Exercices de Programmation & Algorithmique 1

Série 6 – Les listes - le jeu du Mikado
(27 octobre 2022)

Département d'Informatique – Faculté des Sciences – UMONS

Pré-requis : Listes et récursivité (cours jusqu'au **Chapitre 7** compris).

Objectifs : Approfondir l'usage des listes; appliquer la récursivité aux listes.

---

1 Le contrat

1.1 Le jeu du Mikado

Le jeu du Mikado est un jeu d'adresse qui se compose de baguettes longues d'environ 20cm et effilées aux extrémités. Ces baguettes sont disposées aléatoirement de sorte que certaines d'entre elles s'enchevêtrent. L'objectif d'une partie de Mikado est de réussir à retirer une à une les différentes baguettes sans faire bouger les autres.

Il vous est demandé d'implémenter certaines parties d'un jeu de Mikado. Dans le cadre de ce contrat, nous considérons qu'un Mikado est composé de N baguettes numérotées distinctement. Afin de représenter les enchevêtrements de baguettes, une liste de couples est utilisée. Un couple (i, j) de cette liste indique que la baguette numérotée j repose sur la baguette numérotée i. Afin de réussir une partie de Mikado ayant dans sa liste d'enchevêtrements un tel couple (i, j), il faut que la baguette numérotée j soit retirée avant la baguette numérotée i. À tout instant de la partie, l'état du Mikado est représenté à l'aide d'une liste `bag` des baguettes restantes ainsi que d'une liste d'enchevêtrements.

Supposons, à titre d'exemple, une partie de Mikado ayant $N = 4$ baguettes numérotées de 0 à 3. La liste `bag` est initialement égale à `[0, 1, 2, 3]`. Une liste d'enchevêtrements pourrait être `[(0, 1), (0, 2), (3, 2)]`. Cette liste indique que pour retirer la baguette numérotée 0, il faut d'abord retirer les baguettes numérotées 1 et 2. Pour retirer la baguette numérotée 3, il faut d'abord retirer la baguette numérotée 2. Ainsi, un des ordres possibles pour réussir cette partie de Mikado pourrait être `[2, 3, 1, 0]` indiquant que les baguettes sont retirées dans l'ordre suivant : 2, 3, 1 et enfin 0.

1.2 À réaliser sur papier

Les effets de bord sont généralement des propriétés indésirables dans les fonctions. Il s'agit de comportements qui ne sont pas explicitement demandés par la fonction et qui ne sont souvent ni intuitifs, ni documentés. Par exemple, il est attendu que la fonction suivante prenne une liste d'au moins deux éléments et inverse les deux premiers. Le bout de code suivant affiche alors le résultat à l'écran. Pourtant, cette fonction possède plusieurs effets de bord.

```
ma_liste = [1, 4]

def inverser(une_liste):
  copie_liste = une_liste
  copie_liste[0], copie_liste[1] = copie_liste[1], copie_liste[0]
  print(copie_liste)

inverser(ma_liste)
```

1 Modifiez le code ci-dessus de sorte à supprimer l'ensemble des effets de bord de la fonction.

Effectuez l'analyse des fonctions suivantes. Indiquez, pour chaque fonction, les entrées et les sorties de ces dernières. Pour les fonctions récursives, identifiez le cas de base et l'hypothèse de récurrence. Notez qu'aucune de ces fonctions ne peut présenter d'effet de bord.

2 Une fonction booléenne `peut_retirer(i, bag, jeu)` qui retourne la valeur booléenne `True` si et seulement s'il est possible de retirer la baguette numérotée i sans toucher une des autres baguettes. Le paramètre `jeu` correspond à la liste des couples représentant les enchevêtrements de baguettes. Le paramètre `bag` correspond à la liste des baguettes restantes.

3 Une fonction booléenne **récursive** `est_jouable(bag, jeu)` qui retourne la valeur booléenne `True` si et seulement s'il existe un ordre dans lequel il est possible de retirer chacune des baguettes pour réussir une partie.

1.3 À réaliser sur machine

Implémentez les fonctions suivantes dans l'ordre :

4 Une fonction `creer_enchevetrements(bag, i, max)` qui retourne une liste d'enchevêtrements différents impliquant la baguette i. Le nombre d'enchevêtrements est choisi aléatoirement entre 0 et la valeur de `max_enchevetrements`. Les différents enchevêtrements impliquant la baguette i sont de la forme (x, i) où x est l'une des autres baguettes (choisie au hasard) présente dans `bag`.

