Correction TD2 : Listes, tableaux, tuples

1 Tableaux

1. *Écrire un algorithme* `minmax` *qui prend en argument un tableau* `tab`*, et renvoie un couple* `(x,y)` *tel que* x *est l'élément maximal de tab, et* y *l'élément minimal.*

   On stocke les valeurs minimales et maximales rencontrées au cours du parcours des éléments de `tab` dans les variables `mini` et `maxi`.

Algorithme 1 : minmax

```
Entrées : tab : tableau
n ← taille(tab)
mini ← tab[0]
maxi ← tab[0]
pour i de 1 à n − 1 faire
    si tab[i]<mini alors
        mini ← tab[i]
    sinon si tab[i]>maxi alors
        maxi ← tab[i]
retourner (mini,maxi)
```

2. *Écrire un algorithme* `unimodal` *qui prend en argument un tableau* `tab`*, et qui détermine si le tableau représente une séquence unimodale (d'abord croissante puis décroissante).*

   Une première solution consiste à déterminer le premier indice i à partir duquel `tab` est décroissant, puis vérifier qu'il reste décroissant à partir de cet indice.

Algorithme 2 : unimodal

```
Entrées : tab : tableau
n ← taille(tab)
i ← 0
tant que i ≤ n − 2 et tab[i] ≤ tab[i+1] faire
    i ← i + 1
pour j de i à n − 2 faire
    si tab[j] < tab[j+1] alors
        retourner Faux
retourner Vrai
```

   Une seconde solution consiste à vérifier que le tableau ne contient pas d'élément plus petit que le précédent ET le suivant, ce qui est une condition nécessaire et suffisante pour être unimodal.

Algorithme 3 : unimodal

```
Entrées : tab : tableau
n ← taille(tab)
pour i de 1 à n − 2 faire
    si tab[i] < tab[i-1] et tab[i] < tab[i+1] alors
        retourner Faux
retourner Vrai
```

3. *Écrire un algorithme* `occurrences` *qui prend en argument un tableau* `tab` *et une valeur* `x`*, et qui renvoie la liste de toutes les positions* `i` *telles que* `tab[i]=x`*.*

   On initialise une liste vide `res`, à laquelle on ajoute chacune des positions de `x` dans le tableau quand on en rencontre une.

Algorithme 4 : occurrences

Entrées : `tab` : tableau, `x`
$n \leftarrow$ `taille(tab)`
`res` $\leftarrow$ `[]`
pour i *de* 0 *à* $n-1$ **faire**
  si `tab[i]=x` **alors**
    Ajouter `i` à `res`
retourner `res`

4. *Écrire un algorithme* `absents` *qui prend en argument un tableau* `tab` *et un entier* `n`*, et qui renvoie la liste de tous les entiers inférieurs à* n *qui n'apparaissent pas dans* `tab`*.*

   On construit un tableau de booléens `present` qui détermine la présence ou l'absence de chaque entier $< n$ dans `tab`. Pour cela, chaque fois qu'on rencontre un entier $< n$ dans `tab`, on passe la case correspondante dans `present` à `Vrai`. Le résultat souhaité est alors l'ensemble des positions qui contiennent `Faux` dans `present`, ce que l'on obtient en faisant appel à `occurrences(Faux,present)`.

Algorithme 5 : absents

Entrées : `tab` : tableau d'entiers, n : entier
`present` $\leftarrow$ `n*[Faux]`
pour x *dans* `tab` **faire**
  si $x < n$ **alors**
    `present[x]` $\leftarrow$ `Vrai`
retourner `occurrences(Faux,present)`

2 Carrés magiques

Un carré magique $n \times n$ est une matrice de n^2 nombres telle que la somme des éléments d'une même ligne, d'une même colonne, ou d'une même diagonale donne toujours la même valeur. Un carré magique $n \times n$ est naturel si en plus il contient les nombres $1, \ldots, n^2$.

1. *Écrire un algorithme* `Magic` *qui prend en paramètre un tableau bidimensionnel, et qui renvoie* `Vrai` *si ce tableau représente un carré magique, et* `Faux` *sinon.*

   On utilise des fonctions auxiliaires pour calculer la somme d'une ligne, colonne, ou diagonale dans la matrice.

