Récursivité

INFORMATIQUE COMMUNE - TP n° 5 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ expliquer le principe d'un algorithme récursif
- ☞ imaginer une version récursive d'un algorithme
- ☞ trouver et coder une condition d'arrêt à la récursivité
- ☞ coder des algorithmes à récursivité simple ou multiple en Python
- ☞ identifier le type de récursivité d'un algorithme

A Penser récursivement

A1. Somme des n premiers carrés

(a) Coder un algorithme itératif qui calcule la somme des carrés des n premiers entiers , $S_n = \sum_{k=1}^{n} k^2$.

(b) Coder un algorithme récursif équivalent.

(c) Modifier le code récursif pour bien visualiser les appels et les renvois de la fonction, c'est à dire la pile d'exécution. Dans le cas de S_6, l'exécution du code affiche sur la console :

```
------> Called with n = 6
-----> Called with n = 5
----> Called with n = 4
---> Called with n = 3
--> Called with n = 2
-> Called with n = 1
Stop condition
--> Returning 5
---> Returning 14
----> Returning 30
-----> Returning 55
------> Returning 91
```

(d) Coder un algorithme récursif terminal équivalent.

A2. Inverser la position des éléments d'un tableau

(a) Coder un algorithme itératif qui inverse la position des éléments d'un tableau : le premier élément échange sa place avec le dernier, le deuxième avec l'avant dernier... On implémentera le tableau à l'aide d'une liste Python.

(b) Coder un algorithme récursif équivalent à l'algorithme itératif précédent.

B Récursivité multiple, direction le moyen âge

Leonardo Fibonacci est un figure illustre des mathématiques du moyen-âge notamment parce qu'il a introduit le système des chiffres indo-arabes en Italie, c'est à dire la numération de position en base dix à la place des chiffres romains. À l'origine, une histoire de lapins : « Quelqu'un a déposé un couple de lapins dans un certain lieu, clos de toutes parts, pour savoir combien de couples seraient issus de cette paire en une année, car il est dans leur nature de générer un autre couple en un seul mois, et qu'ils enfantent dans le second mois après leur naissance. » Peut-on décrire la croissance de la population des lapins ?

Formulé mathématiquement de nos jours, cela revient à étudier la suite $(u_n)_{n\in\mathbb{N}}$ telle que $u_0 = 0$, $u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$. Cette suite s'appelle la suite de Fibonacci.

B1. Coder une fonction récursive dont le prototype est `rec_fib(n)` où `n` est un paramètre de type `int` et qui renvoie le terme u_n de la suite de Fibonacci.

B2. Modifier le code précédent pour visualiser la pile d'exécution comme dans l'exercice précédent.

B3. Peut-on calculer u_{1200} ? Pourquoi ?

B4. Peut-on calculer u_{42} en un temps raisonnable ? Pourquoi ?

B5. Coder une fonction itérative dont le prototype est `ite_fib(n)` où `n` est un paramètre de type `int` et qui renvoie le terme u_n de la suite de Fibonacci.

B6. Coder une fonction récursive terminale dont le prototype est `term_rec_fib(n, u0=0, u1=1)` où `n` est un paramètre de type `int`, `u0` et `u1` des paramètres optionnels de type `int` et qui renvoie le terme u_n de la suite de Fibonacci.

B7. Comparer les temps d'exécution de ces différentes fonctions et analyser les résultats.

C Où l'on voyage en mathématiques

Contrairement à ce que pourrait laissait penser le titre du jeu, les tours de Hanoï n'ont rien à voir avec le Vietnam mais tout avec le mathématicien Édouard Lucas (1842-1891). Il s'agit un jeu de réflexion qui consiste à déplacer les étages de largeurs différentes d'une tour de départ vers une tour objectif en passant par une tour auxiliaire en un minimum de coups tout en respectant les règles suivantes :

1. ne déplacer qu'un seul étage à la fois, celui du sommet d'une tour,
2. déplacer un étage vers une autre tour uniquement :
   (a) si la tour de destination est vide,
   (b) ou si le dernier étage de la tour de destination est plus grand que lui.

Toute configuration initiale du jeu respecte cette dernière règle. Par exemple, une configuration de départ peut être :

```
          |         |         |
#4 :     ***        |         |
#3 :    *****       |         |
#2 :   *******      |         |
#1 : *********      |         |
```

La configuration finale correspondante est :

```
          |         |         |
#4 :      |         |        ***
#3 :      |         |       *****
#2 :      |         |      *******
#1 :      |         |     *********
```

On modélise les tours par des listes qu'on nommera `start`, `aux` et `target`. Si une liste est vide, cela signifie que la tour ne comporte pas d'étages. Un élément de type `int` d'une liste représente la largeur d'un étage d'une tour. Naturellement, si on respecte les règles, les listes doivent toujours apparaître comme des listes décroissantes.

La configuration initiale décrite ci-dessus peut être modélisée ainsi :

```
start=[4,3,2,1]
aux = []
target =[]
```

---

et la configuration finale :

```
start=[]
aux = []
target =[4,3,2,1]
```

Le programme principal ainsi qu'une méthode pour afficher les tours correctement sur la console sont fournis.

