



**Haute École Bruxelles-Brabant**  
**École Supérieure d'Informatique**  
Rue Royale, 67. 1000 Bruxelles  
02/219.15.46 – esi@he2b.be

# Algorithmique

2020

Bachelor en Informatique  
DEV2

A. Paquot (APA), M. Codutti (MCD), N. Richard (NRI),  
S. Drobisz (SDR), S. Rexhep (SRE) & ???

# Table des matières

<b>1</b>	<b>Les tableaux à 2 dimensions</b>	<b>3</b>
1.1	Définition . . . . .	3
1.2	Notations . . . . .	3
1.3	Parcours . . . . .	7
1.4	Exercices . . . . .	15

# Les tableaux à 2 dimensions

## 1.1 Définition

La **dimension** d'un tableau est le nombre d'indices qu'on utilise pour faire référence à un de ses éléments.



### Attention

Ne pas confondre la dimension avec la taille !

En DEV<sub>1</sub>, nous avons introduit les tableaux à une dimension. Un seul indice suffisait à localiser un de ses éléments. Pour le dire autrement, chaque case possédait **un** numéro. De nombreuses situations nécessitent cependant l'usage de tableaux à deux dimensions. Ils vous sont déjà familiers par leur présence dans beaucoup de situations courantes : calendrier, grille horaire, grille de mots croisés, sudoku, jeux se déroulant sur un quadrillage (damier, échiquier, scrabble... ). Dans ces situations, chaque case est désignée par **deux** numéros.

## 1.2 Notations

### 1.2.1 Déclarer

Pour **déclarer** un tableau à 2 dimensions, on écrira :



```
nomTableau: array of nbLignes × nbColonnes TypeElément
```

où nbLignes et nbColonnes sont des expressions entières quelconques.

**Exemple :**

```
tab: array of 5×10 integers
```

déclare un tableau de 5 lignes par 10 colonnes dont chaque case contient un entier.

### 1.2.2 Utiliser

Pour **accéder** à une case du tableau on donnera les deux indices entre crochets. Comme en DEV<sub>1</sub>, on considère que la première ligne et la première colonne portent le numéro (l'indice) 0.



**Exemple :**

```
print tab[2,4] // affiche le 5° élément de la 3° ligne du tableau nommé tab.
```

### 1.2.3 Visualiser

Notez que la vue sous forme de tableau avec des lignes et des colonnes est une vision humaine. Il n'y a pas de lignes ni de colonnes en mémoire.

**Exemple :** Soit le tableau déclaré ainsi :

nombres: **array of 4×5 integers**

On peut le visualiser à l'aide d'une grille à 4 lignes et 5 colonnes.

	0	1	2	3	4
0	0	1	2	3	4
1	10	11	12	13	14
2	20	21	22	23	24
3	30	31	32	33	34

Ainsi, la valeur de `nombres[2,3]` est 23.

Mais on pourrait considérer que le 1<sup>er</sup> indice désigne la colonne et le second indice la ligne. On le visualiserait alors ainsi :

	0	1	2	3
0	0	10	20	30
1	1	11	21	31
2	2	12	22	32
3	3	13	23	33
4	4	14	24	34

Cela ne change pas le fait que la valeur de `nombres[2,3]` est toujours 23.

On pourrait aussi visualiser un tableau à deux dimensions comme un tableau à une dimension dont chacun des éléments est lui-même un tableau à une dimension. La vision « tableau de tableaux » (ou décomposition en niveaux) donnerait :

0	1	2	3
0 1 2 3 4 0 1 2 3 4	0 1 2 3 4 10 11 12 13 14	0 1 2 3 4 20 21 22 23 24	0 1 2 3 4 30 31 32 33 34

Dans cette représentation, le tableau `nombres` est d'abord décomposé à un premier niveau en quatre éléments auxquels on accède par le premier indice. Ensuite, chaque élément de premier niveau est décomposé en cinq éléments de deuxième niveau accessibles par le deuxième indice.

#### Convention

Pour être précis, on devrait juste parler de première dimension et de deuxième dimension mais la notion de ligne et de colonne est un abus de langage qui simplifie le discours.

