Séance 5: Tableaux de tableaux

Université Paris Cité

1 Échauffements

Les trois exercices qui suivent permettent de nous familiariser avec la manipulation des tableaux de tableaux (aussi appelés « tableaux imbriqués »).

Exercice 1 (Affichage, ⋆)

Écrire la procédure `printArrayOfArrays` *qui attend un tableau de tableaux et affiche son contenu ligne par ligne*

> ***Contrat:***
> *Par exemple,* `printLines({{1, 2, 3}, {4, 5}, {6}})` *va afficher :*

```
1 1  2  3
2 4  5
3 6
```

□

Exercice 2 (Somme ligne par ligne 🔍, ⋆⋆)
Programmer `rowSums`*, la fonction qui attend un tableau de tableaux d'entiers et renvoie le tableau constitué des sommes de chaque ligne.*

> ***Contrat:***
> *Par exemple,* `rowSums({{2, 3}, {5, 8, 13, 21}, {34}})` *doit renvoyer* `{5, 47, 34}`.

□

Exercice 3 (Bonus : triangle de Pascal, ⋆⋆⋆)

Le triangle de Pascal apparaît en algèbre et en probabilités. Il est donné par une famille d'entiers $\binom{n}{k}$*, où* n *et* k *sont des entiers naturels tels que* $k \leq n$*.*
Le triangle de Pascal se calcule par $\binom{n}{n} = \binom{n}{0} = 1$ *et* $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$.

Programme la fonction `pascal`*, qui attend un entier* `sz` *et renvoie le tableau triangulaire d'entiers tels que le terme à la* `n`*-ème ligne et la* `k`*-ème colonne est* $\binom{\texttt{n}}{\texttt{k}}$*, pour* $\texttt{k} \leq \texttt{n} \leq \texttt{sz}$

> ***Contrat:***
> *Par exemple,* `pascal(4)` *va renvoyer* `{{1}, {1, 1}, {1, 2, 1}, {1, 3, 3, 1}, {1, 4, 6, 4, 1}}`.

□

2 Réseau social

Nous voulons dans cette section modéliser un réseau social dans lequel plusieurs utilisateurs ayant les numéros $0, 1, 2 \dots n$ sont inscrits. Pour ce faire, nous allons représenter le réseau par un tableau de tableaux d'entiers `R` où chaque case `R[i][j]` vaut :

- `1` si l'utilisateur `i` est ami avec l'utilisateur `j` ;
- `0` sinon.

Par exemple, un réseau `R` dans lequel il y a trois utilisateurs peut valoir le tableau de tableaux suivant :

```
{{0, 1, 0}, {1, 0, 0}, {0, 0, 0}}
```

où l'utilisateur 0 est ami avec l'utilisateur 1 (car `R[0][1]=1` et `R[1][0]=1`) et où l'utilisateur 2 n'est ami avec personne.

Notez qu'un utilisateur n'est jamais ami avec lui-même et que pour chaque couple d'utilisateurs `i` et `j` : si `i` est l'ami de `j` alors `j` est aussi l'ami de `i`.

Exercice 4 (Les fonctionnalités de base 🔍, $\star - \star\star$)

