Introduction à la Programmation 1 – Travaux Pratiques

Séance 10b: 2048

Université Paris-Diderot

Objectifs:

- Implémenter et utiliser des fonctions étant donnée une spécification
- Manipuler des listes de listes
- Apprendre à utiliser les variables globales.

Dans ce TP on implémente une version du jeu 2048, jouable par exemple à l'adresse https://gabrielecirulli.github.io/2048/. 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 1 (Définitions préliminaires, $\star$)
Dans notre code, la grille sera représentée par la variable globale `board`*, contenant une liste de listes carrée, 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.* `randint(a, b)` *renvoie un entier aléatoire entre* `a` *et* `b` *inclus.*

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

Exercice 2 (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 une liste (une "ligne") de longueur* `boardSize` *initialisée avec des cases vides.*
2. *Écrire une fonction* `slideLeftNoMerge`*, prenant en argument une liste de cases (représentées par des entiers comme décrit plus haut), que l'on suppose de longueur* `boardSize`*. La fonction doit retourner*

une nouvelle liste, 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 la liste à retourner.*

Contrat:

```
row = [0,2,0,2]
print(slideLeftNoMerge(row))
```

doit afficher

```
[2, 2, 0, 0]
```

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

   La fonction doit retourner une nouvelle liste, également de longueur `boardSize`*, où :*
   - *pour deux cases consécutives de la même valeur dans la liste d'entrée, une case avec leur somme est écrite dans la liste retournée ;*
   - *pour les autres cases, elles sont recopiées telles quelles dans la liste retournée.*

   Contrat:

   ```
   row = [2,2,2,0]
   print(slideLeftAndMerge(row))
   ```

   doit afficher

   ```
   [4, 2, 0, 0]
   ```

   Contrat:

   ```
   row = [2,2,2,2]
   print(slideLeftAndMerge(row))
   ```

   doit afficher

   ```
   [4, 4, 0, 0]
   ```

   Contrat:

   ```
   row = [4,2,2,0]
   print(slideLeftAndMerge(row))
   ```

   doit afficher

   ```
   [4, 4, 0, 0]
   ```

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

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

□

Exercice 3 (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 vers la gauche.*
2. *En utilisant* `slideRowLeft`*, et la procédure* `reverse` *(dont la spécification et l'implémentation sont données dans le squelette), implémenter une procédure* `slideBoardRight`*, ne prenant aucun argument, 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'une ligne, c'est équivalent à inverser l'ordre de la ligne, décaler vers la gauche, puis ré-inverser l'ordre de la ligne.
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, et ne prennent aucun argument.*

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 4 (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 5 (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 listes intermédiaires créées, et le nombre de fois où la grille est parcourue.*

   En étant malin, il est possible de ne créer aucune liste intermédiaire, et de ne parcourir la grille qu'une fois par appel à `slideBoard` *!*

□