figure 1, qui est un arbre de décision de hauteur 4. Notez que l'arbre de droite n'est pas un arbre de décision car le nombre d'arêtes d'un chemin de la racine à une feuille n'est pas toujours le même (3 ou 4).

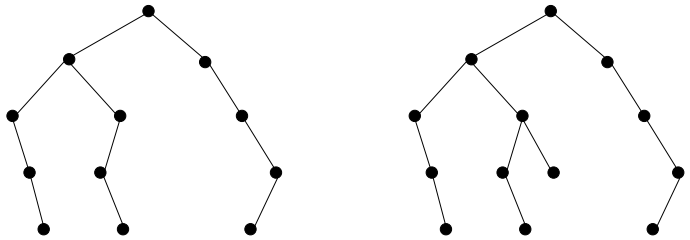

FIGURE 1 –

Pour modéliser des arbres de décision on va utiliser la classe *Arbre* ci-dessous. Notez que à nouveau on utilise des attributs *public*, et qu'il n'y a pas d'attribut *val* puisque l'on ne souhaite pas stocker d'entiers dans l'arbre.

```
class Arbre{
    public Arbre filsG;//filsG non vide : on prend l'élément
    public Arbre filsD;//filsD non vide : on ne prend pas l'élément

    public Arbre(){
        filsG=null;
        filsD=null;
    }
    public boolean estVide(){ return filsG==null;}
}
```

On adoptera la même convention qu'en cours, à savoir que soit $filsG == filsD = null$ (pour l'arbre vide), soit $filsG != null$ et $filsD != null$. Vous pouvez observer dans la Figure 2 la représentation mémoire d'un arbre de décision de hauteur 1 (à gauche), et d'un arbre de décision de hauteur 0 (à droite). Notez que l'arbre vide n'est *pas* un arbre de décision.

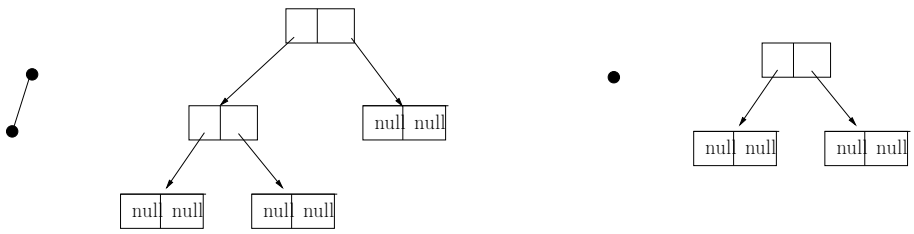

FIGURE 2 –

Ecrire (dans la classe Arbre) la méthode arbreDeHauteurNulle.

Question 6 (1 point)

```
/* Pré-requis : aucun
Résultat : retourne un arbre de décision de hauteur 0
*/
public Arbre arbreDeHauteurNulle () {
```

On va d'abord s'intéresser au problème de reconnaître un arbre de décision de hauteur n.

Question 7 (1 point)
Une récurrence un peu trop simpliste serait de dire (pour n grand) que $this$ est un arbre de décision de hauteur n si et seulement si $this.filsG$ et $this.filsD$ sont des arbres de décision de hauteur $n-1$. Dessinez (avec des points et des traits comme dans la Figure 1) un contre-exemple à cette équivalence.

Ecrire (dans la classe Arbre) la méthode récursive estArbreDec.

Question 8 (3 points)
/* Pré-requis : $n \geq 0$
Résultat : retourne vrai si et seulement si *this* est un arbre de décision de hauteur n (on rappelle qu'un arbre vide n'est pas un arbre de décision)
*/

```
public boolean estArbreDec (int n) {
```

On considère maintenant le problème de, étant donné une liste d'entiers l et un entier cible x, construire toutes les solutions sous forme d'un arbre de décision.

Ecrire (dans la classe Liste) le code de trouveToutes.

Question 9 (3 points)
/* Pré-requis : aucun
Résultat :
si $(this, x)$ n'a pas de solutions alors retourne un arbre vide, sinon retourne un arbre de décision de hauteur longueur(this) encodant toutes les solutions de $(this, x)$
(Attention : si this est vide, distinguer le cas où il n'y a pas de solution (retourne l'arbre vide) de celui où il y a une solution qui consiste à ne rien prendre (retourne l'arbre de décision de hauteur 0).
*/

```
public Arbre trouveToutes (int x)
```