1. *Écrivez une fonction* `int[][] CreateGraph (int n)` *qui renvoie un réseau sous la forme de tableaux de tableaux comme vu ci-dessus. Il y aura dans ce tableau un nombre* `n` *d'utilisateurs et l'amitié entre deux utilisateurs est aléatoire. Vous pouvez pour cela utilisez la fonction* `randRange` *qui vous a été donnée au TP 3.*
2. *Écrivez une fonction* `int friends_nbr (int [] [] R, int a)` *qui pour un réseau* `R` *et un utilisateur* `a` *renvoie le nombre d'amis que possède* `a` *dans le réseau.*
3. *Écrivez une fonction* `int [] friends (int[][] R, int a)` *qui pour un réseau* `R` *et un utilisateur* `a` *renvoie un tableau contenant tous les amis de* `a` *dans le réseau et un tableau vide s'il n'en possède aucun.*
4. *Écrivez une fonction* `int[] popular (int[][] R)` *qui renvoie le tableau contenant les numéros des utilisateurs les plus populaires du réseau. Un utilisateur est populaire s'il possède le plus grand nombre d'amis dans le réseau.*
5. *Écrivez une fonction* `int[] common_friends (int[][] R,int a, int b)` *qui renvoie pour un réseau* `R` *et deux utilisateurs* `a` *et* `b` *un tableau contenant les amis qu'ils ont en commun.*
6. *Écrivez une fonction* `int[][] add_user(int[][] R,int[] t)` *qui ajoute un utilisateur dans le réseau* `R` *et le lie d'amitié avec les utilisateurs qui sont dans le tableau* `t` *. Cette fonction renvoie le nouveau réseau obtenu. Dans le nouveau réseau on aura donc un utilisateur en plus.*
7. *Considérons un tableau* `noms` *qui contient des chaines de caractères. Pour chaque indice* `i` *la case* `noms[i]` *contient le nom de l'utilisateur* `i`*. Par exemple, si* `noms` *vaut :*

```
noms = {"Evan Spiegel", "Mark Zuckerberg", "Jack Dorsey"}
```

Alors l'utilisateur numéro 1 est `noms[1]` *et donc* `"Mark Zuckerberg"`*. Quelle est la longueur de ce tableau ? Modifiez la fonction* `popular` *pour qu'elle renvoie les noms des utilisateurs les plus populaires au lieu de leurs numéros. Attention il faudra bien sûr changer aussi les arguments de la fonction.*

□

Exercice 5 (Événement 🔍, $\star\star\star$)

Dans cette partie on souhaite utiliser notre réseau social afin d'améliorer l'alchimie de votre pendaison de crémaillère à venir.

1. *Pour cela vous devez programmer une fonctionnalité* `int[] invite(int[][] R,int a)` *qui pour un utilisateur* `a` *renvoie tous ses amis sur le réseau mais aussi les amis de ses amis. Bien entendu cette fonction ne doit pas renvoyer* `a`.
2. *Certains sont stricts et veulent que chaque invité connaisse deux amis distincts de* `a`.
   Pour vous aider dans cette tâche on vous demandera de calculer le tableau de tableaux `P` *où chaque case de* `P` *est obtenue de* `R` *de la manière suivante :*

   $$\texttt{P[i][j]} = \Sigma_{k=0}^{n-1} \texttt{R[i][k] * R[k][j]}$$

   où n *est le nombre d'utilisateurs du réseau. Une fois* `P` *calculé, pour chaque couple d'utilisateurs* `i, j` *si* `P[i][j] = 2` *alors* `i` *connaît deux amis distincts de* `j`.
   Écrivez la fonction `int[] strict_invite(int[][] R, int a)` *qui renvoie le tableau d'invités selon les conditions strictes.*
3. *Modifiez les fonctions ainsi que la liste de leurs paramètres pour que les valeurs renvoyées soient les noms des utilisateurs au lieu de leurs numéros.*

□

3 2048

Dans cette section on implémente une version du jeu 2048, jouable par exemple à l'adresse https://play2048.co/. Vous pouvez l'utiliser pour observer les règles du jeu : on peut faire glisser les cases vers la gauche, la droite, le haut, ou le bas avec les flèches du clavier. Deux cases de valeur égale fusionnent, et à chaque tour une nouvelle case vide aléatoire est remplacée par un 2 (ou plus rarement un 4). Pour gagner, le but est d'obtenir une case contenant 2048.

Vous trouverez fourni un squelette de code à remplir.

