

Listes de listes

Ceci est le corrigé

Si vous ne traitez pas une question, vous pouvez admettre son résultat pour traiter les questions suivantes. Quand vous ré-utilisez le résultat d'une question précédente, traitée ou pas, rappelez le numéro de la question concernée.

La percolation désigne le passage d'un fluide à travers un solide poreux. Ce terme s'applique bien entendu au café produit par le passage de l'eau à travers une poudre de café comprimée, mais s'applique aussi à l'infiltration de l'eau de pluie jusqu'aux nappes phréatiques ou, dans un sens plus étendu, à la propagation des feux de forêt par contact entre les feuillages d'arbres voisins.

Pour modéliser la percolation, nous allons considérer une grille carrée $n \times n$ dans laquelle chaque case peut être "ouverte" ou "fermée". La question à laquelle nous allons essayer de répondre est la suivante : est-il possible de joindre le haut et le bas de la grille par une succession de cases ouvertes adjacentes ?

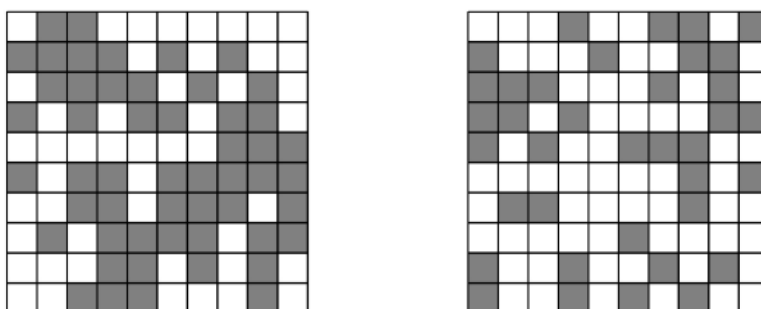


FIGURE 1 – Deux exemples de grilles 10×10 . La percolation n'est possible que dans le second cas (les cases ouvertes sont les cases blanches).

La grille de percolation sera représentée par une liste (de taille n) de listes de taille n . Pour une grille appelée `grille`, la case à la ligne i et à la colonne j est référencée par `grille[i][j]`. Dans la grille, on représentera les cases ouvertes par l'entier 0 (le fluide peut passer) et les cases fermées par l'entier 1 (le fluide ne peut pas passer).

Question 1. Écrire une fonction `grille_vide` qui prend en argument un entier n et renvoie une grille de taille $n \times n$ où toutes les cases sont ouvertes. Par exemple, `grille_vide(3)` renvoie `[[0, 0, 0], [0, 0, 0], [0, 0, 0]]`.

On veillera à ce que les différentes lignes de la grille ne soient pas des alias les unes des autres.

Solution.

On propose le code suivant :

```
def grille_vider(n):
    return [[0]*n for i in range(n)]
```

Remarque

Il existe beaucoup d'autres solutions, telles que

```
def grille_vider(n):
    grille = []
    for i in range(n):
        ligne = []
        for j in range(n):
            ligne.append(0)
        grille.append(ligne)
    return grille
```

Par contre, pour éviter que les lignes soient des alias les unes des autres, il faut éviter de mettre la même ligne plusieurs fois, ou de créer la grille elle-même par multiplication de liste. Ainsi, les propositions suivantes sont **fausses** :

```
def grille_vider_fausse(n):
    return [[0] * n] * n
```

X

```
def grille_vider_fausse(n):
    ligne = []
    for i in range(n):
        ligne.append(0)
    grille = ligne * n
    return grille
```

X

```
def grille_vider_fausse(n):
    grille = []
    ligne = []
    for i in range(n):
        ligne.append(0)
    for i in range(n):
        grille.append(ligne)
    return grille
```

X

Question 2. Écrire une fonction `lasagne` qui prend en argument un entier n et renvoie une grille de taille $n \times n$ dont les lignes sont une succession de “ligne avec toutes les cases fermées” et “ligne avec toutes les cases ouvertes”. Par exemple, l’appel `lasagne(3)` renvoie `[[1, 1, 1], [0, 0, 0], [1, 1, 1]]`.

Solution.

(Code à la page suivante)

```
def lasagne(n):  
    resultat = []  
    for i in range(n):  
        if i % 2 == 0:  
            resultat.append([1] * n)  
        else:  
            resultat.append([0] * n)  
    return resultat
```

Question 3. Écrire une fonction `foret` qui prend en argument un entier n et renvoie une grille de taille $n \times n$ dont les colonnes sont une succession de “colonne avec toutes les cases fermées” et “colonne avec toutes les cases ouvertes”. Par exemple, l’appel `foret(3)` renvoie `[[1, 0, 1], [1, 0, 1], [1, 0, 1]]`. On veillera à ce que les différentes lignes de la grille ne soient pas des alias les unes des autres.

Solution.

Les cases fermées seront celles dont l’indice de colonne est pair :

```
def foret(n):  
    grille = []  
    for i in range(n):  
        ligne = []  
        for j in range(n):  
            if j % 2 == 0:  
                ligne.append(1)  
            else:  
                ligne.append(0)  
        grille.append(ligne)  
    return grille
```

Une fois la grille créée, les cases ouvertes de la première ligne sont remplies par un fluide, ce qui sera représenté par la valeur 2 dans les cases correspondantes. Le fluide pourra ensuite être diffusé à chacune des cases ouvertes voisines d'une case contenant déjà le fluide jusqu'à remplir toutes les cases ouvertes accessibles.

