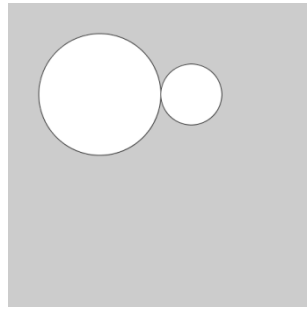


Exercices Processing

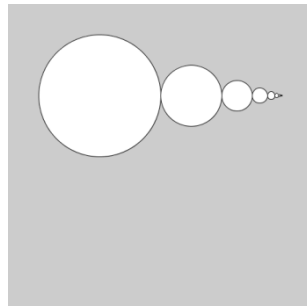
Exercice 1 : Fractales

La fenêtre fait 500 par 500

1. Le grand cercle est centré en (150, 150) et son rayon est de 100. Le petit cercle est tangent au grand et son rayon est de 50.



2. Le processus continue tant que les rayon des cercles est supérieur à 1



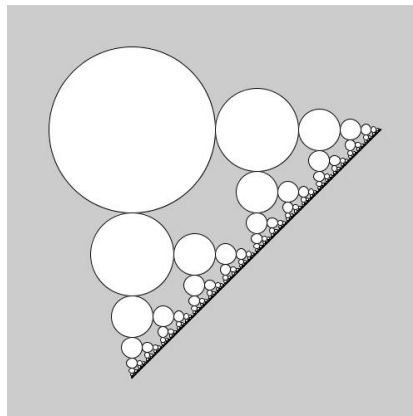
version a : utiliser une boucle for ou while

version b : utiliser la récursivité (boucle interdites)

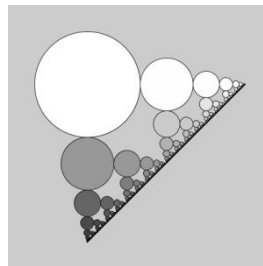
Code de départ

```
void setup() {  
  size(500, 500);  
  fractale(150, 150, 100);  
}  
void fractale(float x, float y, float r) {  
  // condition d'arret : tant que le rayon est supérieur à 0.1  
  // appel(s) récursif(s)  
}  
void cercle(float x, float y, float r) {  
  ellipse(x, y, 2*r, 2*r);  
}
```

3. En utilisant la récursivité, reproduire le dessin :



4. Bonus : intégrer des niveaux de gris qui montrent l'ordre de dessin
Ajouter une variable globale (exceptionnellement) float couleur = 255 et la décrémenter de 0.2 à chaque dessin de cercle. Autre option pour éviter d'estimer le pas (0.2) des niveaux de gris : modifier la plage de valeur des niveau de gris (0-255 par défaut) avec colorMode

