Fiche d'exercices
TD19 - Récursivité

Objectifs :
- Savoir définir et appeler des fonctions récursives
- Bien définir les conditions d'arrêt de récursivité

Exercice 1 : Arithmétique

1.1 Factorielle

Pour rappel, $0! = 1$ et $n! = 1 \times 2 \times 3 \times ... \times n = (n-1)! \times n$

Une version du calcul de la factorielle de n sans récursivité serait :

```
1 def factorielle (n):
2     res = 1
3     for i in range(1, n+1):
4         res = res * i
5     return res
```

L'objectif est de proposer cette fois-ci une écriture qui utilise complètement le concept de récursivité.

(1.1) Écrire la fonction récursive $factorielle\_rec$ qui prend en paramètre un entier n et renvoie $n!$.

1.2 Suite récurrente

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par $u_0 = 4$ et $u_{n+1} = \dfrac{u_n}{2} + 1$ pour tout $n \in \mathbb{N}$.

(1.2) Écrire la fonction récursive $terme$ qui prend en paramètre un entier n et renvoie u_n.

1.3 Puissance

(1.3) Écrire la fonction récursive $puissance$ qui prend en paramètre deux entiers x et n et renvoie x^n.

Les mathématiciens indiens mirent au point une méthode plus efficace, se basant sur cette remarque :

$$x^{2n+1} = (x^2)^n * x \quad \text{et} \quad x^{2n} = (x^2)^n$$

Par exemple, pour calculer x^{13}, on peut se rendre compte que $x^{15} = (x^2)^6 * x$, et pour calculer $(x^2)^6 = y^6$, on peut aussi écrire $y^6 = (y^2)^3$, soit $(x^2)^6 = ((x^2)^2)^3$. En faisant de même avec $y^3 = y^2 * y$, on obtient finalement la valeur de x^{13} en écrivant :

$$x^{13} = ((x^2)^2)^2 * (x^2)^2 * x = x^{8+4+1}$$

Ainsi, là où $puissance$ nécessitait 12 multiplications pour calculer x^{13}, cette méthode n'en nécessite que 5 (calcul de $x^2, (x^2)^2, ((x^2)^2)^2$, puis multiplications des différents termes).

(1.4) Écrire la fonction récursive $puissance\_v2$ qui prend en paramètre deux entiers x et n et renvoie x^n, calculé à l'aide de cette méthode. ***Cette fonction ne fera pas appel à la fonction*** $puissance$***.***

(1.5) Comparer le nombre de multiplications effectuées par ces deux fonctions.

1.4 Coefficient Binomial

Le triangle de Pascal est une présentation des coefficients binomiaux sous la forme d'un triangle :

n | **p**	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

On peut de fait définir un coefficient binomial $\binom{n}{p} = \frac{n!}{p!(n-p)!}$ de manière récursive :

$$\begin{cases} \binom{0}{p} &= 1 \\ \binom{n}{0} &= 1 \\ \binom{n}{n} &= 1 \\ \binom{n}{p} &= \binom{n-1}{p-1} + \binom{n-1}{p} \end{cases}$$

(1.6) Écrire la fonction $coeff\_binomial$ qui prend en paramètre deux entiers n et p et renvoie $\binom{n}{p}$, calculé de manière récursive.

Exercice 2 : Tri fusion

Soit une liste d'entiers de longueur n donnée en entrée. Le tri par fusion est l'un des algorithmes de tri les plus populaires et les plus efficaces, qui est basé sur le paradigme **Diviser pour régner**.

La remarque qui explique l'intérêt de ce tri est qu'on peut facilement construire une liste triée comportant les éléments issus de deux listes initialement triées (leur **fusion**). Ainsi, dans la figure 1, la partie basse (en vert) correspond à des fusions de 2 listes initialement triées.

Il "suffit" donc, en amont de ces fusions, de découper (**diviser**) récursivement la liste initiale en sous-listes de plus petite taille, jusqu'à obtenir des listes de taille unitaire (partie haute en orange de la Figure 1).