### 1.2.4 Exemples

**Exemple 1 – Remplir les coins.** Dans ce petit exemple, on a un tableau de chaînes et on donne des valeurs aux coins.

"NO"				"NE"
"SO"				"SE"

```
// Déclare un tableau et donne des valeurs aux coins.
```

```
algorithm remplirCoins
```

```
  grille: array of 3×5 string
```

```
  grille[0,0] = "NO"
```

```
  grille[0,4] = "NE"
```

```
  grille[2,0] = "SO"
```

```
  grille[2,4] = "SE"
```

```
end
```

La version JAVA :

```
1 public static void remplirCoins() {
2     String[][] grille = new String[3][5];
3     grille[0][0] = "NO";
4     grille[0][4] = "NE";
5     grille[2][0] = "SO";
6     grille[2][4] = "SE";
7 } // Pour afficher : System.out.println(Arrays.deepToString(grille));
```

### Syntaxe Java

On vous renvoie à votre cours de JAVA pour les précisions sur la syntaxe JAVA.

Notez juste ici :

- ▷ Comment est déclaré/créé le tableau.
- ▷ Le double [] pour accéder à un élément.

**Exemple 2 – Gestion des stocks.** Reprenons l'exemple du stock de 10 produits qui a servi d'introduction au chapitre sur les tableaux mais, cette fois, pour chaque jour de la semaine.

	article0	article1	article2	...	article7	article8	article9
lundi	cpt[0,0]	cpt[0,1]	cpt[0,2]	...	cpt[0,7]	cpt[0,8]	cpt[0,9]
mardi	cpt[1,0]	cpt[1,1]	cpt[1,2]	...	cpt[1,7]	cpt[1,8]	cpt[1,9]
mercredi	cpt[2,0]	cpt[2,1]	cpt[2,2]	...	cpt[2,7]	cpt[2,8]	cpt[2,9]
jeudi	cpt[3,0]	cpt[3,1]	cpt[3,2]	...	cpt[3,7]	cpt[3,8]	cpt[3,9]
vendredi	cpt[4,0]	cpt[4,1]	cpt[4,2]	...	cpt[4,7]	cpt[4,8]	cpt[4,9]
samedi	cpt[5,0]	cpt[5,1]	cpt[5,2]	...	cpt[5,7]	cpt[5,8]	cpt[5,9]
dimanche	cpt[6,0]	cpt[6,1]	cpt[6,2]	...	cpt[6,7]	cpt[6,8]	cpt[6,9]

```
// Calcule et affiche la quantité vendue de 10 produits
```

```
// pour chaque jour de la semaine (de 0 : lundi à 6 : dimanche).
```

```
algorithm statistiquesVentesSemaine
```

```
  cpt: array of 7×10 integers
```

```
  initialiser(cpt)
```

```
  for jour from 0 to 6
```

```
    // Pour chaque jour de la semaine
```

```
    print "jour : ", jour traiterStock1Jour(cpt, jour)
```

```
    for produit from 0 to 9
```

```
    | print "quantité vendue de produit ", produit, " ce jour ", jour, " : ", cpt[jour,produit]
```

```
    end
```

```
  end
```

```
end
```

```
// Initialise le tableau d'entiers à 0
```

```
algorithm initialiser(entiers: array of 7×10 integers)
```

```
  for i from 0 to 6
```

```
    for j from 0 to 9
```

```
    | entiers[i,j] = 0
```

```
    end
```

```
  end
```

```
end
```

```
// Effectue le traitement du stock pour une journée.
algorithm traiterStock1Jour(cpt: array of 7×10 integers, jour: integer)
    numéroProduit, quantité: integers
    numéroProduit = ask "Introduisez le numéro du produit :"
    while numéroProduit ≥ 0 et numéroProduit < 10
        quantité = ask "Introduisez la quantité vendue :"
        cpt[jour,numéroProduit] = cpt[jour,numéroProduit] + quantité
        numéroProduit = ask "Introduisez le numéro du produit :"
    end
end
```