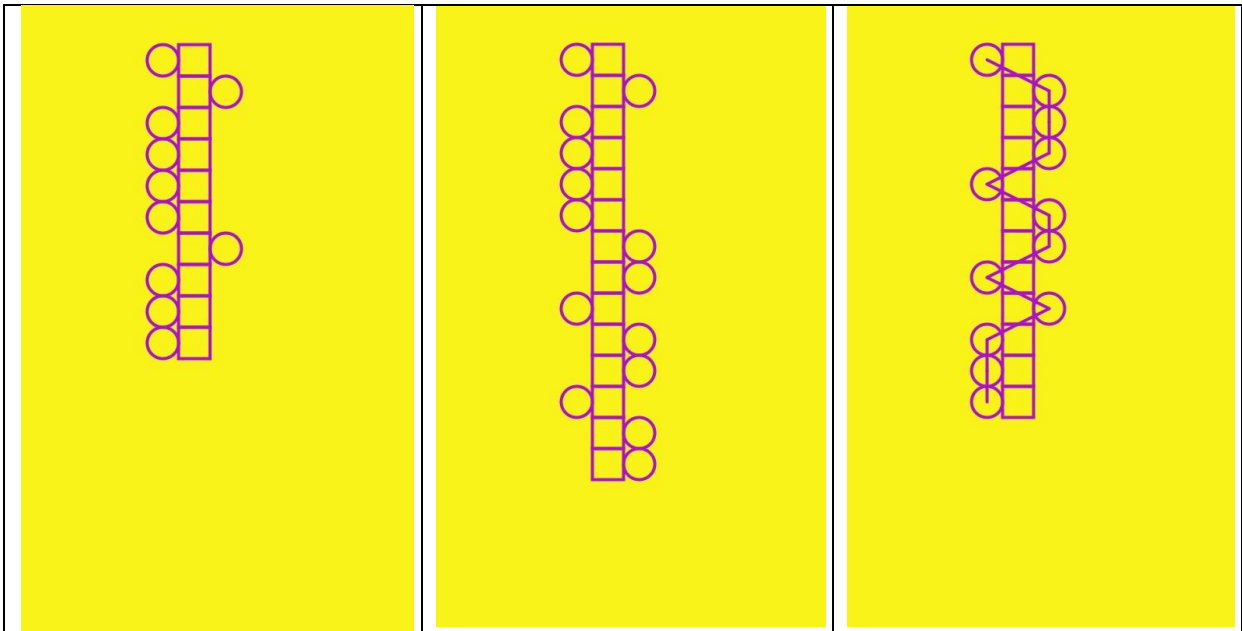


Exercice 2.5 Récursion terminale

Construction du dessin à reproduire dans une fenêtre 500*800 à l'aide d'une fonction récursive (paramétrage libre)

Le coin supérieur gauche du premier carré est en (200, 50). A côté de chaque carré (de côté = 40), on dessine un cercle de rayon (de diamètre = 40) : soit à gauche, soit à droite (selon un tirage aléatoire)

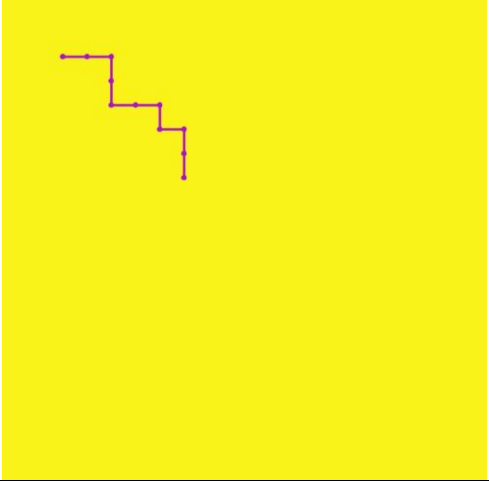
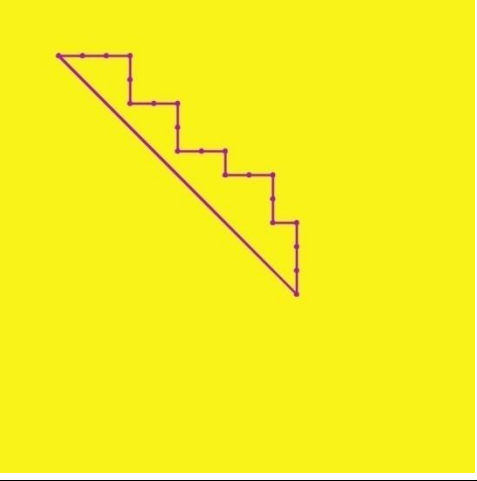
Version 1	Version 2	Version 3
10 maillons exactement	au moins 8 maillons et autant à de cercles à droite et à gauche	idem avec les cercles reliés