5 Une fonction `creer_mikado(bag)` qui retourne une liste d'enchevêtrements. Ces enchevêtrements sont générés à l'aide de la fonction `creer_enchevetrements`. Pour chaque baguette, il y aura au maximum `|bag|` enchevêtrements.

6 La fonction `peut_retirer(i, bag, jeu)`. Cette fonction ne peut pas avoir d'effet de bord.

7 Une fonction **récursive** `quel_ordre(bag, jeu)` qui retourne un des ordres possibles dans lequel les baguettes peuvent être retirées pour réussir une partie. Si aucun ordre n'est possible, cette fonction retourne la valeur `None`. Cette fonction ne peut pas avoir d'effet de bord. Vous pouvez vous inspirer du code de la fonction `est_jouable` (exercice supplémentaire) afin de vous aider pour la réalisation de `quel_ordre`.

2 Exercices supplémentaires

Des exercices supplémentaires sur les listes sont également disponibles dans la Série 5.

★★☆ 8 Écrivez le code de la fonction `est_jouable(bag, jeu)`. Cette fonction ne peut pas avoir d'effet de bord.

★☆☆ 9 Écrivez une fonction `enumeration(seq)` qui, étant donné une séquence `seq` d'éléments, retourne une liste de couples dont le premier membre est un compteur (débutant à 0) et le second membre est un élément de la séquence donnée : `[(0, seq[0]), (1, seq[1]), ...]`.

★☆☆ 10 Écrivez une fonction `groupe(*seq)` prenant en paramètre un nombre variable de séquences. Cette fonction retourne une liste dont chaque élément est un n-uple dont chaque membre est un élément d'une des séquences données en paramètre : `[(seq[0][0], seq[1][0], ...), (seq[0][1], seq[1][1], ...), (seq[0][2], seq[1][2], ...), ... ]`. Le nombre d'éléments de la liste résultante est égal au nombre d'éléments de la plus petite des séquences.

★★☆ 11 Écrivez le code des fonctions suivantes :

(a) Une fonction **récursive** `somme(seq)` prenant en paramètre une liste de nombres réels. Cette fonction retourne la somme de ces nombres.

(b) Une fonction **récursive terminale** effectuant le même travail. Une fonction récursive terminale (ou finale) est une fonction dans laquelle l'appel récursif est toujours effectué en dernier dans la fonction et cette dernière instruction ne peut comporter que l'appel récursif.

★★☆ 12 Le jeu de la vie, conçu en 1970 par John Horton Conway, n'est pas un jeu à proprement parler. Il s'agit d'un automate cellulaire, un système dans lequel chaque état (cellule) est construit à partir de l'état précédent en suivant des règles strictes prédéfinies.

Le jeu de la vie repose sur une grille de longueur définie $n \times m$ dont les cases, les cellules par analogie aux cellules d'un organisme vivant, peuvent être soit vivantes, soit mortes.

À chaque étape, l'état d'une cellule est défini par l'état de ses 8 voisins directs :
— Une cellule morte possédant exactement 3 voisins vivants devient vivante.
— Une cellule vivante possédant 2 ou 3 voisins vivants reste vivante, sinon elle meurt.

En fonction de la configuration initiale, des motifs peuvent apparaître au fil de l'évolution de la grille. Certains de ces motifs se répètent, d'autres se transforment et se déplacent.

Il vous est demandé d'implémenter le jeu de la vie en suivant les consignes suivantes :

(a) Créez une fonction `next_state(grid, i, j)` qui, en fonction de la grille `grid` et d'une position définie par i et par j retourne Vrai si la cellule à cette position devient (ou reste) vivante, Faux sinon.

(b) Créez une fonction `next_grid(grid)` qui, en fonction de la grille `grid` correspondant à l'étape précédente, calcule une nouvelle grille de mêmes dimensions en suivant les règles établies.

(c) Créez une fonction `display_grid(grid)` qui affiche proprement à l'écran la grille passée en paramètre.

(d) Enfin, créez une fonction `simulate_life(grid, t)` qui, sur base d'une grille `grid` calcule et retourne une nouvelle grille obtenue après `t` itérations du jeu de la vie.

Testez ensuite le jeu de la vie en fournissant une grille initiale et, pour chaque étape, en affichant à l'écran le résultat d'une évolution. Conseil : vous pouvez utiliser l'instruction `input()` afin d'obliger l'utilisateur à appuyer sur `[Enter]` avant de passer à l'étape suivante, et donc de pouvoir suivre dans des conditions normales le développement de votre grille à l'écran. Vous pouvez également utiliser la fonction `sleep` du module `time`. Celle-ci prend un nombre en paramètre représentant le nombre de secondes durant lequel le programme se met en pause.