En JAVA :

```
1 public static void statistiquesVentesSemaine() {
2     int[][] cpt = new int[7][10];
3     initialiser(cpt);
4     for (int jour = 0; jour < 7; jour++) {
5         System.out.println("Jour : " + jour);
6         traiterStock1Jour(cpt, jour);
7         for (int produit = 0; produit < 10; produit++) {
8             System.out.println("quantité vendue du produit " + produit
9                 + " ce jour " + jour + " : " + cpt[jour][produit]);
10        }
11    }
12 }
```

```
1 private static void initialiser(int[][] cpt) {
2     // Rien à faire : En Java, les tableaux d'entiers sont initialisés à 0.
3 }
```

```
1 private static void traiterStock1Jour(int[][] cpt, int jour) {
2     int numéroProduit, quantité;
3     // askInt est une méthode utilitaire pour lire un entier de façon conviviale et robuste
4     numéroProduit = askInt("Introduisez le numéro du produit");
5     while (numéroProduit >= 0 && numéroProduit < 10) {
6         quantité = askInt("Introduisez la quantité vendue");
7         cpt[jour][numéroProduit] += quantité;
8         numéroProduit = askInt("Introduisez le numéro du produit");
9     }
10 }
```

### 1.2.5 Exercices

Ces exercices ne nécessitent pas encore de parcours du tableau (pas de boucle). Nous verrons les algorithmes de parcours juste après.

#### Exercice 1 Case nulle ?

Écrire un algorithme qui reçoit un tableau d'entiers (à  $n$  lignes et  $m$  colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et qui retourne un booléen indiquant si la case désignée contient ou pas la valeur nulle.

```
algorithm estNul(tab: array of  $n \times m$  integers, lg, col: integers) → boolean
```

#### Exercice 2 Case existe ?

Écrire un algorithme qui reçoit un tableau d'entiers (à  $n$  lignes et  $m$  colonnes) ainsi que des coordonnées et qui retourne un booléen indiquant si ces coordonnées désignent bien une case du tableau, c-à-d qu'elles sont dans le bon intervalle (0 à  $n - 1$  et 0 à  $m - 1$ ).

```
algorithm existe(tab: array of  $n \times m$  integers, lg, col: integers) → boolean
```

**Exercice 3****Assigner une case**

Écrire un algorithme qui reçoit un tableau d'entiers (à  $n$  lignes et  $m$  colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et une valeur entière. L'algorithme met la valeur donnée dans la case indiquée pour autant que la case contienne actuellement la valeur nulle. Dans le cas contraire, l'algorithme ne fait rien.

```
algorithm assigner(tab: array of  $n \times m$  integers, lg, col, val: integers)
```

**Exercice 4****Un bord du tableau**

Écrire un algorithme qui reçoit un tableau d'entiers (à  $n$  lignes et  $m$  colonnes) ainsi que les coordonnées d'une case (ligne, colonne). L'algorithme doit indiquer si la case donnée est ou non sur un **bord** du tableau.

```
algorithm estBord(tab: array of  $n \times m$  integers, lg, col: integers) → boolean
```

**Exercice 5****Un coin du tableau**

Écrire un algorithme qui reçoit un tableau d'entiers (à  $n$  lignes et  $m$  colonnes) ainsi que les coordonnées d'une case (ligne, colonne). L'algorithme doit indiquer si la case donnée est ou non sur un des 4 **coins** du tableau.

```
algorithm estCoin(tab: array of  $n \times m$  integers, lg, col: integers) → boolean
```

**1.3 Parcours**

Comme nous l'avons fait pour les tableaux à une dimension, envisageons le parcours des tableaux à deux dimensions ( $n$  lignes et  $m$  colonnes). Nos algorithmes sont valables quel que soit le type des éléments. Utilisons  $T$ <sup>1</sup> pour désigner un type quelconque.

```
tab: array of  $n \times m$  T
```

Commençons par des cas plus simples où on ne parcourt qu'une seule des dimensions puis attaquons le cas général.

**1.3.1 Parcours d'une dimension**

Certains parcours ne visitent qu'une partie du tableau et ne nécessitent qu'une seule boucle. Examinons quelques cas.

**Une ligne.** On peut vouloir ne parcourir qu'une seule ligne du tableau. Si on parcourt la ligne  $l$ , on visite les cases  $(l, 0)$ ,  $(l, 1)$ ,  $\dots$ ,  $(l, m - 1)$ . L'indice de ligne est constant et c'est l'indice de colonne qui varie.