Exercice 2.7 Récursion terminale

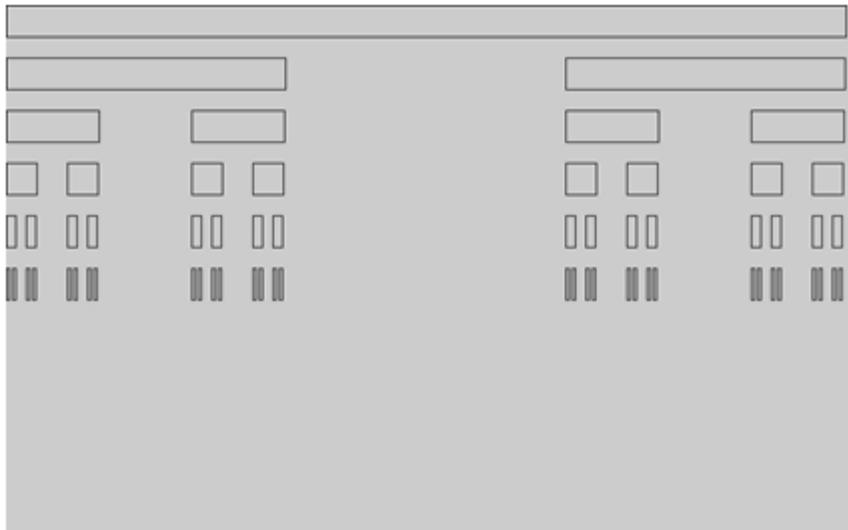
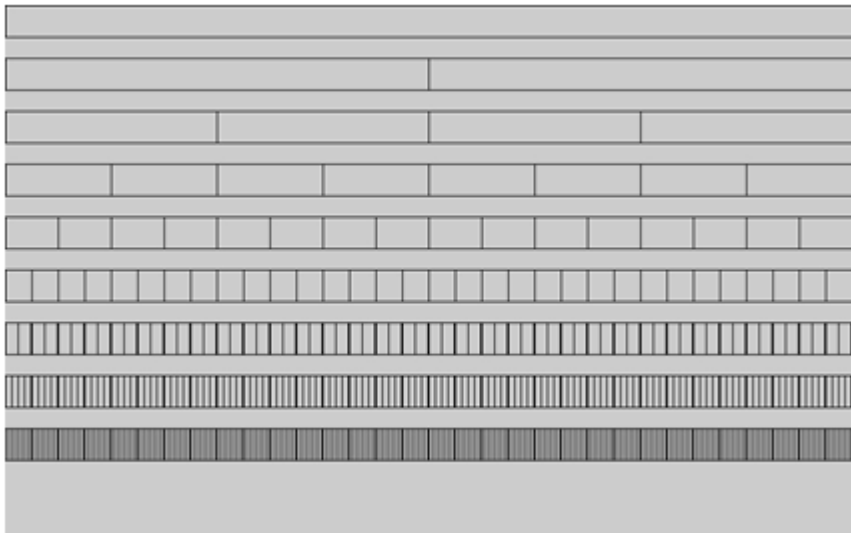
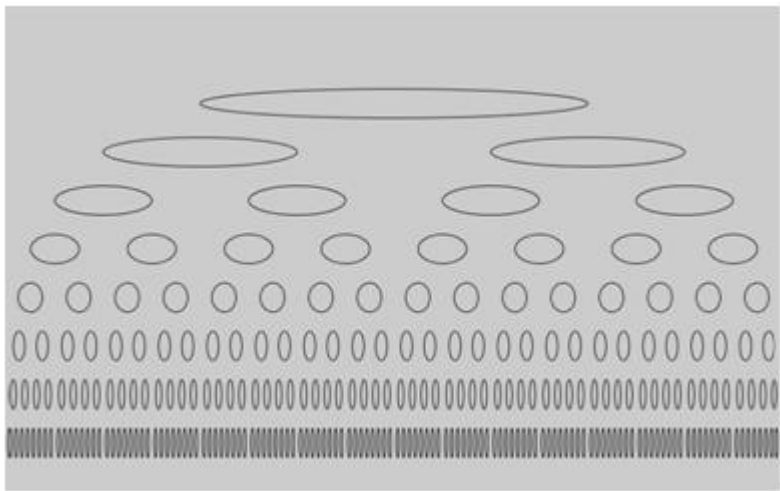
Construction du dessin à reproduire dans une fenêtre 800*800 à l’aide d’une fonction récursive (paramétrage libre)

