Université de Lille 1 SCIENCES ET TECHNOLOGIES

ufr d'IEEA
Formations en
Informatique de
Lille 1

L2 Informatique - 2015–2016

Algorithmes et Programmation 2

DS2 - 05/12/2015

durée 2h - documents et calculatrices non autorisés

Ce devoir contient trois exercices indépendants. Sauf mention du contraire, on ne vous demande pas la documentation des fonctions que vous réaliserez.

Sauf mention explicite du contraire, les listes et piles que vous utiliserez dans ce devoir sont celles étudiées en cours. Vous devez donc utiliser les opérations primitives qui les accompagnent.

Exercice 1 *Zip de deux listes*

« *Zipper* »[1] deux listes ℓ_1 et ℓ_2 c'est construire une liste de même longueur que la longueur supposée commune de ℓ_1 et ℓ_2 dont les éléments sont des couples dont la première composante est dans ℓ_1 et la seconde dans ℓ_2.

Question 1 Réalisez une fonction nommée `zip` qui prend deux listes de même longueur en paramètre et les *zippe*.

En voici un exemple d'utilisation :

```
>>> l1 = native2list ([1, 3, 5, 7])
>>> l2 = native2list (['Timoleon', 'Calbuth', 'Talon', 'Carmen'])
>>> l = zip (l1,l2)
>>> list2native (l)
[(1, 'Timoleon'), (3, 'Calbuth'), (5, 'Talon'), (7, 'Carmen')]
```

Question 2 Réalisez la fonction réciproque que vous nommerez `unzip`.

```
>>> l1,l2 = unzip (l)
>>> list2native (l1)
[1, 3, 5, 7]
>>> list2native (l2)
['Timoleon', 'Calbuth', 'Talon', 'Carmen']
```

Exercice 2 *Fusion de deux piles*

Question 1 Réalisez une fonction nommée `stack_merge` paramétrée par deux piles triées dans l'ordre croissant (du sommet vers le fond) et qui produit une liste triée dans l'ordre croissant des éléments contenus dans ces deux piles.

```
>>> p1 = stack.create ()
>>> stack.push (11, p1)
>>> stack.push (5, p1)
>>> stack.push (1, p1)
>>> p2 = stack.create ()
>>> stack.push (9, p2)
```

1. Rien à voir avec la compression des données, mais plutôt avec les fermetures éclairs.

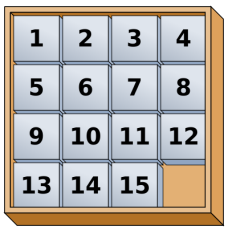
```
>>> stack.push (7, p2)
>>> list2native (stack_merge (p1,p2))
[1, 5, 7, 9, 11]
```

Question 2 Votre fonction a–t–elle un effet de bord ?

Question 3 Reprendre la première question en supposant les deux piles triées dans l'ordre croissant du fond vers le sommet. La liste à construire doit toujours être triée dans l'ordre croissant.

Exercice 3 *Jeu du taquin*

Le *taquin* est un puzzle qui,classiquement, se présente sous la forme de 15 petits carreaux numérotés de 1 à 15 disposés dans un cadre 4×4 de sorte qu'une case est vide. Le puzzle consiste à mettre les carreaux dans l'ordre donné dans la figure de droite à partir d'une configuration initiale quelconque en opérant une série de déplacements horozontaux ou verticaux de la case vide.



Taquin 4×4 dans sa configuration à atteindre (source Wikipedia)

Un module `taquin` a été conçu pour gérer ce puzzle. Il définit un certain nombre de fonctions.

— Une fonction `create` paramétrée par un entier n. Cette fonction renvoie un puzzle $n \times n$ à résoudre.

```
>>> taq = taquin.create (4)
```

— Une fonction `get_frame` qui renvoie la disposition des pièces sous forme d'une liste de nombres.

```
>>> taquin.get_frame (taq)
[2, 6, 16, 4, 1, 3, 5, 8, 9, 10, 11, 12, 13, 7, 14, 15]
```

— Une fonction `print_frame` permettant d'imprimer le puzzle.

```
>>> taquin.print_frame (taq)
+----+----+----+----+
|  2 |  6 |    |  4 |
+----+----+----+----+
|  1 |  3 |  5 |  8 |
+----+----+----+----+
|  9 | 10 | 11 | 12 |
+----+----+----+----+
| 13 |  7 | 14 | 15 |
+----+----+----+----+
```