L'algorithme du tri fusion peut donc s'écrire de la manière suivante :

```
# debut/fin représente l'indice de début et de fin entre lesquels on chercher à trier
fonction tri_fusion(l, debut, fin):
    # Par défaut, la liste résultat contient l'unique élément l[debut]

    # Si la liste a plus qu'un seul élément
        # Trouvez le point milieu pour diviser le tableau en deux morceaux
        # Appelez tri_fusion pour la partie de gauche (entre début et milieu)
        # Appelez tri_fusion pour la partie de droite (entre milieu et fin)
        # Fusionnez les deux morceaux
    #  retourner le résultat

fonction tri(l):
    #renvoyer tri_fusion(l, 0, taille(l)-1)
```

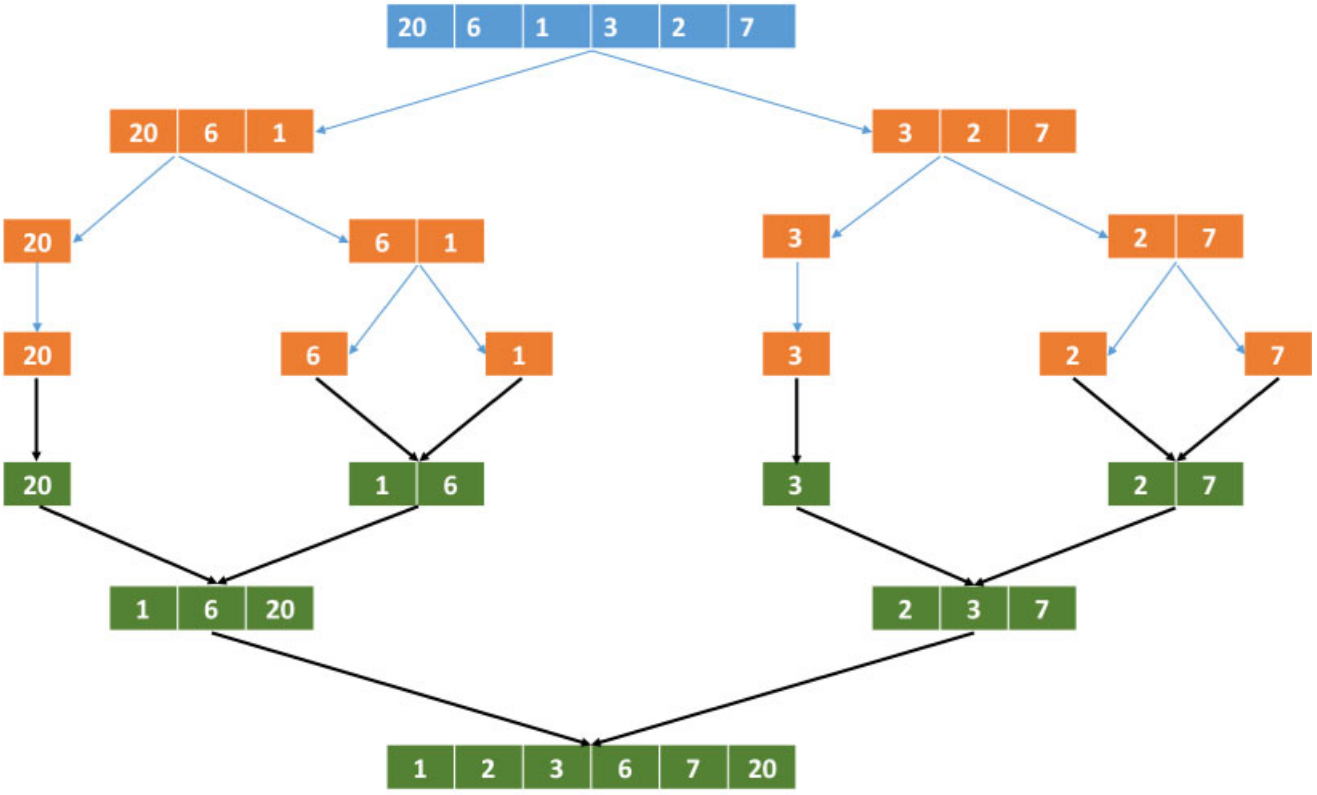
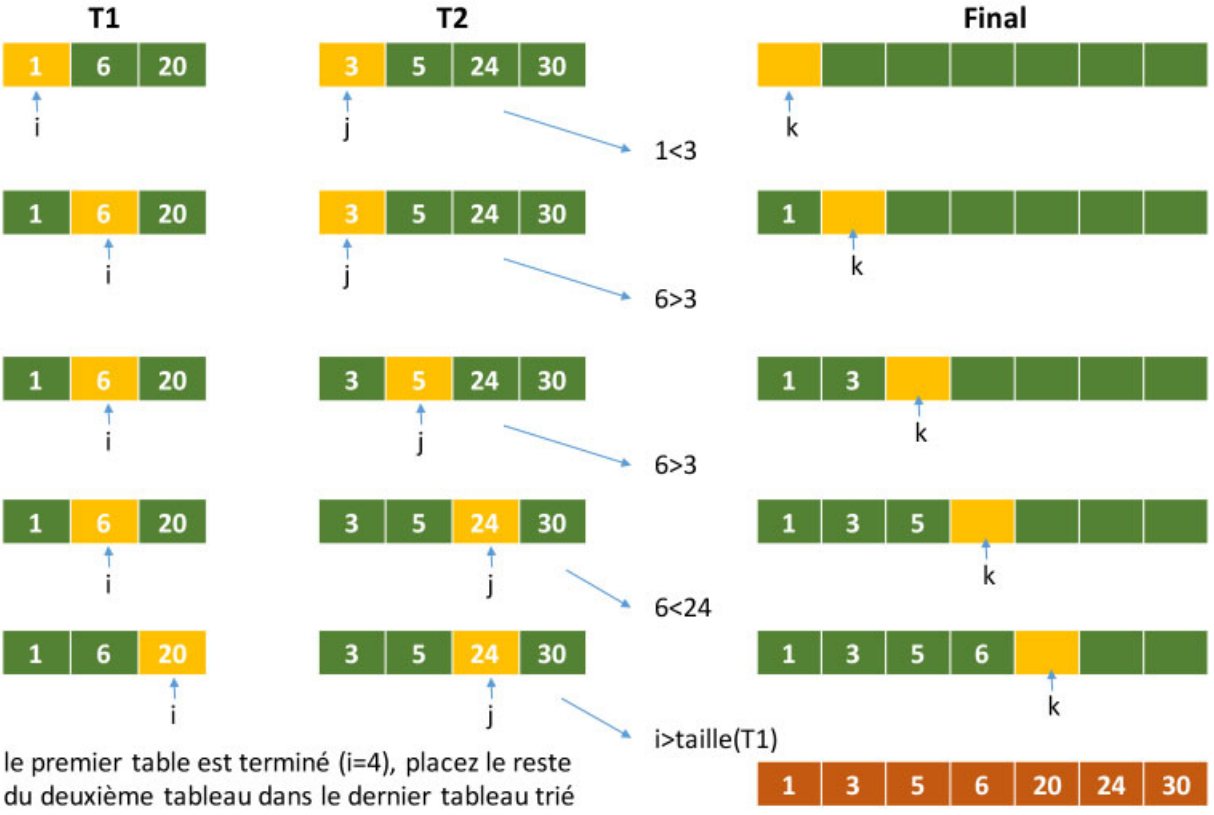