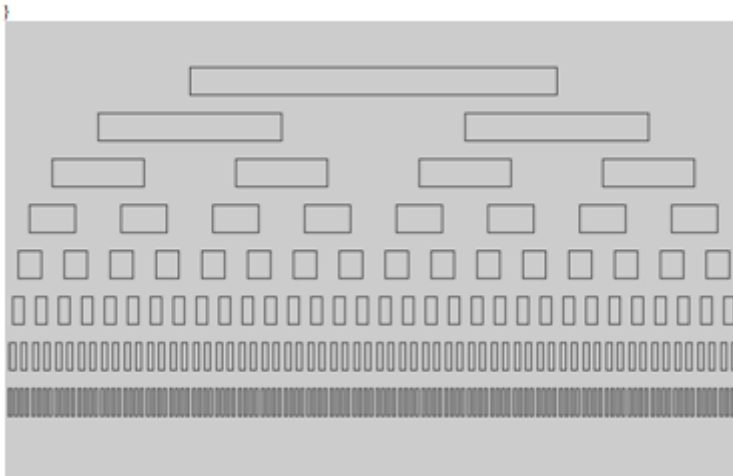
Le principe est celui d’un agent aléatoire qui va, à chaque pas, soit à gauche, soit en bas (40 pixels). A chaque pas, il dessine également un petit cercle. Le point de départ est en (100, 100).

Version 1	Version 2	Version 3
10 pas exactement	au moins 6 pas et autant à de cercles à droite et à gauche et avec une ligne entre les points de depart et d’arrivée.	idem mais en changeant de couleur quand on “traverse la diagonale”
		

Exercice 2.8

Reproduire les dessins suivants :



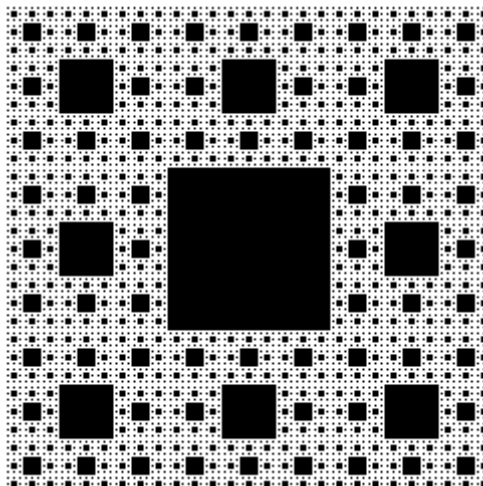


Aide à la construction :

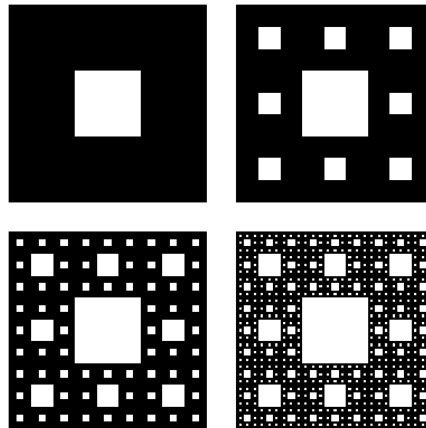
- le dessin se fait du haut en bas.
- les couches font 30 pixels de hauteur
- il y a 20 pixels entre chaque couche
- Entre chaque couche, les largeurs des formes est divisé par 2 ou 3
- On s'arrête quand les formes font moins de 2 pixels de largeur

