Exercices sur les matrices

22 février 2024

1 Exercices simples sur les matrices

Chaque exercice consiste à réaliser une fonction qui permettra par la suite d'être appelée et également être testée via un test unitaire. Ainsi aucun de ces exercices nécessite d'imprimer ou de lire des informations via la console.

Pour chaque exercice, je vous demande d'essayer de réaliser:

1. hypothèses de départ
2. détail des variables utilisées
3. le pseudo-code
4. l'écriture du code dans un langage de programmation, de préférence Java car celui-ci permet d'utiliser des vecteurs (matrices) avec des types primitifs et éventuellement Python en utilisant les listes.

1.1 Renvoyez la valeur minimum d'une matrice d'entiers

Écrivez la fonction « int rechercheMin(int[][] m) » qui devra retourner l'entier le plus petit.

1.2 Écrivez une fonction qui vérifie si un élément existe dans la matrice

La fonction « boolean existeInMat(int[][] m, int valeur) » devra parcourir la matrice et renvoyé *true* si la valeur donnée existe dans la matrice.

$m1 =$

	0	*1*	*2*	*3*
0	9	-1	6	2
1	4	5	1	3
2	2	3	3	-6

$existeInMat(m1, -2) \Rightarrow false$
$existeInMat(m1, 3) \Rightarrow true$

1.3 Écrivez une fonction qui vérifie si une matrice est symétrique

« boolean estSymmetrique(int[][] m) »
Une matrice carrée est symétrique si $\forall i, j \rightarrow m_{i,j} = m_{j,i}$

- En entrée, vous avez une matrice carrée d'entiers (n*n)
- En sortie, un booléen qui renvoie *true* si la matrice est symétrique

Exemple:

$m1 =$

	0	*1*	*2*
0	9	-1	6
1	-1	5	1
2	6	1	3

est symétrique

1.4 Écrivez une fonction qui vérifie la validité d'une séquence de nombres

La fonction « boolean validerSeq(int[] v, boolean[][] m) »
le vecteur « v » contient une suite de chiffres (0..9)
la matrice carrée « m » de 10 sur 10 contient des booléens qui indique les transitions valables d'un chiffre vers un autre chiffre.
La fonction devra renvoyer un booléen pour indiquer que la séquence présente dans le vecteur respecte la matrice de transitions:
Exemple avec une matrice 3 sur 3 (pour simplifier)

$m1 =$

	0	*1*	*2*	*3*
0	true	false	true	true
1	false	true	false	true
2	true	false	true	false
3	true	true	false	false

Une transition de i vers j est valable si la case $m_{i,j} = true$.
0,0,2,2,0,3 est valable mais 0,2,1,1,3 ne l'est pas car la transition de 2 vers 1 n'est pas autorisée.

1.5 Produit de 2 matrices

Écrivez la fonction « int[][] produitMat(int[][]m1,int[][]m2) » qui renvoie une nouvelle matrice contenant le produit de m1*m2.
Voici la formule du produit de 2 matrices:

$m1 : matrice\ de\ dimension\ n * k$
$m2 : matrice\ de\ dimension\ k * m$
$m3 : matrice\ de\ dimension\ n * m$

$$\forall i : 1 \cdots n\ et\ \forall j : 1 \cdots m \quad m3_{i,j} = \sum_{l=1}^{k} m1_{il} * m2_{lj}$$

Exemple: m1*m2=m3

$$\begin{pmatrix} 1 & 2 \\ 5 & 3 \\ -1 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 11 & 10 \\ 34 & 22 \\ 7 & 14 \end{pmatrix}$$

2 Exercices plus difficiles

2.1 Vérifiez que chaque ligne, chaque colonne et que les 2 diagonales principales d'une matrice carrée, donnent la même somme

Astuces:

- utilisez 2 variables *dirX* et *dirY* qui pourront avoir une valeur de -1, 0 ou 1 pour désigner une direction (les incréments à faire pour avancer dans une direction)
- créez une fonction qui calcule la somme des cellules dans la direction donnée (dirX, dirY) à partir d'une position (posLigne, posColonne):

  « int calculeSomme(in[][]m, int posLigne, int posColonne, int dirX, int dirY) »
- créez la fonction principale que se chargera d'appeler la fonction *calculeSomme* le nombre de fois nécessaire, en ajustant la direction et la case initiale (posLigne, posColonne):

  « boolean sommeIdentique(int[][] m) »


Arrêtez le processus dès qu'une somme est différente!