FIGURE 1 – Exemple d'exécution

L'étape de fusion est, en elle-même, délicate. On peut la coder via l'algorithme suivant :

```
Est-ce que nous avons atteint la fin de l'un des tableaux?
-> Non
        - Comparer les éléments actuels des deux tableaux (T1[i] et T2[j])
        - Copiez l'élément le plus petit dans le tableau trié
        - Déplacer le pointeur de l'élément contenant un élément plus petit (i ou j)
-> Oui
        - Copiez tous les éléments restants du tableau non vide
```



(2.1) *Concevoir une fonction* ***non-récursive*** $fusion\_triee$ *fusionnant deux listes triées. Vous pourrez reprendre le code réalisé pendant le TD 15.*

(2.2) *Coder l'algorithme du tri fusion.*

(2.3) *Tester votre programme sur des listes générées aléatoirement (vous pouvez réutiliser la fonction du TD* $creer\_liste\_aleatoire$ *du TD 15.*

(2.4) *Modifier votre programme pour qu'il compte le nombre de comparaisons effectuées lors d'une exécution. Comparer les performances de cet algorithme de tri avec celui du tri à bulle à l'aide de tests sur des listes aléatoires.*

Exercice 3 : Flocon de Von Koch

Le but de cet exercice est de tracer une ligne brisée qui s'approche de l'objet fractal appelé le Flocon de Von Koch. Pour construire cette ligne brisée, on utilise une approche récursive : on part d'un triangle équilatéral, et on applique à chaque étape les modifications suivantes à chaque segment

1. Chaque côté de la figure est découpé en 3 segments de longueurs égales.
2. Parmi les 3 nouveaux segments de chaque côté, celui du milieu sert de base pour créer un triangle équilatéral.
3. On supprime le segment qui a servit de base pour l'étape 2.

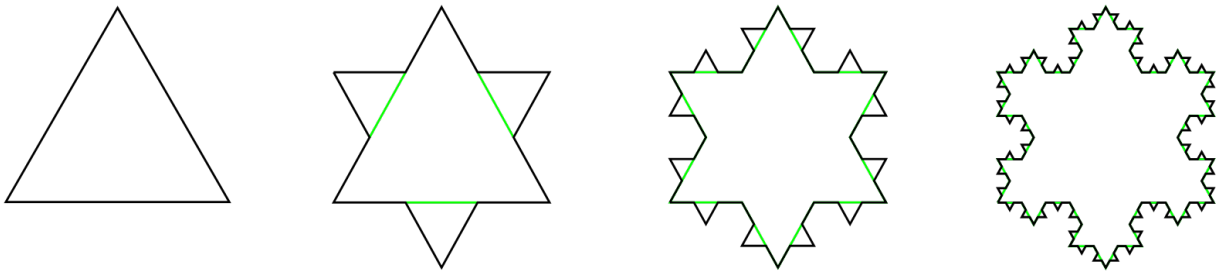


FIGURE 2 – Les quatres premières itérations

Ce Flocon est en fait constitué de trois côtés qui sont similaires à rotation près, qu'on appelle aussi courbe de Koch.