Exercice 3

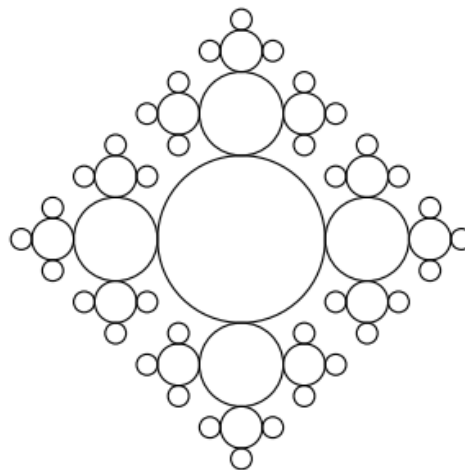
Reproduire le dessin suivant :



Indice

**Exercice 2 Fractales (suite)**

Ecrire une fonction récursive *Fractale2* permettant d'obtenir le dessin ci-dessous. On pourra introduire un paramètre supplémentaire qui précise la position du voisin-père. On pourra choisir pour le paramètre position un char (n, s, e, o).



Les fonctions récursives :

- 1. Peuvent permettre d'écrire du code plus simplement*
- 2. souvent au prix d'une exécution plus lente et/ou plus couteuse en mémoire*
- 3. dans les algos de type « force brute », permet de paramétrer la dimension de l'espace dont on cherche à énumérer les éléments.*

TP de synthèse

Tableau magique & n-uplets

Dans cet exercice, on s'intéresse aux tableaux d'entiers (de taille quelconque, notée n) dits *magiques* car présentant la propriété suivante :

$t[i] = \text{nombre de fois où l'entier } i \text{ apparaît dans le tableau}$

Exemple

- le tableau $t = \{1, 2, 1, 0\}$ pour $n=4$
 - o t contient
 - une fois la valeur 0
 - deux fois la valeur 1
 - une fois la valeur 2
 - zero fois la valeur 3
- le tableau $t = \{2, 0, 2, 0\}$ pour $n=4$
- le tableau $t = \{2, 1, 2, 0, 0\}$ pour $n=5$
- le tableau $t = \{6, 2, 1, 0, 0, 0, 1, 0, 0, 0\}$ pour $n=10$

0	1	2	3
1	2	1	0

Le but du TP est de déterminer, pour un entier n donné en paramètre¹

- si un tel tableau existe
- si oui, quelles sont ses valeurs
- et s'il est unique

Question préalable

- ➔ Ecrire une fonction `boolean ok(int[] tab)` qui renvoie *true* sur de tels tableaux et *false* sinon

Solution :

```
boolean ok(int[] tab) {
    for (int i = 0; i < tab.length; i++) {
        int nb = 0;
        for (int j = 0; j < tab.length; j++) {
            if (tab[j] == i) nb++;
        }
        if (tab[i] != nb) return false;
    }
    return true;
}
```

¹ la valeur de n (la taille du tableau) peut être issue d'un tirage aléatoire ou d'une saisie clavier : la valeur de n est connue à l'exécution du code, et pas au moment de son écriture.

Méthode 1 : Avec un générateur de code

1. Vérifier que la fonction `check4` affiche bien tous les tableaux magiques à 4 éléments

```
void setup() {
  check4( );
}

void check4( ) {
  int a, b, c, d;
  for (a = 0; a<4; a++)
    for (b = 0; b<4; b++)
      for (c = 0; c<4; c++)
        for (d = 0; d<4; d++) {
          int [] tab = {a, b, c, d};
          if (ok(tab))println(tab);
        }
}
```

Version compacte:

```
void check4( ) {
  int [] tab = new int [4];
  for (tab[0] = 0; tab[0]<tab.length; tab[0]++)
    for (tab[1] = 0; tab[1]<tab.length; tab[1]++)
      for (tab[2] = 0; tab[2]<tab.length; tab[2]++)
        for (tab[3] = 0; tab[3]<tab.length; tab[3]++)
          if (ok(tab))println(tab);
}
```