$l$	1	2	3	4	5

1. Nos versions JAVA seront écrits avec des tableau d'entiers. Il est possible de les écrire avec un type *générique*  $T$  mais ça requiert des notions avancées de JAVA.

Ce qui donne l'algorithme :

```
// Parcours de la ligne lg d'un tableau à deux dimensions
algorithm afficherLigne(tab: array of  $n \times m$  T, lg: integer)
|    $\forall$  col : numéro de colonne // Pour tous les numéros de colonne (0 à  $m - 1$ )
|   | print tab[lg, col] // On peut faire autre chose qu'afficher
|   end
end
```

```
1 public static void afficherLigne(int[][] tab, int lg) {
2     for (int col = 0; col < tab[0].length; col++) {
3         System.out.println(tab[lg][col]);
4     }
5 }
```

### Retenons

Pour parcourir une ligne, on utilise une boucle sur les colonnes.

**Une colonne.** Symétriquement, on pourrait considérer le parcours d'une colonne avec l'algorithme suivant.

```
// Parcours de la colonne col d'un tableau à deux dimensions
algorithm afficherColonne(tab: array of  $n \times m$  T, col: integer)
|    $\forall$  lg : numéro de ligne // Pour tous les numéros de ligne
|   | print tab[lg, col] // On peut faire autre chose qu'afficher
|   end
end
```

```
1 public static void afficherColonne(int[][] tab, int col) {
2     for (int lg = 0; lg < tab.length; lg++) {
3         System.out.println(tab[lg][col]);
4     }
5 }
```

**La diagonale descendante.** Si le tableau est carré ( $n = m$ ) on peut aussi envisager le parcours des deux diagonales.

Pour la diagonale descendante, les éléments à visiter sont  $(0, 0)$ ,  $(1, 1)$ ,  $(2, 2)$ , ...,  $(n - 1, n - 1)$ .

1		
	2	
		3

Une seule boucle suffit comme le montre l'algorithme suivant.

```
// Parcours de la diagonale descendante d'un tableau carré
algorithm afficherDiagonaleDescendante(tab: array of  $n \times n$  T)
|   for i from 0 to  $n-1$ 
|   | print tab[i,i] // On peut faire autre chose qu'afficher
|   end
end
```

```
1 public static void afficherDiagonaleDescendante(int[][] tab) {
2     for (int i = 0; i < tab.length; i++) {
3         System.out.println(tab[i][i]);
4     }
5 }
```



**La diagonale montante.** Pour la diagonale montante, on peut envisager deux solutions, avec deux indices ou un seul en se basant sur le fait que  $lg + col = n - 1 \Rightarrow col = n - 1 - lg$ .

		1
	2	
3		

```
// Parcours de la diagonale montante d'un tableau carré - version 2 indices
// lg part du début et augmente / col part de la fin et diminue
algorithm afficherDiagonaleMontante(tab: array of  $n \times n$  T)
|   col: integer
|   col = n-1
|   for lg from 0 to n-1
|   |   print tab[ lg, col ]           // On peut faire autre chose qu'afficher
|   |   col = col - 1
|   end
end
```

```
// Parcours de la diagonale montante d'un tableau carré - version 1 indice
algorithm afficherDiagonaleMontante(tab: array of  $n \times n$  T)
|   for lg from 0 to n-1
|   |   print tab[ lg, n-1-lg ]       // On peut faire autre chose qu'afficher
|   end
end
```

```
1 // Version avec deux variables
2 public static void afficherDiagonaleMontanteV1(int[][] tab) {
3     int col = tab[0].length - 1;
4     for (int lg = 0; lg < tab.length; lg++) {
5         System.out.println(tab[lg][col]);
6         col--;
7     }
8 }
```

```
1 // En Java, on peut placer les 2 variables dans le for
2 public static void afficherDiagonaleMontanteV2(int[][] tab) {
3     for (int lg = 0, col = tab[0].length - 1; lg < tab.length; lg++, col--) {
4         System.out.println(tab[lg][col]);
5     }
6 }
```

```
1 // Version avec la colonne calculée à partir de la ligne
2 public static void afficherDiagonaleMontanteV3(int[][] tab) {
3     for (int lg = 0; lg < tab.length; lg++) {
4         System.out.println(tab[lg][tab.length - 1 - lg]);
5     }
6 }
```

### 1.3.2 Parcours des deux dimensions

Les parcours précédents ne concernaient qu'une partie du tableau. Voyons à présent comment visiter toutes les cases du tableau.