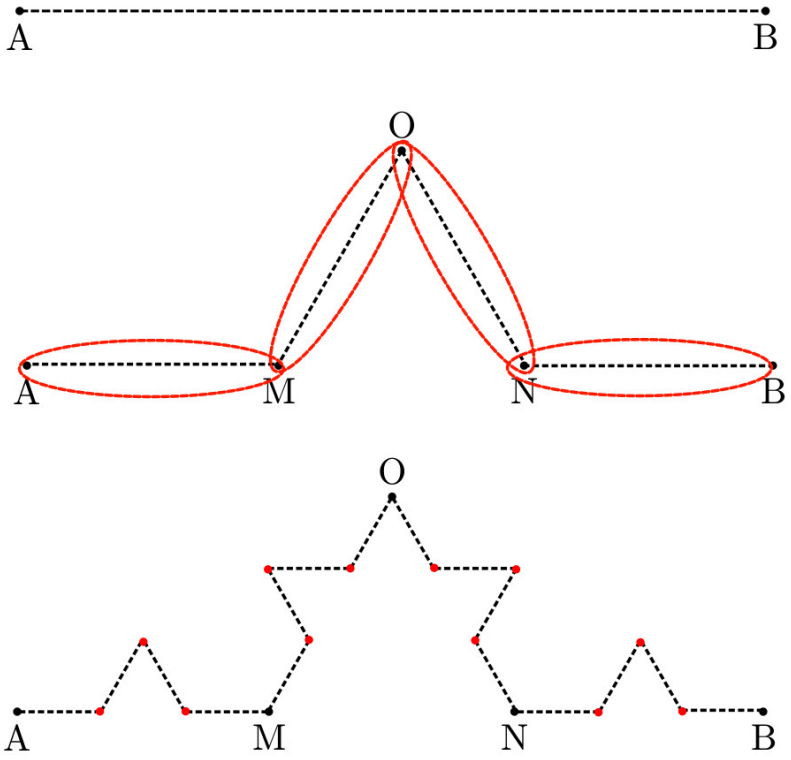


FIGURE 3 – Les deux premières itérations de la courbe de Koch

La fonction $trace$ ci-dessous prend en entrée deux tableaux Numpy à deux éléments (représentant des points du plan) et trace le segment les reliant.

```
import matplotlib.pyplot as plt
import numpy as np

def trace(a,b):
    x0,y0 = a
    x1,y1 = b
    plt.plot([x0,x1], [y0,y1])
```

On admet que le code suivant permet de calculer la pointe du triangle de la prochaine itération, étant donné d et f les extrémités d'un segment sur lequel on veut appliquer la transformation décrite ci-dessus. Par exemple, sur la figure ci-dessus, il permet de calculer les coordonnées du point O à partir des coordonnées des points A et B.

```
def calcul_pointe(deb, fin):
    '''
        Entrées: les coordonnées (numpyarray de dimension 2)
            des extrémités d'un segment
        Sortie: les coordonnées de la pointe correspondante
    '''
    l = np.linalg.norm(fin - deb)
    rotation = np.array([[0,-1],[1,0]])
    vecteur_unitaire = (fin - deb)/l
    perpendiculaire = np.dot(rotation,vecteur_unitaire)
    milieu = (fin + deb)/2
    a2 = milieu + perpendiculaire*(np.sqrt(3)*l/6)
    return a2
```

(3.1) Écrire la fonction $koch$, qui prends en entrée le nombre d'itérations, le point de début et de fin du segment, et qui trace la courbe de Koch correspondante sur ce segment.

On pourra ensuite utiliser cette fonction pour tracer le flocon en entier avec la fonction suivante :

```
def flocon(n):
    plt.axis("equal")
    plt.axis("off")
    a = np.array([0, 0])
    b = np.array([1, 0])
    c = np.array([0.5, np.sqrt(3)/2])
    koch(n,a,c)
    koch(n,c,b)
    koch(n,b,a)
    plt.show()
```

Exercice 4 : (Bonus) Le problème des dames

Le but du problème des huit dames est de placer huit dames d'un jeu d'échecs sur un échiquier de 8×8 cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs. Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale. La figure qui suit est un exemple de solution.