2. Ecrire une fonction `String ecrireCheckN(int n)` qui renvoie un `String` contenant le code ci avant (setup compris) pour `n = 4`. Vous pouvez ajouter la fonction `ok`. Pour copier automatiquement ce code dans un nouveau sketch processing, vous pouvez utiliser l'instruction :

```
String[] t={res}; saveStrings("../check"+n+"/check"+n+".pde", t);
```

Méthode 2 : Avec la récursivité

Préambule : affichage des chaînes de bits de taille n

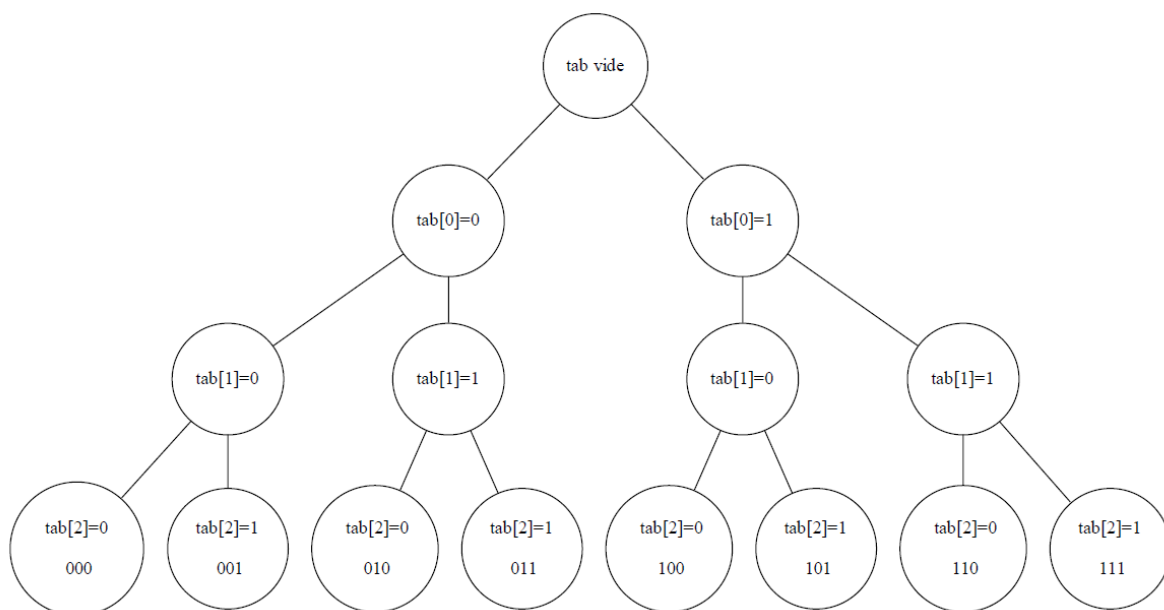
Par exemple, pour $n = 2$, la console doit afficher : 00, 01, 10, 11

Soit `tab` un tableau de taille n . On cherche à remplir ce tableau de toutes les manières possibles avec des 0 et des 1. Pour ce faire, on va écrire une fonction récursive `void nUplets(int [] tab, int ind)` qui devra :

- affecter une valeur à `tab[ind]` (0 ou 1)
- s'appeler sur l'indice d'après

Début de solution à compléter :

```
void setup() {
  int n = 3;
  int[] tab = new int[n];
  nUplets(tab, 0);
}
void nUplets(int [] tab, int ind) {
  if (//condition d'arrêt) {
    // ?
  } else {
    tab[ind]=0;
    nUplets(tab, ind+1);
    tab[ind]=1;
    nUplets(tab, ind+1);
  }
}
```



➔ Adapter cette solution au problème des tableaux magiques

➔ **Amélioration : backtracking**

- Combien vaut la somme des éléments d'un tableau solution ?
- L'amélioration proposée consiste à ne pas faire d'appels récursif si la somme des éléments est déjà *strictement* supérieure à cette somme maximale. On pourra coder une fonction `int sommeTab(int[] tab, int ind)` qui renvoie la somme des éléments de tab dont l'indice est inférieur ou égal à ind
- Comparer le nombre total d'appels récursifs (variable globale permise à cet effet), ainsi que le temps d'exécution. Expliquer le résultat obtenu.

