Fiche 4.0 : récursivité

BUT Informatique
IUT de Vélizy

0 Empilements

☐ 0. **Prise en main - `empilements0`.** Lisez le code et exécutez-le. Cet exercice utilise une mini-bibliothèque `libEmpilements` qui étend `tkiteasy` ; les exercices qui suivent seront faits sur le même modèle. L'objectif de cette question est seulement de comprendre l'utilisation de la méthode `empilerCube`. Modifiez librement le script en ajoutant des cubes supplémentaires empilés avec les premiers ou sur des espaces vides.

☐ 1. **Trois fonctions - `empilements1`.** Ce script définit trois fonctions qui s'appellent entre elles. L'objectif ici est de comprendre en regardant le code pourquoi les cubes tombent exactement dans cet ordre et sur ces positions.

☐ 2. **Sans tricher ! - `empilements2`.** Si vous avez compris, essayez de prévoir dans quel ordre et à quels endroits vont tomber les cubes si l'on fait l'appel de fonction `fa(g,2)` (il faut l'ajouter). Prenez un papier et essayez de noter dans quel ordre tombent les cubes, sur quel emplacement et avec quelle couleur ! Si vous vous êtes trompé, ce n'est pas grave mais essayez de comprendre pourquoi !

☐ 3. **Cette fois-ci, vraiment sans tricher ! - `empilements3`.** Même exercice si l'on ajoute dans ce script l'appel de fonction `fA(g,2)`.

☐ 4. **Chassé-croisé - `empilements4`.** Essayez de prévoir ce que va faire l'appel `f0(g,0)`.

☐ 5. **suite - `empilements4`.** A la fin de la fonction `f0`, ajoutez l'empilement d'un cube rouge sur la même position que le cube bleu. Comment et dans quel ordre cela va-t'il se passer ?

Dans l'exercice précédent, vous avez vu deux fonctions `f0` et `f1` qui s'appellent l'une l'autre. Il ne nous reste plus beaucoup de chemin à faire pour utiliser maintenant une fonction qui s'appelle elle-même : une fonction *récursive*.
Attention : dans les exercices ci-dessous, celle ou celui qui écrit une boucle va au coin !

☐ 6. **Première fonction récursive - `empilements5`.** La fonction qui effectue le travail d'empilement de cubes est la fonction `f0`. Comme vous le voyez cette fonction ne contient pas de boucle. La ligne où la fonction `f0` appelle la fonction `f0` dans son propre code est *l'appel récursif*.

☐ 7. **Suite - `empilements5`.** Déplacez l'appel récursif *avant* l'empilement. Que se passe-til ? Expliquez !

☐ 8. **Suite - `empilements5`.** Modifiez le code afin d'empiler un cube bleu de gauche à droite, puis un cube vert de droite à gauche, avec un seul appel récursif !

☐ 9. **A vous - `empilements6`.** Sur le modèle de l'exercice précédent, écrivez une fonction récursive mais qui fasse une récurrence descendante, de la position 9 à la position 0 en empilant des cubes. Puis modifiez le code afin d'obtenir un empilement de droite à gauche, suivi d'une seconde ligne de gauche à droite (toujours un seul appel récursif.)

☐ 10. **remplissage - `empilements7`** La fonction `f0` de l'exercice précédent, dans sa première version permettait de remplir toute une ligne. Pouvez-vous créer une deuxième fonction `f1` qui va appeler `f0` pour remplir une ligne, puis qui va s'appeler elle-même et ainsi du suite afin de remplir les lignes du-dessus ? Si cela fonctionne, tout l'écran sera rempli par récurrence. Attention, il faut s'arrêter à un moment quand les lignes deviennent trop hautes (vous pouvez avoir besoin d'un compteur passé en paramètre). Attention, dans cet exercice, toujours pas de boucle !

1 Exploration-Diffusion

Dans ce nouvel environnement graphique, nous allons étudier le concept de diffusion ou d'exploration, qui consiste en général à partir d'un endroit précis et d'explorer de proche en proche tous les endroits accessibles.

☐ 11. **démonstration -** `demo_grille1.mp4` visualiser le film `demo_grille1.mp4` afin de voir concrètement l'objectif de cette partie.