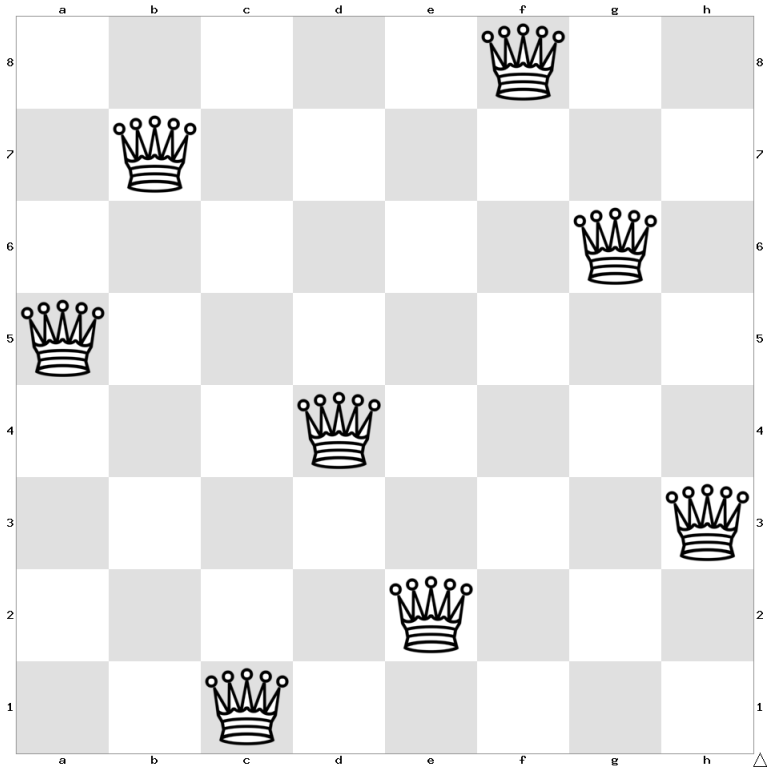


FIGURE 4 – Exemple de solution au problème des huits dames

Ce problème se généralise naturellement à un échiquier de taille $n \times n$ et l'objectif de l'exercice est de calculer (et afficher) les solutions à ce problème. Pour ce faire, nous allons procéder ligne par ligne en essayant toutes les possibilités compatibles avec les choix déjà fait jusqu'alors. Pour représenter les positions des dames déjà placées durant l'exécution, nous stockerons la liste des colonnes correspondantes. Ainsi la liste [5, 1, 6] correspond aux trois premiers lignes de la solution illustrée précédemment, tandis que la solution complète peut être encodée par la liste [5, 1, 6, 0, 3, 7, 4, 2].

```
n = 8
positions = [5, 1, 6, 0, 3]
affiche(n, positions)
# .....X..
# .X......
# ......X.
# X.......
# ...X....
# ........
# ........
# ........
```

```
en_prise(2, 0, 0, 0) # True
en_prise(2, 1, 0, 0) # False
en_prise(2, 2, 0, 0) # True
en_prise(1, 3, 0, 0) # False
en_prise(0, 3, 0, 0) # True
```

(4.1) *Écrire la fonction $affiche(n, positions)$ qui prend en entrée la taille de l'échiquier et les positions des dames placées suivant la convention précédente, et qui affiche la solution de la manière suivante :*

(4.2) *Écrire la fonction $en\_prise(x1, y1, x2, y2)$ qui prend en entrée une case (x_1, y_1) et la position (x_2, y_2) d'une dame et renvoie $True$ si la dame menace la case, et $False$ sinon. La coordonnée X correspond à l'indice de la ligne tandis que la deuxième coordonnée y correspond à l'indice de la colonne.*

(4.3) *Écrire la fonction $est\_libre(x1, y1, positions)$ qui prend en entrée une case (x_1, y_1) et la liste des positions des dames déjà placées (selon la convention précédente), et qui renvoie $True$ si la case n'est menacée par aucune dame, et $False$ sinon.*

```
est_libre(0, 5, [5, 1, 6, 0, 3]) # False
...
est_libre(6, 7, [5, 1, 6, 0, 3]) # False
est_libre(7, 5, [5, 1, 6, 0, 3]) # True
```

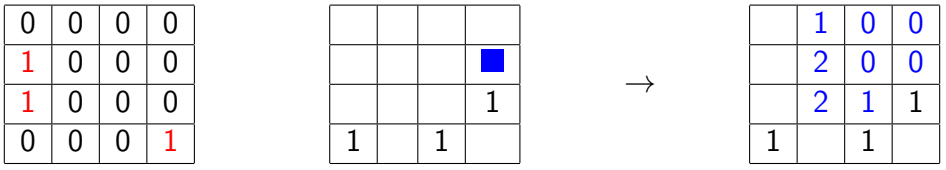
(4.4) *Écrire la fonction récursive $place\_dames(n, positions)$ qui calcule le nombre de solutions du problème, et qui les affiche.*