**Parcours par lignes et par colonnes.** Les deux parcours les plus courants sont les parcours ligne par ligne et colonne par colonne. Les tableaux suivants montrent dans quel ordre chaque case est visitée dans ces deux parcours.

Parcours ligne par ligne

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Parcours colonne par colonne

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

Le plus simple est d'utiliser deux boucles imbriquées

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne
algorithm afficherLigneParLigne(tab: array of  $n \times m$   $T$ )
   $\forall$  lg : numéro de ligne
  |  $\forall$  col : numéro de colonne
  |   print tab[lg,col]                                // On peut faire autre chose qu'afficher
  |   end
  | end
end
```

Il suffit d'inverser les deux boucles pour effectuer un parcours par colonne.

```
// Parcours d'un tableau à 2 dimensions, colonne par colonne
algorithm afficherColonneParColonne(tab: array of  $n \times m$   $T$ )
   $\forall$  col : numéro de colonne
  |  $\forall$  lg : numéro de ligne
  |   print tab[lg,col]                                // On peut faire autre chose qu'afficher
  |   end
  | end
end
```

```
1 public static void afficherLigneParLigne(int[][] tab) {
2     for (int lg = 0; lg < tab.length; lg++) {
3         for (int col = 0; col < tab[0].length; col++) {
4             System.out.println(tab[lg][col]);
5         }
6     }
7 }
```

```
1 public static void afficherColonneParColonne(int[][] tab) {
2     for (int col = 0; col < tab[0].length; col++) {
3         for (int lg = 0; lg < tab.length; lg++) {
4             System.out.println(tab[lg][col]);
5         }
6     }
7 }
```

Mais on peut obtenir le même résultat avec une seule boucle si l'indice sert juste à compter le nombre de passages ( $n \times m$ ) et que les indices de lignes et de colonnes sont gérés manuellement.

L'algorithme suivant montre ce que ça donne pour un parcours ligne par ligne. La solution pour un parcours colonne par colonne est similaire et laissée en exercice.

```
// Parcours d'un tableau à 2 dimensions via une seule boucle
algorithm afficherLigneParLigne(tab: array of  $n \times m$   $T$ )
  On commence à la 1re case, en haut à gauche.
  repeat  $n \times m$  times
  |   print tab[lg,col]                                // On peut faire autre chose qu'afficher
  |   if on n'est pas sur le bord droit
  |   |   Passer sur la case de droite
  |   else
  |   |   Passer à la case de gauche de la ligne suivante
  |   end
  | end
end
```

En plus détaillé :

```
// Parcours d'un tableau à 2 dimensions via une seule boucle
algorithm afficherLigneParLigne(tab: array of  $n \times m$  T)
  lg, col: integers
  lg = 0
  col = 0
  for i from 1 to n*m
    print tab[lg,col]                                // On peut faire autre chose qu'afficher
    col = col + 1                                    // Passer à la case suivante
    if col == m                                     // On déborde sur la droite, passer à la ligne suivante
      col = 0
      lg = lg + 1
    end
  end
end
```

Et en JAVA :

```
1  public static void afficherLigneParLigneV2(int[][] tab) {
2      int nbÉléments = tab.length * tab[0].length;
3      int lg = 0;
4      int col = 0;
5      for (int i = 0; i < nbÉléments; i++) {
6          System.out.println(tab[lg][col]);
7          col++;
8          if (col == tab[0].length) {
9              col = 0;
10             lg++;
11         }
12     }
13 }
```

L'avantage de cette solution apparaîtra quand on verra des situations plus difficiles.

### 1.3.3 Interrompre le parcours.

Comme avec les tableaux à une dimension, envisageons l'arrêt prématuré lors de la rencontre d'une certaine condition. Nous allons voir une version adaptée du parcours à deux boucles et une version adaptée du parcours à une boucle.

**Version avec 2 boucles.** Comme avec les tableaux à une dimension, transformons d'abord nos *pour* en *tant que*.

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne, via un tant que
algorithm afficherLigneParLigne(tab: array of  $n \times m$  T)
  lg, col: integers
  lg = 0
  while lg < n
    col = 0
    while col < m
      print tab[lg, col]                                // On peut faire autre chose qu'afficher
      col++
    end
    lg++
  end
end
```

On peut à présent introduire le test comme on l'a fait dans les algorithmes de parcours des tableaux à une dimension.