Exemples de configuration	PosLigne	PosColonne	dirX	dirY
1ère ligne	0	0	0	1
2ème ligne	1	0	0	1
1ère colonne	0	0	1	0
2ème colonne	0	1	1	0
$1^{ère}$ diagonale principale	0	0	1	1
$2^{ème}$ diagonale principale	n	n	-1	-1

2.2 Vérifiez qu'une matrice carré est un carré magique

Cet exercice est identique au précédent à l'exception que les valeurs de la matrice doivent être des entiers >=0.
Un telle matrice s'appelle un carré magique.
« boolean estMagique(int[][] m) »

2.3 Vérifiez qu'une matrice carré est un carré magique normal

Cet exercice est identique au précédent avec comme contrainte supplémentaire que dans le cas d'une matrice n*n, les valeurs autorisées sont les entiers de 1 à n*n.
Un telle matrice s'appelle un carré magique normal.
« boolean estMagiqueNormal(int[][] m) »
Exemple où la somme de chaque ligne donne 15:

8	1	6
3	5	7
4	9	2

Astuce: créez un vecteur de booléens de taille (n*n) +1 qui sera par défaut à faux. Lorsque vous rencontrerez un nombre en parcourant la matrice, vérifiez si le booléen associé à celui-ci est à *false*, si tel est le cas cela signifie que vous n'avez pas encore rencontré cette valeur. Mettez ensuite le booléen en question à *true* pour indiquer que ce nombre a déjà été pris.

2.4 Construire un carré magique de taille impaire

« int[][] creeCarreMagiqueNormal(int taille) » , « taille » doit être un nombre impaire!
Construisez un carré magique de taille impaire sur base de la construction suivante :

- on va introduire les entiers de 1 à n*n
- le 1 se positionne au milieu de la première ligne
- on avance dans la direction NE (-1,+1) modulo n
- si la prochaine case est occupée, décaler d'une case vers le bas (0,-1) et poursuivez (en rouge sur l'exemple ci-dessous)

Exemple:

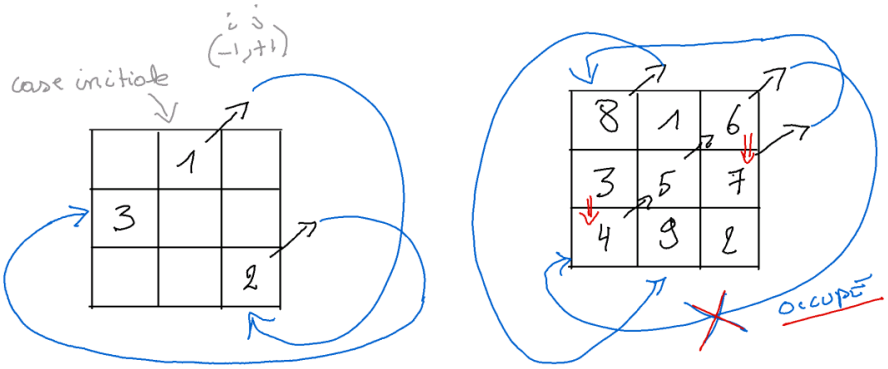
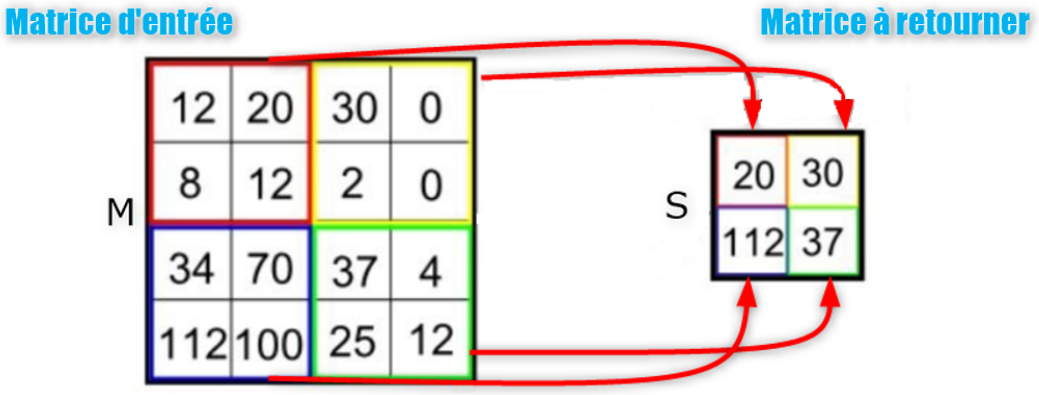