Exercice 6 (Définitions préliminaires, $\star$)
Dans notre code, la grille sera représentée par la variable globale `board`, *contenant un tableau de tableaux carré, de côté de taille* `boardSize` *(qui sera dans notre cas égale à 4).*

Chaque case de la grille sera représentée par un entier, valant `0` *dans le cas d'une case vide, ou la valeur de la case pour une case non vide.*

1. *Écrire une procédure* `initBoard` *qui initialise* `board` *pour représenter une grille vide, à part pour les cases de position* $(1, 0)$ *et* $(3, 3)$, *qui doivent contenir des 2.*
2. *Écrire une fonction* `isBoardWinning`, *ne prenant pas d'argument et renvoyant un booléen, qui vérifie si la grille est une grille gagnante (i.e. elle contient une case de valeur 2048).*
3. *On aura besoin d'une fonction donnant la valeur des nouvelles cases insérées à chaque tour. Écrire une fonction* `newSquareValue`, *ne prenant pas d'argument, et renvoyant un 2 pour* 90% *du temps, et un 4 les* 10% *du temps restants.*
   On utilisera la fonction `randInt` *fournie.*

Après avoir implémenté ces 3 fonctions, le squelette devrait compiler, et afficher la grille telle qu'initialisée par `initBoard`. *Vérifier que tout fonctionne bien.* □

Exercice 7 (Décaler les cases : sur une ligne, $\star\star$ - $\star\star\star$)
L'étape suivante consiste à implémenter la logique pour décaler les cases vers la gauche, la droite, le haut ou le bas. Pour l'instant, on va seulement s'intéresser au problème de décaler les cases vers la gauche, et uniquement sur une ligne.

1. *Écrire une fonction auxiliaire* `newEmptyRow`, *ne prenant pas d'arguments, et renvoyant un tableau (une "ligne") de longueur* `boardSize` *initialisé avec des cases vides.*

2. *Écrire une fonction* `slideLeftNoMerge`, *prenant en argument un tableau de cases (représentées par des entiers comme décrit plus haut), que l'on suppose de longueur* `boardSize`. *La fonction doit renvoyer un nouveau tableau, également de longueur* `boardSize`, *où toutes les cases non vides ont été décalées vers la gauche et mises à la suite, sans cases vides intermédiaires.*

   Pour le moment on n'essaye pas de fusionner les cases de même valeur. On pourra utiliser `newEmptyRow` *pour créer et initialiser le tableau à renvoyer.*

   Contrat:
   ```
   int[] row = {0,2,0,2};
   printIntArray(slideLeftNoMerge(row))
   ```
   doit afficher
   ```
   2 2 0 0
   ```

3. *On implémente maintenant la fusion de cases. Implémenter une fonction* `slideLeftAndMerge`, *prenant en argument un tableau de cases, supposé de longueur* `boardSize`, *et pour lequel on suppose que les cases non vides ont été décalées vers la gauche (comme ce que fait* `slideLeftNoMerge`*).*

   La fonction doit renvoyer un nouveau tableau, également de longueur `boardSize`, *où :*
   — *pour deux cases consécutives de la même valeur dans le tableau d'entrée, une case avec leur somme est écrite dans le tableau renvoyé ;*
   — *pour les autres cases, elles sont recopiées telles quelles dans le tableau renvoyé.*

   Contrat:
   ```
   int[] row = {2,2,2,0};
   printIntArray(slideLeftAndMerge(row))
   ```
   doit afficher
   ```
   4 2 0 0
   ```

   Contrat:
   ```
   int[] row = {2,2,2,2};
   printIntArray(slideLeftAndMerge(row))
   ```
   doit afficher
   ```
   4 4 0 0
   ```

   Contrat:
   ```
   int[] row = {4,2,2,0};
   printIntArray(slideLeftAndMerge(row))
   ```
   doit afficher
   ```
   4 4 0 0
   ```

   Indice : utiliser une boucle `while`, *et deux indices, l'un indiquant où lire dans le tableau en entrée, et l'autre indiquant où écrire dans le tableau en sortie.*