— Un dictionnaire `MOVES` associant à chacune des quatre lettres `U`, `D`, `R` et `L` une fonction paramétrée par un taquin qui renvoie la configuration (frame) obtenue en déplaçant la case vide dans l'une des quatre directions (lorsque le déplacement est impossible, ces fonctions renvoient la configuration (frame) du taquin inchangée).

```
MOVES = {'U' : up, 'D' : down, 'L' : left, 'R' : right}
```

— Un modificateur de la disposition des carreaux : `set_frame` paramétrée par un taquin et une configuration qui remplace la configuration du taquin par celle passée en paramètre. Cette fonction ne renvoie rien (`None`).

```
>>> taquin.set_frame (taq, taquin.MOVES['U'] (taq))
>>> taquin.print_frame (taq)
```

```
+----+----+----+----+
|  2 |  6 |    |  4 |
+----+----+----+----+
|  1 |  3 |  5 |  8 |
+----+----+----+----+
|  9 | 10 | 11 | 12 |
+----+----+----+----+
| 13 |  7 | 14 | 15 |
+----+----+----+----+
```

— Un prédicat déterminant si le puzzle est résolu.

```
>>> taquin.is_solved (taq)
False
```

À titre d'exemple, voici une fonction permettant de jouer au taquin écrite à l'aide de ce module ainsi qu'une fonction `read_move` qui renvoie l'un des quatre caractères `U`, `D`, `L` ou `R`.

```
def main (n):
    taq = taquin.create (n)
    taquin.print_frame (taq)
    while not taquin.is_solved (taq):
        move = read_move ()
        taquin.set_frame (taq, MOVES[move] (taq))
        taquin.print_frame (taq)
    print ('Puzzle solved.')
```

Le but de cet exercice consiste à ajouter une fonctionnalité permettant au joueur d'abandonner la résolution du puzzle et obtenir du programme qu'il lui fournisse une solution présentée sous la forme d'une chaîne de caractères construite avec les quatre lettres `UDLR`, donnant la suite des déplacements de la case vide à effectuer pour parvenir à la configuration ordonnée.

Première version Une façon simple de proposer une solution du puzzle au joueur consiste à mémoriser

1. tous les déplacements de la case vide effectués par `create` pour mélanger le puzzle,
2. ainsi que tous les déplacements que le joueur a effectués dans sa recherche d'une solution.

Il suffit alors de donner les déplacements opposés de tous ces déplacements dans l'ordre inverse.

Question 1 Quelle structure de données permet aisément de répondre à ce besoin ?

Vous justifierez votre proposition et vous l'illustrerez sur un exemple de trois déplacements successifs.

Question 2 Réalisez une fonction nommée `solve` paramétrée par la structure contenant les déplacements qui renvoie une chaîne de caractères décrivant les déplacements à effectuer pour résoudre le puzzle.

Pour la question qui suit, vous supposerez que

— la fonction `taquin.create` renvoie un couple composé d'un taquin, et de la structure mémorisant tous les déplacements effectués lors du mélange ;

— la fonction `read_move` renvoie un caractère qui, en plus des quatre `UDLR`, peut être un `A` pour signifier l'abandon du joueur.

Question 3 Modifiez la procédure `main` de sorte que le joueur puisse abandonner, et dans ce cas imprimer une suite de mouvements aboutissant à la résolution du puzzle.

Deuxième version La solution proposée ci-dessus peut ne pas être optimale, parce qu'elle amène le puzzle plusieurs fois dans la même configuration. Par exemple, si la configuration dans laquelle abandonne le joueur est

```
+----+----+----+----+
|  1 |  2 |  3 |  4 |
+----+----+----+----+
|  5 |  6 |  7 |  8 |
+----+----+----+----+
|  9 | 10 | 11 |    |
+----+----+----+----+
| 13 | 12 | 14 | 15 |
+----+----+----+----+
```

il se peut que la solution proposée par la procédure précédente soit `DLURDLURDLURDLLURDRULLDRR`. Mais on peut constater qu'après avoir effectué les douze premiers déplacements, on est revenu à la configuration de départ. Ces douze premiers déplacements sont donc inutiles dans la résolution du puzzle.

Question 4 Comment modifier la gestion de la structure de données, au besoin en s'aidant d'une autre structure de données, pour raccourcir les solutions proposées en évitant de passer plusieurs fois par la même configuration du puzzle ?

Troisième version (Partie facultative hors barême)

Dans cette version, on abandonne totalement l'idée de mémoriser tous les déplacements.

Question 5 Proposez une version récursive de `solve`.

Vous pourrez supposer qu'une fonction `copy` paramétrée par un taquin et renvoyant une copie de ce taquin est définie dans le module `taquin`.