★★☆ 13 Le jeu du Morpion est un jeu dans lequel s'opposent deux joueurs. Le but dans ce jeu est d'aligner n fois un même jeton dans la grille de taille $n \times n$ de départ. Il vous est demandé d'implémenter une version du Morpion en suivant les consignes suivantes :

(a) Créez une fonction `create_morpion(n)` qui crée et retourne une grille de taille $n \times n$ sur laquelle les joueurs seront amenés à jouer.

(b) Créez une fonction `next_turn(morpion, player, i, j)` qui simule l'ajout d'un jeton en position i, j par le joueur `player` (1 ou 2). Cette fonction retourne une nouvelle grille correspondant à l'ajout du jeton sur la grille `morpion` passée en paramètre. Si le coup joué n'est pas valide (mauvaises positions, position déjà complète, etc.), cette fonction retourne `None`.

(c) Créez une fonction `who_wins(morpion)` qui retourne -1 si la partie n'est pas terminée, 0 si la grille correspond à une situation de match nul, 1 si le joueur 1 a remporté la grille, 2 pour le joueur 2. Un joueur emporte la partie s'il y a une ligne, une colonne ou une (des deux) diagonale qui ne contient que des jetons de ce joueur.

(d) Créez une fonction `play_morpion()` qui demande à l'utilisateur la taille de la grille, et propose à deux joueurs d'effectuer une partie du Morpion.

★★☆ 14 Le jeu du Mastermind est un jeu dans lequel il faut deviner une combinaison ordonnée de 4 jetons de couleurs. 4 couleurs différentes sont disponibles, pour les explications nous dirons qu'il y a des jetons rouges, bleus, verts et jaunes. Au début de la partie, l'ordinateur définit une combinaison secrète. Il s'agit d'une combinaison ordonnée de 4 couleurs choisies aléatoirement parmis les 4 couleurs disponibles. Il se peut qu'une même couleur apparaisse plusieurs fois dans la combinaison. Voici un exemple de combinaison secrète : `rouge - vert - vert - jaune`.

Une fois la combinaison secrète établie, la partie peut commencer. Le but du jeu est de deviner cette combinaison en un minimum d'étapes. A chaque étape, le joueur entre une combinaison ordonnée de 4 couleurs. Le jeu compare la combinaison entrée avec la combinaison secrète et donne au joueur deux informations : le nombre de couleurs qui sont bien positionnées, ainsi que le nombre de couleurs qui sont mal positionnées mais présentes dans la combinaison à deviner.

Pour faciliter les choses, on peut remplacer les couleurs par des numéros. Par exemple, associer 1 à rouge, 2 à bleu, etc. Dans la variante du jeu présentée ci-dessus, il y 4 couleurs différentes, et les combinaisons sont de longueur 4. Néanmoins, d'autres variantes de ce jeu existent dans lesquelles il y a par exemple 6 couleurs différentes, et des combinaisons de longueur 5.

Ecrivez une fonction `mastermind(secret, combi)` qui prend en paramètre une liste `secret` de numéros correspondant à la combinaison secrète et une liste `combi` de numéros correspondant à une combinaison tentée. Cette fonction retourne un couple (x, y) où x est le nombre d'éléments de `combi` qui sont présents et bien placés dans `secret`, et y est le nombre d'éléments de `combi` qui sont également présents dans `secret` mais pas à la bonne position.

Tests :

```
mastermind([1,2,2,3], [1,3,4,2])  →  (1,2)
mastermind([1,1,1,2], [2,1,1,1])  →  (2,2)
mastermind([3,5,5,4], [3,3,4,4])  →  (2,0)
```

★★★ 15 (Suite du Mikado) Une fonction **récursive** `quelsOrdres(bag, jeu)` qui retourne une liste contenant tous les ordres possibles dans lesquels les baguettes peuvent être retirées pour réussir une partie.

★★★ 16 Une machine à billets distribue des billets de \$3 et \$7. Elle peut distribuer n'importe quelle valeur $n \geq n_0$ en billets de \$3 et \$7.

(a) Déterminez la constante n_0 la plus petite possible.

(b) Pour la constante n_0 proposée, donnez deux algorithmes récursifs différents qui, étant donné $n \geq n_0$, calculent le nombre de billets de \$3 et \$7 permettant de réaliser la valeur n. Le premier algorithme se basera sur une récurrence faible, tandis que le second se basera sur une récurrence forte.

Exam 17 (20 novembre 2009) Problème 3 : Ordonnanceur de tâches.

Exam 18 (11 juin 2010) Problème 4 : Jeu du Mikado.

Exam 19 (30 août 2010) Problème 1 : Déplacement dans une maison.