4. *Combiner les deux fonctions définies précédemment, et écrire une fonction* `slideRowLeft`, *prenant en argument un tableau de longueur* `boardSize`, *et retournant un nouveau tableau, où les cases ont été décalées vers la gauche, avec fusion des cases adjacentes de même valeur.*

□

Exercice 8 (Décaler les cases : sur toute la grille, ⋆⋆ - ⋆⋆⋆)

1. *En utilisant* `slideRowLeft`, *implémenter une procédure* `slideBoardLeft` *ne prenant aucun argument, et mettant à jour* `board` *de sorte que les cases de toutes les lignes de la grille soient décalées à gauche.*
2. *En utilisant* `slideRowLeft`, *et la procédure* `reverse` *(dont la spécification et l'implémentation sont données dans le squelette), implémenter une procédure* `slideBoardRight`, *qui décale vers la droite les cases de toutes les lignes de la grille.*

   Indice : on remarquera que décaler vers la droite les cases d'un tableau, c'est équivalent à inverser l'ordre du tableau, décaler vers la gauche, puis ré-inverser l'ordre du tableau.

3. *En utilisant* `slideBoardLeft` *et* `slideBoardRight`*, ainsi que* `transpose` *(dont la spécification et l'implémentation sont données dans le squelette), implémenter deux procédures* `slideBoardUp` *et* `slideBoardDown`*. Celles-ci doivent respectivement décaler vers le haut ou vers le bas les cases de la grille.*

   Indice : on remarquera par exemple que décaler vers le haut est équivalent à transposer la grille, décaler vers la gauche puis re-transposer...

4. *En utilisant les fonctions implémentées aux questions précédentes, compléter le code de la procédure* `slideBoard`*, qui prend en argument une direction (à comparer avec les directions pré-définies dans le squelette :* `LEFT`, `RIGHT`, `UP` *et* `DOWN`*), et décale les cases de façon correspondante.*

À ce point là, vous pouvez tester que les décalages se font conformément à ce qui est attendu, en utilisant les flèches du clavier. □

Exercice 9 (Nouveaux carrés, ⋆)

Pour l'instant, le jeu n'est pas très intéressant car il n'y a pas de production de nouveaux carrés.

1. *La fonction* `addSquare` *est appelée après chaque coup, et doit insérer un nouveau carré de la valeur fournie, à une case vide de la grille. Implémenter* `addSquare`*, prenant en argument la valeur du carré à insérer, et écrivant cette valeur dans la première case vide de la grille.*
2. *(Bonus) Au lieu d'insérer cette valeur dans la première case vide, l'insérer dans une case vide choisie aléatoirement.*

Le jeu devrait maintenant être jouable, et relativement fidèle à l'original ! □

Exercice 10 (Bonus, ⋆⋆)

1. *Actuellement, le joueur est autorisé à effectuer un coup qui ne change rien à l'état du jeu (par exemple, "droite" alors que toutes les cases sont déjà alignées à droite et ne peuvent être fusionnées). La conséquence est que le joueur peut facilement produire des nouvelles cases sans devoir décaler celles existantes dans une direction potentiellement désavantageuse.*

   Pour rendre le jeu plus difficile, on veut n'autoriser que les coups qui changent l'état de la grille. Complétez la fonction `isValidMove`*, afin que celle-ci renvoie un booléen indiquant si la direction passée en argument correspond à un coup qui changerait l'état de la grille.*

2. *Les exercices 2 et 3 sont guidés de manière à implémenter ce que l'on veut d'un manière la plus simple possible, mais pas la plus efficace. Optimiser le code déjà écrit pour minimiser le nombre de tableaux intermédiaires alloués, et le nombre de fois où la grille est parcourue.*

   En étant malin, il est possible de n'allouer aucun tableau intermédiaire, et de ne parcourir la grille qu'une fois par appel à `slideBoard` *!*

□