**Rappelons-nous**

Nous avons vu deux solutions pour un parcours interrompu.

- ▷ Une 1<sup>re</sup> version où le **test** est placé directement **dans la condition** de la boucle.
- ▷ Une seconde version utilisant un **booléen** indiquant si on continue ou pas. Ce booléen est **mis à jour dans la boucle**.

Illustrons-le au travers d'un algorithme qui cherche un élément particulier. et retourne un booléen indiquant si l'élément a été trouvé ou pas.

Avec deux boucles, la solution la **plus simple** est celle qui introduit un **booléen**.

```
// Parcours avec test d'arrêt - deux boucles et un booléen
algorithm chercherElément(tab: array of  $n \times m$  T, élt: T)  $\rightarrow$  boolean
| lg, col: integers
| trouvé: boolean
| trouvé = faux
| lg = 0
| while lg < n ET NON trouvé
| | col = 0
| | while col < m ET NON trouvé
| | | trouvé = tab[lg, col] == élt
| | | col++
| | end
| | lg++
| end
| return trouvé
end
```

S'il est important de connaître la position de l'élément trouvé, il faut veiller à ne pas incrémenter la ligne et/ou la colonne lorsque l'élément est trouvé. Ce qui donne :

```
// Parcours avec test d'arrêt - deux boucles et un booléen
algorithm chercherElément(tab: array of  $n \times m$  T, élt: T)  $\rightarrow$  boolean
| lg, col: integers
| trouvé: boolean
| trouvé = faux
| lg = 0
| while lg < n ET NON trouvé
| | col = 0
| | while col < m ET NON trouvé
| | | if tab[lg, col] == élt
| | | | trouvé = vrai
| | | else // Ne pas modifier les indices si arrêt demandé
| | | | col++
| | | end
| | end
| | if NON trouvé // Ne pas modifier les indices si arrêt demandé
| | | lg++
| | end
| end
| return trouvé
end
```

En JAVA, ça donne :

```

1  public static boolean chercherLigneParLigneV1(int[][] tab, int élt) {
2      int lg, col;
3      boolean trouvé;
4
5      trouvé = false;
6      lg = 0;
7      while (lg < tab.length && !trouvé) {
8          col = 0;
9          while (col < tab[0].length && !trouvé) {
10             if (tab[lg][col] == élt) {
11                 trouvé = true;
12             } else {
13                 col++;
14             }
15         }
16         if (!trouvé) {
17             lg++;
18         }
19     }
20     return trouvé;
21 }

```

**Version avec 1 boucle.** Utilisons la même stratégie pour le parcours avec une seule boucle. Avec une boucle, on peut facilement utiliser la version avec le **test dans la condition**.

Voici ce que ça donne pour notre recherche d'un élément :

```

// Parcours avec test d'arrêt - une boucle et pas de booléen
algorithm chercherElément(tab: array of  $n \times m$   $T$ , élt:  $T$ )
|   lg, col, i: integers
|   lg = 0
|   col = 0
|   i = 1
|   while  $i \leq n*m$  ET  $tab[lg, col] \neq \text{élt}$ 
|       col = col + 1
|       if col == m                                // Passer à la case suivante
|           col = 0                                // On déborde sur la droite, passer à la ligne suivante
|           lg = lg + 1
|       end
|       i = i + 1
|   end
|   return  $i \leq n * m$ 
end

```

Remarquez qu'ici, naturellement, la ligne et la colonne s'arrêtent exactement à l'endroit où la valeur est trouvée.