IMAGE 1 – construction

2.5 Écrivez une fonction qui retourne une convolution d'une matrice carrée

Exemple d'entête de la fonction en Java« public static int[][] convolution(int[][] m) »



A partir d'une matrice carrée d'entiers de taille **n*n** où **n est un multiple de 2** (matrice de gauche M). Renvoyez une nouvelle matrice carrée de taille n/2 (matrice de droite S). La valeur maximum de chaque carré de 2 sur 2 de la matrice de départ sera encodée dans la cellule correspondante dans la matrice de sortie S:

Le maximum du 1^{er} carré 2*2 de la matrice M | 12 | 20 | / | 8 | 12 | est 20 qui se retrouvera dans la cellule s(0,0), le maximum du $2^{ème}$carré | 34 | 70 | / | 112 | 100 | est 112 qui sera dans la cellule s(1,0), le maximum du carré | 30 | 0 | / | 2 | 0 | est 30 qui sera dans la cellule s(0,1),...

Comme indiqué la matrice « m » peut avoir une autre taille que 4*4.

2.6 Réalisez un contour des éléments d'une matrice

Considérons une matrice d'entiers de taille n*n où:

valeur	signification
0	case vide
1	cellule
2	contour que votre fonction doit rajouter

Écrivez la fonction « » qui va effectuer le contour des valeurs à 1 de la matrice:
Exemple:

$$\begin{matrix}0&0&0&0&0&0\\0&0&0&0&0&1\\0&0&1&0&0&0\\0&0&1&1&0&0\\0&0&0&0&0&0\end{matrix} \Longrightarrow \begin{matrix}0&0&0&0&2&2\\0&2&2&2&2&1\\0&2&1&2&2&2\\0&2&1&1&2&0\\0&2&2&2&2&0\end{matrix}$$

2.7 Exercices sur le jeu d'Othello

Dans le dépôt vous avez dans la package « jeu », une classe « Othello », cependant plusieurs de ses méthodes ne sont pas encore implémentées.

Voici les quelques méthodes à réaliser:

2.7.1 Affichez la matrice du jeu sur la console (sans la bordure)

Écrivez la fonction « afficheJeu ».

2.7.2 Retournez tous les pions dans une direction donnée si on prend des pièces en fourchette

Écrivez la fonction « retournePions » qui va, s'il existe une fourchette dans la direction donnée, retourner tous les pions de l'adversaire.

Elle renverra le nombre de pions retournés.

2.7.3 Créez une fonction « retournePions » qui va appeler la méthode précédente pour chaque direction

Écrivez la fonction « void retournePions(Position choixJ) » qui va appeler la fonction précédente « retournePions » et ceci pour chaque direction.

Elle devra aussi mettre à jour le nombre de pions blancs et noirs.

2.7.4 Créez la fonction « ajusteChoixJoueur » qui va mettre à jour l'ensemble « choixPossiblesJoueur »

Cette fonction appellera la fonction « checkDirection » pour chaque direction jusqu'à ce qu'il y a une fourchette dans une direction ou que l'on ai testé toutes les directions.

2.7.5 Créez la fonction « checkDirection » qui va vérifier s'il est possible de prendre en fourchette des pions adverses

Elle renverra « *true* » si une fourchette existe, « *false* » autrement.