Il s'agit d'une version préliminaire des algorithmes de plus court chemin. Dans cette version, Pacman se trouve dans un environnement 2D où il peut se déplacer dans 4 directions, et ne peut pas traverser les murs (cases noires). L'objectif est de déterminer l'ensemble des cases auxquelles Pacman peut accéder, en visualisant de plus les appels récursifs par des flèches.

☐ 12. **tuto de l'API -** `grille0.` lisez le code de `grille0.py` et exécutez. Attention surveillez aussi le terminal. Le but est de comprendre l'interface qui sera utilisée ensuite.

☐ 13. **première tentative -** `grille1` Même chose, sauf qu'ici on fait se déplacer pacman jusqu'à ce qu'il soit coincé. Essayez de modifier ce code pour que tout se passe comme dans le film `demo_grille2.py`.

— lorsque Pacman se déplace, on marque son déplacement avec une flèche rouge.
— lorsque Pacman rencontre un obstacle, il essaie de repartir dans une autre direction. Indication : on peut faire `dx, dy = dy, -dx` pour effectuer un quart de tour sur la direction !
— Si toutes les directions sont bloquées car soit déjà explorées (couleur "blue") ou invalides, alors le programme se termine.

Peut-on faire mieux avec cet algorithme ? Réussir à visiter toutes les cases accessibles ?

☐ 14. **version récursive -** `grille2` Finalement, la technique ci-dessus a clairement ses limites : Pacman peut se retrouver bloqué et il faut trouver le moyen de "repartir en arrière". C'est là qu'intervient la récursivité et la pile d'appels (ou pile de récursivité) fait ce travail pour nous. Complétez la fonction `exploration` qui se trouve dans le fichier grille2. La spécification de la fonction donne les instructions à suivre.

2 Exercices non graphiques

Après le déroulement graphique passionant des deux parties précédentes, vous avez développé une intuition solide du déroulement d'une fonction récursive. Les exercices qui suivent, plus classiques, sont à traiter dans des scripts python sans sortie graphique (terminal uniquement). Vous pouvez également utiliser `jupyter`.

☐ 15. Faites tourner ce programme sur papier (ou dans votre tête) et déterminez ce qu'il fait :

```
def f15(t:list[int], n:int=0)->int:
    if n == len(t):
        return 0

    if t[n] > 0:
        return 1+f15(t, n+1)
    else:
        return f15(t, n+1)

l = [5,3,-4,2,5,-4,-2,3,-1,1,7,-2]
print(f15(l))
```

☐ 16. Sur le principe de l'exercice précédent, écrivez une fonction récursive qui détermine si tous les éléments d'un tableau sont strictement positifs. Interdit de compter avec la fonction précédente, votre fonction doit avoir le prototype : `tous_positifs(t:list[int], debut:int)-> bool`

☐ 17. Faites tourner ce programme sur papier (ou dans votre tête) et déterminez ce qu'il fait :

```
def f17(t:list[int], n:int, d:int, f:int)->bool:
    if d>f:
        return False

    m = (d+f)//2
    if t[m] == n:
        return True
    elif t[m] > n:
        return f17(t, n, d, m-1)
    else:
        return f17(t, n, m+1, f)

tab = [1,4,8,9,12,17,21,25,90]
print(f17(tab, 8, 0, len(l)-1))
print(f17(tab, 10, 0, len(l)-1))
print(f17(tab, 1, 0, len(l)-1))
```

☐ 18. Écrire une fonction récursive qui renvoie la somme des entiers $0^2 + 1^2 + \cdots + n^2$, l'entier n étant donné en paramètre (sans utiliser de formule).

☐ 19. Écrire une fonction récursive qui calcule le factoriel de n, soit

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

pour $n \geq 1$.

☐ 20. Écrire une fonction récursive qui calcule n^k en se basant sur n^{k-1} (sans utiliser l'opérateur puissance ** de python !) Combien de mutliplications effectue cette fonction ?

☐ 21. (*) Observez cette façon de procéder :

$$3^{21} = 3^{20} \times 3$$
$$3^{20} = (3^{10})^2$$
$$3^{10} = (3^5)^2$$
$$3^5 = 3^4 \times 3$$
$$3^4 = (3^2)^2$$

On calcule alors $3^2 = 9$ et on remplace en remontant les lignes, en utilisant seulement des carrés. Le nombre de multiplication total est 6 et non 8 ; il s'agit d'exponentiation rapide.

Ecrire une fonction récursive utilisant ce principe. Evaluez le nombre de multiplications/carrés effectués pour calculer n^k ?