En JAVA, ça donne :

```

1  public static boolean chercherLigneParLigneV2(int[][] tab, int élt) {
2      int nbÉléments = tab.length * tab[0].length;
3      int lg = 0;
4      int col = 0;
5      int i = 1;
6      while (i <= nbÉléments && tab[lg][col] != élt) {
7          col++;
8          if (col == tab[0].length) {
9              col = 0;
10             lg++;
11         }
12         i++;
13     }
14     return i <= nbÉléments;
15 }

```

**Parcours plus compliqué - le serpent.** Envisageons un parcours plus difficile où on inverse le sens du parcours d'une ligne à l'autre comme illustré par le tableau suivant.

1	2	3	4	5
10	9	8	7	6
11	12	13	14	15

Le plus simple est d'adapter l'algorithme de parcours avec une seule boucle en introduisant un sens de déplacement, ce qui donne l'algorithme :

```
// Parcours du serpent dans un tableau à deux dimensions
algorithm afficherElémentsSerpent(tab: array of  $n \times m$  T)
    Se placer sur la 1re case
    Commencer à se déplacer à droite
    repeat  $n * m$  times
        print tab[lg, col]
        if le déplacement dans le sens courant (droite ou gauche) ne sort pas du tableau
            Se déplacer dans le sens courant
        else
            Passer à la ligne suivante
            Inverser le sens de déplacement (droite  $\iff$  gauche)
        end
    end
end
```

Pour retenir le sens du déplacement, nous allons utiliser un entier ( $-1$  pour gauche et  $1$  pour droite) ce qui va faciliter le calcul du déplacement.

```
// Parcours du serpent dans un tableau à deux dimensions
algorithm afficherElémentsSerpent(tab: array of  $n \times m$  T)
    lg, col, depl: integers
    lg = 0; col = 0; sens = 1 // 1 pour avancer, -1 pour reculer
    for i from 1 to  $n*m$ 
        print tab[lg, col] // On peut faire autre chose qu'afficher
        if  $0 \leq col + sens$  ET  $col + sens < m$  // On se déplace dans la ligne
            col = col + sens
        else // On passe à la ligne suivante
            lg = lg + 1 // et on change de sens
            sens = -sens
        end
    end
end
```

Voici la version JAVA :

```
1 public static void parcoursSerpent(int[][] tab) {
2     int nbÉléments = tab.length * tab[0].length;
3     int lg = 0;
4     int col = 0;
5     int sens = 1;
6     for (int i = 0; i < nbÉléments; i++) {
7         System.out.println(tab[lg][col]);
8         if (0 <= col + sens && col + sens < tab[0].length) {
9             col += sens;
10        } else {
11            lg++;
12            sens = -sens;
13        }
14    }
15 }
```

## 1.4 Exercices

### Exercice 6 Affichage

Écrire un algorithme qui affiche tous les éléments d'un tableau (à  $n$  lignes et  $m$  colonnes) ligne par ligne.



Écrivez un autre algorithme qui affiche cette fois les éléments colonne par colonne

### Exercice 7 Cases adjacentes

Écrire un algorithme qui reçoit un tableau d'entiers (à  $n$  lignes et  $m$  colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et *affiche* les coordonnées des cases *adjacentes*.

### Exercice 8 Les nuls

Écrire un algorithme qui reçoit un tableau ( $n \times m$ ) d'entiers et qui retourne la proportion d'éléments nuls dans ce tableau.



### Exercice 9 Le tableau de cotes

Soit un tableau à  $n$  lignes et  $m$  colonnes d'entiers où une ligne représente les notes sur 20 d'un étudiant et les colonnes toutes les notes d'un cours.

Écrire un algorithme recevant ce tableau en paramètre et retournant le pourcentage d'étudiants ayant obtenu une moyenne supérieure à 50%.

### Exercice 10 Le triangle de Pascal

Le triangle de Pascal est construit de la façon suivante :

- ▷ la ligne initiale contient un seul élément de valeur 1 ;
- ▷ chaque ligne possède un élément de plus que la précédente ;
- ▷ chaque ligne commence et se termine par 1 ;
- ▷ pour calculer un nombre d'une autre case du tableau, on additionne le nombre situé dans la case située juste au-dessus avec celui dans la case à la gauche de la précédente.

Écrire un algorithme qui reçoit en paramètre un entier  $n$ , et qui renvoie un tableau contenant les  $n + 1$  premières lignes du triangle de Pascal (indiquées de 0 à  $n$ ).