Méthode 3 : Par itération dans un ensemble et recherche de point fixe

1. Modifier la fonction ok en une fonction `int[] step(int[] tab)`
 - qui prend en paramètre un tableau d'entiers tab
 - et qui renvoie un tableau res, tel que :
res[i] = nombre de fois où l'entier i apparaît dans le tableau tab
2. Ecrire une fonction `int[] converge(int[] tab)`
 - qui prend en paramètre un tableau initialisé aléatoirement
 - qui itère la fonction step sur le tableau renvoyé
 - jusqu'au moment où ce tableau ne varie plus²
 - renvoyer ce tableau

Par exemple, la fonction step doit vous donner la trajectoire de tableaux suivants :

{1, 1, 1, 2} → {0, 3, 1, 0} → {2, 1, 0, 1} → {1, 2, 1, 0} → {1, 2, 1, 0} → {1, 2, 1, 0} ...

Remarque : si la solution n'existe pas, ce qui est le cas pour n = 2 par exemple, ou si la fonction converge est coincée dans une boucle (par exemple, pour n = 7, 3300100 → 4102000 → 4110100 → 3300100). Il sera judicieux de prévoir un système d'arrêt après un temps de recherche maximum (cf https://processing.org/reference/millis_.html)

```
import java.util.Arrays;
void setup() {
  int[] tab1 = {1, 2, 3};
  int[] tab2 = {1, 2, 3};
  println(tab1==tab2);
  println(Arrays.equals(tab1, tab2));
}
```

² `import java.util.Arrays` puis `Arrays.equals(tab1, tab2)`

Méthode 4 : Génération des tableaux concaténés

Exemple sur chaînes de bits de taille n

Ici, l'idée est de créer un grand tableau, où tous les tableaux possibles sont collés bout à bout. Par exemple, pour les 2^3 chaînes de $n=3$ bits, le tableau comprend 24 cases et vaut :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0	0	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	0	1	1	1

Indice : faire $n=3$ boucles pour remplir le tableau :

- pour les indices tels que $\text{indice} \% 3 = 0$: 0, 0, 0, 0, 1, 1, 1, 1
- pour les indices tels que $\text{indice} \% 3 = 1$: 0, 0, 1, 1, 0, 0, 1, 1
- pour les indices tels que $\text{indice} \% 3 = 2$: 0, 1, 0, 1, 0, 1, 0, 1

➔ Adapter cette solution au problème des tableaux magiques

Méthode 5 : Programmation dynamique

L'idée est de créer les tableaux possibles à n éléments à partir de l'ensemble des tableaux possibles à $n-1$ éléments (et ainsi de suite).

Version sans liste :

- Les tableaux seront stockés dans un tableau de tableau : `int[][] tab`
- on pourra coder une fonction `int[] add(int[] tab, int val)` qui « ajoute » `val` à la fin du tableau `tab` (et renvoie le tableau ainsi obtenu)

Version avec : vous pouvez utiliser le code suivant :

```
ArrayList<int[]>maliste = new ArrayList<int[]>(); //une liste maliste
de tableau d'entier
int[] tab = {2, 4, 7, 2, 9};
int[] tab2 = {2, 4, 7, 2, 9};
maliste.add(tab); //pour ajouter un élément à l
maliste.add(tab2);
int[] val = maliste.get(1); // pour accéder à l'élément de rang 1
```