☐ 22. Une *sous-chaîne* d'une chaîne de caractère `s` est une chaîne de caractère qui peut être obtenue en supprimant des caractères de `s`. Par exemple, les chaînes `"lgy"`, `"ggon"`, `"lna y"` sont des sous-chaînes de `"langage python"`, mais `"zou"`, `"ya"` et `"l  n"` ne le sont pas. Écrire une fonction récursive qui détermine si une chaîne de caractère `s1` est une sous-chaîne de caractère de `s2`.

☐ 23. Considérons une arborescence de fichiers élémentaires où l'on représente les répertoires et les fichiers par des listes et des chaînes de caractères. Exemple :

```
arborescence = [[['a',['b','c',['d'],'e',['f','g']],
['h',['i',['j','k']]],['l','m']]],['n','o','p'],['q','r'],
[[['s'],[['t','u']],'v'],'w','x',[['y'],'z']]]
```

On demande d'écrire des fonctions récursives qui :
— affichent tous les noms fichiers de l'aborescence dans le terminal
— testent si un fichier appartient à l'arborescence
— comptent les fichiers
On considère ici qu'un répertoire n'est pas un fichier.

☐ 24. Écrire une fonction récursive calculant le k-ième terme F_k de la suite de Fibonacci

$$F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, F_5 = 8, F_6 = 13, F_7 = 21, F_8 = 34, F_9 = 55 \dots$$

(chaque terme est la somme des deux précédents). Ensuite, afin d'évaluer l'efficacité de votre fonction, faites en sorte de compter le nombre total d'appels de fonctions qui sont faits pour calculer F_{36} (vous pouvez utiliser pour cela une variable globale). Si le nombre est grand, esssayez de réécrire la fonction autrement pour obtenir une meilleure complexité algorithmique.

☐ 25. Écrire une fonction récursive qui, pour un entier donné n, renvoie sous forme de chaîne de caractères une représentation de n en binaire (bit de poids fort en premier). Indication : considérer le dernier chiffre de la représentation binaire, en lien avec la parité de n.

☐ 26. (*) Écrire une fonction qui génère toutes les chaînes de caractères de longueur k (par exemple, les mots de passe !) que l'on peut écrire étant donné une liste de caractères de la forme `caracteres = ['a','c','x',...]`. Ceci peut être pratique pour casser un mot de passe en utilisant la force brute !

☐ 27. (*) Écrire une fonction récursive qui renvoie la liste de toutes les suites strictements croissantes de longueur k composées d'entiers de 1 à n.
Exemple : Si $k = 3$ et $n = 5$ la fonction doit renvoyer (éventuellement dans un autre ordre)

$$[[1,2,3],[1,2,4],[1,2,5],[1,3,4],[1,3,5],[1,4,5],[2,3,4],[2,3,5],[2,4,5],[3,4,5]]$$

☐ 28. (**) écrire un programme qui complète une grille de sudoku si possible. Exemple :

```
G= [ [ 3 , 0 , 0 , 4 , 1 , 0 , 0 , 8 , 7 ],
[ 0 , 0 , 9 , 0 , 0 , 5 , 0 , 6 , 0 ],
[ 4 , 0 , 0 , 7 , 9 , 0 , 5 , 0 , 3 ],
[ 0 , 7 , 3 , 2 , 4 , 0 , 0 , 0 , 0 ],
[ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ],
[ 0 , 0 , 0 , 0 , 7 , 8 , 2 , 4 , 0 ],
[ 6 , 0 , 2 , 0 , 8 , 3 , 0 , 0 , 5 ],
[ 0 , 5 , 0 , 1 , 0 , 0 , 3 , 0 , 0 ],
[ 1 , 3 , 0 , 0 , 2 , 4 , 0 , 9 , 6 ] ]
```

Il vous faudra, étant donnée une solution partielle, déterminer récursivement de quelles façons elle peut ou non être complétée.

☐ 29. (**) Quel est le nombre maximal de reines que l'on peut poser sur un échiquier $N * N$ sans que deux reines soient sur la même ligne, la même colonne ou la même diagonale ? Comme pour le sudoku, votre fonction récursive va déterminer, étant donné N et une solution partielle, s'il est possible ou non de compléter cette solution en une solution complète (on parle de backtracking). Mieux encore, calculez une ou des solutions.