Algorithme 6 : SommeLigne

Entrées : `mat` : matrice `i` : entier
`res` $\leftarrow 0$
pour x *dans* `mat[i]` **faire**
  `res` $\leftarrow$ `res` $+ x$
retourner `res`

Algorithme 7 : SommeColonne

Entrées : `mat` : matrice `i` : entier
$n \leftarrow$ `taille(mat)`
`res` $\leftarrow 0$
pour j *de* 0 *à* $n-1$ **faire**
  `res` $\leftarrow$ `res+mat[j][i]`
retourner *res*

Algorithme 8 : SommeDiagonale

Entrées : `mat` : matrice, `dir` $\in \{-1, 1\}$
$n \leftarrow$ `taille(mat)`
`res` $\leftarrow 0$
si *dir>0* **alors**
  $i_0 \leftarrow 0$
sinon
  $i_0 \leftarrow n-1$
pour i *de* 0 *à* $n-1$ **faire**
  `res` $\leftarrow$ `res+mat[i][`i_0`+i*dir]`
retourner *res*

On calcule ensuite la valeur de la somme d'une des diagonales, et on vérifie si chacune des autres sommes est identique à cette valeur.

Algorithme 9 : Magic

Entrées : `mat` : matrice
$n \leftarrow$ `taille(mat)`
`value` $\leftarrow$ `SommeDiagonale(mat,1)`
si *SommeDiagonale(mat,-1)* $\neq$ *value* **alors**
  retourner *Faux*
pour i *de* 0 *à* $n-1$ **faire**
  si *SommeLigne(mat,i)* $\neq$ *value* ***ou*** *SommeColonne(mat,i)* $\neq$ *value* **alors**
    retourner *Faux*
retourner *Vrai*

2. *Écrire un algorithme* `Natural` *qui prend en paramètre un tableau bidimensionnel, et qui renvoie* `Vrai` *si ce tableau représente un carré magique naturel, et* `Faux` *sinon.*

   On vérifie que la matrice ne contient ni un doublon, ni une valeur en dehors de celles autorisées, ce qui suffit à vérifier que les valeurs présentes sont $\{1, \ldots, n^2\}$. Pour cela, on garde en mémoire chaque valeur rencontrée à l'aide d'un tableau de booléen afin de pouvoir tester les doublons en temps constant.

   Une fois cette condition vérifiée, il ne reste plus qu'à tester que l'on a bien un carré magique.

Algorithme 10 : Natural

```
Entrées : mat : matrice
n ← taille(mat)
present ← (n*n)*[Faux]
pour i de 0 à n − 1 faire
    pour j de 0 à n − 1 faire
        si mat[i][j] < 1 ou mat[i][j] > n * n alors
            retourner Faux                          //mat[i][j] sort de l'intervalle.
        si present[mat[i][j]-1] alors
            retourner Faux                          //mat[i][j] a déjà été rencontré.
        present[mat[i][j]-1] ← Vrai                 //On indique qu'on a rencontré mat[i][j].
retourner Magic(mat)
```

3 Bon parenthésage

1. *Écrire un algorithme* `BienParenthese` *qui vérifie si une expression donnée en paramètre sous la forme d'une chaîne de caractères* `s` *est bien parenthésée.*

   → Résolu avec la question 2.

2. *ÉMême question, mais cette fois-ci différent types de parenthèses peuvent apparaître dans l'expression :* `( ) [ ] { }`.

   On implémente une fonction qui associe à chaque symbole fermant son symbole ouvrant correspondant.

Algorithme 11 : Complement

```
Entrées : c : caractère
si c=')' alors
    retourner '('
si c=']' alors
    retourner '['
si c='}' alors
    retourner '{'
```

On empile tous les symboles ouvrants, puis à chaque symbole fermant rencontré on dépile le dernier symbole ouvrant rencontré et vérifie qu'il est compatible. Il faut également s'assurer que la pile est vide à la fin.

Algorithme 12 : BienParenthese

```
Entrées : s : string
pile ← []
pour c dans s faire
    si c='(' ou c='[' ou c='{' alors
        Ajouter c à pile
    sinon si c=')' ou c=']' ou c='}' alors
        si EstVide(pile) ou Extraire(pile) ≠ Complement(c) alors
            retourner Faux
retourner EstVide(pile)
```