Code 4 – Code de départ pour les tours d'Hanoi

```
global start, aux, target
start = []
aux = []
```

```
target = []


def draw_stage(k, tower, size):
    if k <= len(tower):  # there is something to draw
        wm = 2 * tower[k - 1] + 1
        dec = size - wm // 2
        stage = " " * dec + "*" * wm + " " * dec
    else:
        stage = (" " * size + "|" + " " * size)
    return stage


def show_game():
    global start, aux, target
    size = max(max(start) if start else 0, max(aux) if aux else 0, max(target) if
        target else 0)
    print()
    pole = (" " * size + " |" + " " * size)
    print("      ", pole * 3)
    for k in range(size, 0, -1):
        s = draw_stage(k, start, size)
        a = draw_stage(k, aux, size)
        t = draw_stage(k, target, size)
        print("#", k, " : ", s, a, t)


def init_game(n):
    global start, aux, target
    # TODO :  INIT global start, aux and target
    pass


def move_from_a_to_b(from_a, to_b):
    # TODO : show_game(), move disc from a to b, show_game()
    pass


def hanoi(n, s, a, t):
    # TODO : do not forget stop condition !
    pass


# MAIN PROGRAM
n = 4
init_game(n)
print(start, aux, target)
show_game()
hanoi(n, start, aux, target)
show_game()
```

C1. Compléter la fonction `init_game` afin de créer une configuration initiale pour le jeu. Cette fonction initialise les variables globales `start`, `aux` et `target`. Pour $n = 4$, on obtient la configuration initiale représentée plus haut, c'est à dire qu'il y a quatre étages sur la tour de départ.

C2. Coder la fonction `move_from_a_to_b` qui déplace le disque présent sur le dessus de la tour `a` vers la tour `b`.

C3. Coder la fonction récursive `hanoi` afin de résoudre le jeu. Ne pas oublier la condition d'arrêt. On peut formuler cet algorithme en français comme suit :

> Déplacer $n-1$ étages de la tour de départ vers la tour auxiliaire, puis déplacer l'étage restant (le plus grand) de la tour de départ vers la tour objectif, puis déplacer les $n-1$ (plus petits) étages de la tour auxiliaire vers la tour objectif.

C4. Cet algorithme est-il à récursivité simple ou multiple ?

C5. On s'intéresse au nombre minimal de coups qu'il est nécessaire de jouer pour gagner. $(u_n)_{n\in\mathbb{N}^\star}$ représente ce nombre minimal de coups qu'il faut pour transférer n étages sur la tour objectif.

(a) Trouver les valeurs de u_n pour $n = 1, 2$ et 3.

(b) Inférer de ces résultats une définition de la suite $(u_n)_{n\in\mathbb{N}^\star}$ sous la forme d'une suite récurrente linéaire d'ordre un, c'est à dire $u_{n+1} = \alpha u_n + \beta$.

(c) Donner une définition explicite de $(u_n)_{n\in\mathbb{N}^\star}$ [1].

(d) Combien de coups faut-il au minimum pour transférer n disques ?

C6. À l'aide de la question précédente, vérifier que l'algorithme récursif joue un minimum de coups. Dans but, on pourra se servir d'une variable globale `moves` initialisée à zéro et incrémentée à chaque déplacement d'un étage.

R Une variable globale est déclarée tout au début du fichier en Python. On peut alors lire cette variable dans tout le fichier. Pour modifier sa valeur dans une fonction, il est nécessaire de déclarer `global moves` au début de la fonction en question [a]. Par exemple :

```
def move_from_a_to_b(a,b):
    global moves
    ...
```

a. Sous-entendu, si on ne modifie pas la valeur mais qu'on ne fait que la lire, on n'a pas besoin de cette déclaration. Mais c'est tout de même une bonne pratique de signaler l'usage d'une variable globale dans toutes les fonctions.

C7. L'ordinateur peut-il résoudre le jeu pour une tour de 64 étages [2] ?

D Diviser pour régner

D1. L'algorithme 1 permet de calculer a^n. Combien de fois l'instruction de la ligne 4 est-elle exécutée ?

D2. L'algorithme 2 permet de calculer plus rapidement a^n. Si n est pair, combien d'appels récursifs seront nécessaires pour effectuer le calcul ? Combien de fois la multiplication de la ligne 6 sera-t-elle effectuée ?

D3. Coder l'algorithme d'exponentiation rapide.

D4. Écrire une version récursive de l'algorithme de recherche dichotomique (cf. algorithme 3).

1. Si vous n'avez pas encore vu ces suites en mathématiques, expliciter directement $u_n = f(n)$.
2. La tour de 64 étages fait l'objet du «Les Brahmes tombent ! » dans le livre d'Édouard Lucas intitulé *Récréations mathématiques*. À lire en ligne ici.

Algorithme 1 Exponentiation naïve a^n

```
1 : Fonction EXP_NAIVE(a,n)
2 :     api ← 1
3 :     pour i de 0 à n − 1 répéter
4 :         api ← api × a
5 :     renvoyer api
```

Algorithme 2 Exponentiation rapide a^n

```
1 : Fonction EXP_RAPIDE(a,n)
2 :     si n = 0 alors                                  ▷ Condition d'arrêt
3 :         renvoyer 1
4 :     sinon si n est pair alors
5 :         p ← EXP_RAPIDE(a, n//2)                     ▷ Appel récursif
6 :         renvoyer p × p
7 :     sinon
8 :         p ← EXP_RAPIDE(a, (n-1)//2)                 ▷ Appel récursif
9 :         renvoyer p × p × a
```

Algorithme 3 Recherche récursive d'un élément par dichotomie dans un tableau trié

```
1 : Fonction REC_DICH(t, g, d, elem)
2 :     si g > d alors                                  ▷ Condition d'arrêt
3 :         renvoyer l'élément n'a pas été trouvé
4 :     sinon
5 :         m ← (g+d)//2                                ▷ Diviser
6 :         si t[m] = elem alors
7 :             renvoyer m
8 :         sinon si elem < t[m] alors
9 :             REC_DICH(t, g, m-1, elem)               ▷ résoudre
10 :        sinon
11 :            REC_DICH(t, m+1, d, elem)               ▷ résoudre
```