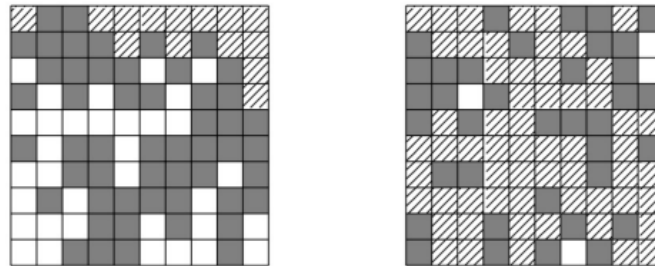


FIGURE 2 – les deux grilles de la figure 1, une fois le processus de percolation terminé (le fluide est représenté par des hachures).

Nous allons créer une fonction intermédiaire permettant de savoir si pour une case de coordonnées i et j dans la grille, un de ses voisins est rempli (valeur 2) ou non.

Question 4. Écrire une fonction `a_un_voisin_rempli` qui prend en entrée une grille de percolation grille, un indice de ligne i et un indice de colonne j , et qui renvoie `True` si un des voisins de la case de coordonnées (i, j) est rempli, et `False` sinon.

Solution.

Les voisins de la case de coordonnées (i, j) sont les cases $(i + 1, j)$ (en bas), $(i - 1, j)$ (en haut), $(i, j + 1)$ (à droite) et $(i, j - 1)$ à gauche. **Il faut faire attention à ce qu'elles soient à l'intérieur de la grille** avant de regarder si elles sont remplies ou pas. On introduit pour cela la fonction auxiliaire `est_dans_la_grille_et_rempli`.

```
def est_dans_la_grille_et_rempli(grille, i, j):
    if i < 0 or j < 0 or i >= len(grille) or j >= len(grille):
        return False
    else:
        return grille[i][j] == 2

def a_un_voisin_rempli(grille, i, j):
    return (est_dans_la_grille_et_rempli(grille, i+1, j) or
            est_dans_la_grille_et_rempli(grille, i-1, j) or
            est_dans_la_grille_et_rempli(grille, i, j+1) or
            est_dans_la_grille_et_rempli(grille, i, j-1))
```

Voici en français la description d'un algorithme **faux** de percolation dans une grille :

- Dans la première ligne de la grille, remplir de liquide toutes les cases ouvertes (passage de la valeur 0 à la valeur 2).
- Pour chaque ligne, de haut en bas :
 - Pour chaque case de la ligne, de gauche à droite :
 - Si la case est ouverte et a un voisin rempli de liquide, la remplir de liquide.

Question 5. Si la grille initiale est $[[1, 0, 1], [0, 0, 0], [0, 1, 1]]$, décrivez l'exécution de l'algorithme et expliquez en quoi le résultat obtenu est faux.

Solution.

On commence avec $[[1, 0, 1], [0, 0, 0], [0, 1, 1]]$, après avoir rempli les cases ouvertes de la première ligne, on a $[[1, 2, 1], [0, 0, 0], [0, 1, 1]]$. Quand on traite la première case de la deuxième ligne, on **ne** la remplit **pas** car à ce moment aucune de ses cases voisines est remplie. Puis on traite les deux suivantes : on remplit celle du milieu car son voisin du dessus est rempli. On remplit celle de droite car son voisin de gauche est rempli. On obtient donc, après avoir traité la deuxième ligne, la grille suivante : $[[1, 2, 1], [0, 2, 2], [0, 1, 1]]$. Enfin, quand on traite la seule case ouverte de la dernière ligne, aucune de ses cases voisines est remplie de liquide, donc on ne la remplit pas.

On obtient $[[1, 2, 1], [0, 2, 2], [0, 1, 1]]$ en résultat final, ce qui est faux car les premières cases de la deuxième et troisième ligne auraient dû se remplir de liquide.

Question 6. Proposer en français une correction à la description de l'algorithme.

Solution.

Plutôt que de passer une seule fois sur la grille, on devrait y passer autant de fois que nécessaire jusqu'à ce qu'il n'y ait plus de changements dans la grille.

Remarque

Une autre solution serait de, quand on remplit une case, re-parcourir ses cases voisines.

Question 7. Coder une fonction percolation qui prend en entrée une grille et qui simule de façon correcte le phénomène de percolation.

Solution.

On propose le code suivant :

```
def percolation(grille):  
    # D'abord on traite la première ligne  
    for i in range(len(grille)):  
        if grille[0][i] == 0:  
            grille[0][i] = 2  
    # Puis on parcourt la grille jusqu'à ce qu'elle ne change plus  
    changement = True  
    while changement:  
        changement = False  
        for i in range(len(grille)):  
            for j in range(len(grille)):  
                if grille[i][j] == 0 and  
                    a_un_voisin_rempli(grille, i, j):  
                    grille[i][j] = 2  
                    changement = True
```

Remarque

On pourrait renvoyer la grille après percolation. Mais vu qu'on a modifié la grille passée en argument, il n'est pas nécessaire de la renvoyer.