Exercice 5 : A FAIRE SUR PAPIER Démineur

Dans cet exercice on revient sur le jeu du démineur et plus précisément sur ce qui se passe quand on révèle une case qui n'est adjacente à aucune bombe : les cases adjacentes sont révélées, et si l'une d'entre elles est également adjacente à aucune bombe, on révèle ses voisines, etc.
On suppose qu'on a :

- une variable $grille\_privee$ qui contient des 0 et des 1, un 1 représentant une bombe
- une variable $grille\_publique$ qui représente la grille affichée à l'utilisateur : elle contient -1 si la case est cachée, et sinon un nombre entre 0 et 8 correspondant au nombre de bombes autour de cette case.
- une fonction $bombes\_voisines$ qui prend deux entiers x et y et qui renvoit le nombre de bombes dans les cases voisines de la case (x, y).

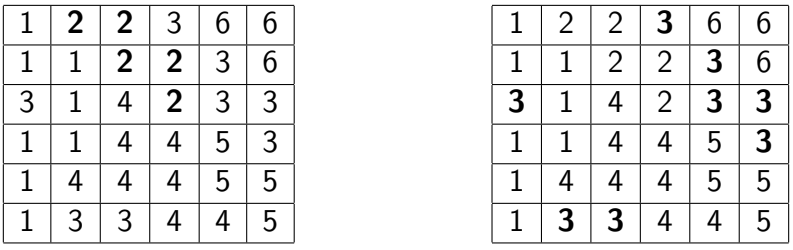
Par exemple, supposons que les deux grilles dessous à gauche soient la grille privée et la grille publique, et que l'utilisateur clique sur la case surlignée, alors la nouvelle grille publique est celle de droite.



(5.1) Proposer une fonction $revele$ qui prends la grille privée, la grille publique, les coordonnées d'une case (x, y), et qui met à jour la grille publique en fonction du nombre de bombes adjacentes à la case (x, y).

Exercice 6 : A FAIRE SUR PAPIER CandyCrush étendu

Dans cet exercice on s'intéresse au niveau 3 proposé dans le projet CandyCrush, et plus précisement à la détection de l'ensemble des cases adjacentes contenant le même numéro qu'une case de départ. Par exemple sur la grille de nombres ci-dessous **à gauche**, on a mis en avant toutes les cases reliées à la case $(1, 2)$:



(6.1) Proposer une fonction qui donne la liste des voisins d'une case contenant la même valeur

(6.2) Proposer une fonction qui renvoie la liste de toutes les cases reliées à une case donnée

(6.3) Comment faudrait-il adapter le code précédent si on considère aussi les cases en diagonales dans la recherche d'une combinaison ?

On s'intéresse maintenant à un autre jeu qui se joue également sur une grille mais où l'objectif est de faire une suite d'échanges de bonbons adjacents pour rassembler tous les mêmes bonbons de couleur similaire. Par exemple, sur la grille précédente **de gauche**, tous les bonbons de couleur 2 sont reliés, mais pas ceux de couleur 3 puisqu'ils sont répartis en 3 zones différentes, comme illustré sur la grille précédente **de droite**.

(6.4) Proposer une fonction qui prend une grille et qui détecte si la partie est gagnée ou pas sur la grille.

On s'intéresse maintenant à un labyrinthe : une grille constitué de cases qui sont soit vide, soit avec un # pour indiquer qu'elle contient un mur :

	#	#	#		
		#	#		#
	#	#			#
				#	
#		#			
		#		#	

	#		#		
			#		#
	#	#			#
		#		#	
#		#			
		#		#	

(6.5) Proposer une fonction qui teste si on peut entrer dans le labyrinthe depuis le coin en haut à gauche et en sortir depuis le coin en bas à droite