N.B. : le « triangle » sera bien entendu renvoyé dans un tableau carré (ce qui ne sera forcément le cas en Java).  
Quid des cases non occupées ?

Par exemple, pour  $n$  qui vaut 5, on aura le tableau ci-contre.

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1



### Exercice 11 Tous positifs

Écrire un algorithme qui reçoit un tableau ( $n \times m$ ) d'entiers et qui vérifie si tous les nombres qu'il contient sont strictement positifs. Bien sûr, on veillera à éviter tout travail inutile ; la rencontre d'un nombre négatif ou nul doit arrêter l'algorithme.



### Exercice 12 Toute une ligne de valeurs non nulles ?

Écrire un algorithme qui reçoit un tableau d'entiers (à  $n$  lignes et  $m$  colonnes) ainsi qu'un numéro de ligne et qui retourne un booléen indiquant si la ligne donnée du tableau ne contient que des valeurs non nulles.

```
algorithm lignePleine(tab: array of n × m integers, lg: integer) → boolean
```

Faites de même pour une colonne.

**Exercice 13****Le carré magique**

Un carré magique est un tableau d'entiers carré (c'est-à-dire possédant autant de lignes que de colonnes) ayant la propriété suivante : si on additionne les éléments d'une quelconque de ses lignes, de ses colonnes ou de ses deux diagonales, on obtient à chaque fois le même résultat.

Écrire un algorithme recevant en paramètres le tableau ( $n \times n$ ) d'entiers représentant le carré et renvoyant une valeur booléenne indiquant si c'est un carré magique ou pas.

**Exercice 14****Le contour du tableau**

On donne un tableau d'entiers `tabEnt` à  $n$  lignes et  $m$  colonnes. Écrire un algorithme retournant la somme de tous les éléments *impairs* situés sur le bord du tableau.

Exemple : pour le tableau suivant, l'algorithme doit renvoyer 32

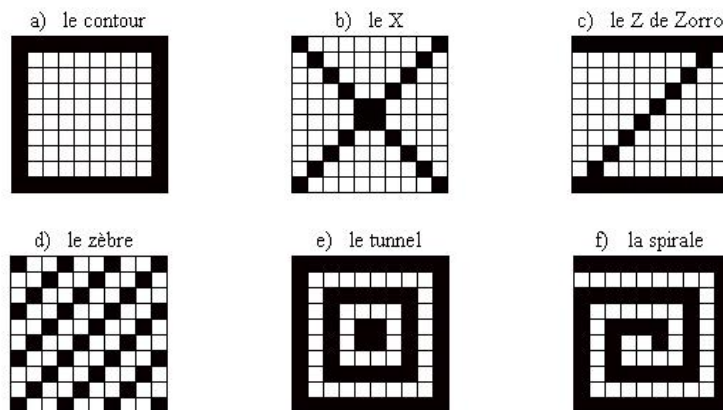
3	4	6	11
2	21	7	9
1	5	12	3

Et pour le suivant, l'algorithme doit renvoyer 6

4	1	2	8	5
---	---	---	---	---

**Exercice 15****À vos pinceaux !**

On possède un tableau à  $n$  lignes et  $n$  colonnes dont les éléments de type Couleur valent NOIR ou BLANC. On suppose que le tableau est initialisé à BLANC au départ. Écrire un algorithme qui *noircit* les cases de ce tableau comme le suggèrent les dessins suivants (les exemples sont donnés pour un tableau  $10 \times 10$  mais les algorithmes doivent fonctionner quelle que soit la taille du tableau).

**Notes**

- ▷ Le zèbre doit toujours présenter des lignes obliques et parallèles, quelle que soit la taille.
- ▷ La spirale est un véritable défi et vous est donné comme exercice facultatif. Ne le faites pas si vous êtes en retard.

**Exercice 16****Exercices sur la complexité**

Quelle est la complexité

- a) d'un algorithme de parcours d'un tableau  $n \times n$  ?
- b) des algorithmes que vous avez écrits pour les exercices : "Les nuls", "Tous positifs", "Le carré magique" et "Le contour d'un tableau" ?
- c) des algorithmes que vous avez écrits pour résoudre les exercices du pinceau ?