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.

Solution :

Code 1 – De l'itératif au récursif en visualisant la pile d'exécution

```
def square_sum(n):
    acc = 0
    for k in range(1, n + 1):
```

```python
        acc += k * k
    return acc


def rec_square_sum(n):
    if n == 1:
        return 1
    else:
        return n * n + rec_square_sum(n - 1)


def term_rec_square_sum(n, acc=0):
    if n == 0:
        return acc
    else:
        return term_rec_square_sum(n - 1, acc + n * n)


def call_stack_rec_square_sum(n):
    print(f"{'-' * n}> Called with n = {n}")
    if n == 1:
        print(f"Stop condition")
        return 1
    else:
        result = n * n + call_stack_rec_square_sum(n - 1)
        print(f"{'-' * n}> Returning {result}")
        return result


if __name__ == "__main__":
    N = 6
    S = N * (N + 1) * (2 * N + 1) / 6
    assert square_sum(N) == S
    assert rec_square_sum(N) == S
    assert term_rec_square_sum(N) == S

    print(square_sum(N))
    print(rec_square_sum(N))
    print(term_rec_square_sum(N))
    call_stack_rec_square_sum(N)
```

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'antépénultième, etc. On implémentera le tableau à l'aide d'une liste Python.

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

Solution :

Code 2 – Inverser la position des éléments d'un tableau

```python
from random import randint
```

```python


def swap(t, i, j):
    t[i], t[j] = t[j], t[i]


def reverse_array(t):
    for i in range(len(t) // 2):
        swap(t, i, len(t) - 1 - i)


def rec_reverse_array(t, i, j):
    if i < j:
        swap(t, i, j)
        rec_reverse_array(t, i + 1, j - 1)


if __name__ == "__main__":
    N = 8
    M = 100

    t = [randint(0, M) for _ in range(N)]
    print(t)
    reverse_array(t)
    print(t)
    rec_reverse_array(t, 0, len(t) - 1)
    print(t)
    print(t[::-1])  # simpler !
```

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 récursive terminale dont le prototype est `term_rec_fib(n, acc=0)` où `n` est un paramètre de type `int`, `acc` un paramètre optionnel de type `int` et qui renvoie le terme u_n de la suite de Fibonacci.

B6. Modifier le code de la fonction récursive terminale pour visualiser la pile d'exécution comme dans l'exercice précédent. Peut-on calculer u_{42} en un temps raisonnable ? Pourquoi ?

B7. 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.

B8. Comparer les temps d'exécution de ces différentes fonctions et analyser les résultats.

Solution : La première formulation récursive est inefficace car elle fait des appels récursifs redondants : certaines de u_n valeurs sont calculées plusieurs fois. Par exemple pour u_6, u_2 est calculé cinq fois, ce qui est inutile.

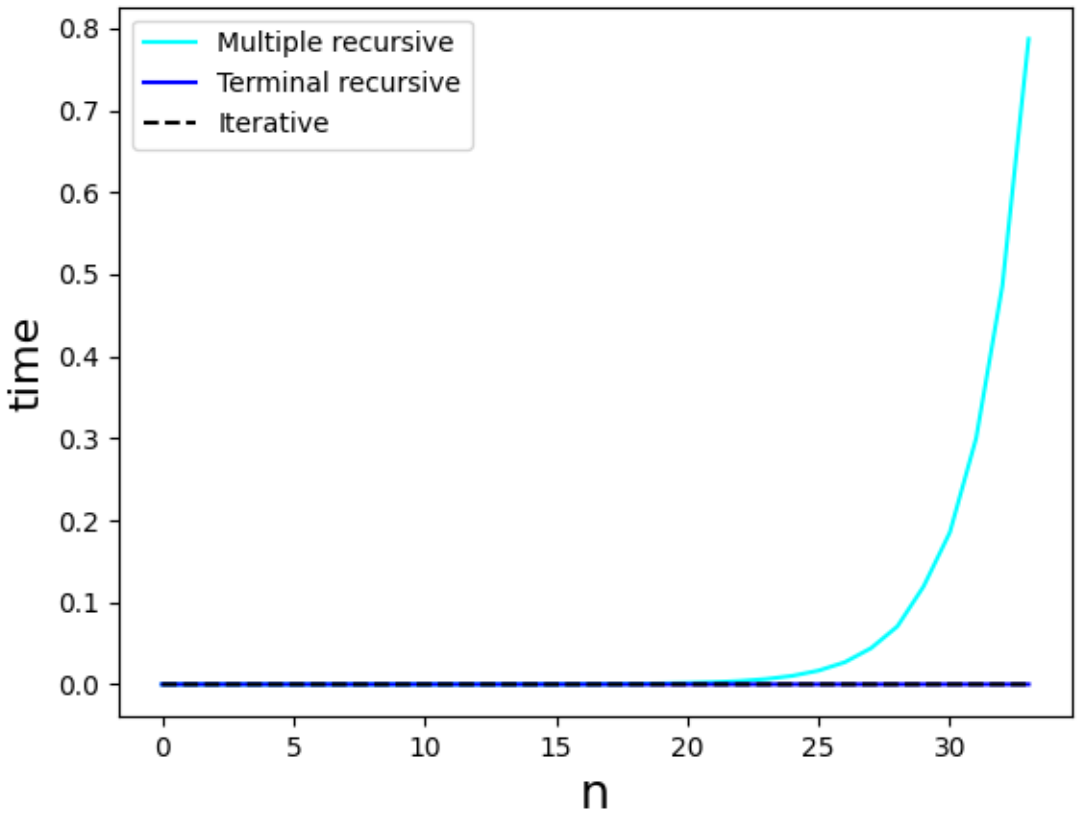
La version itérative tout comme la version récursive terminale ne font pas de calculs redondants, c'est pourquoi elles sont plus rapides. La version itérative ne fait pas d'appels de fonction c'est pourquoi elle est plus rapide.

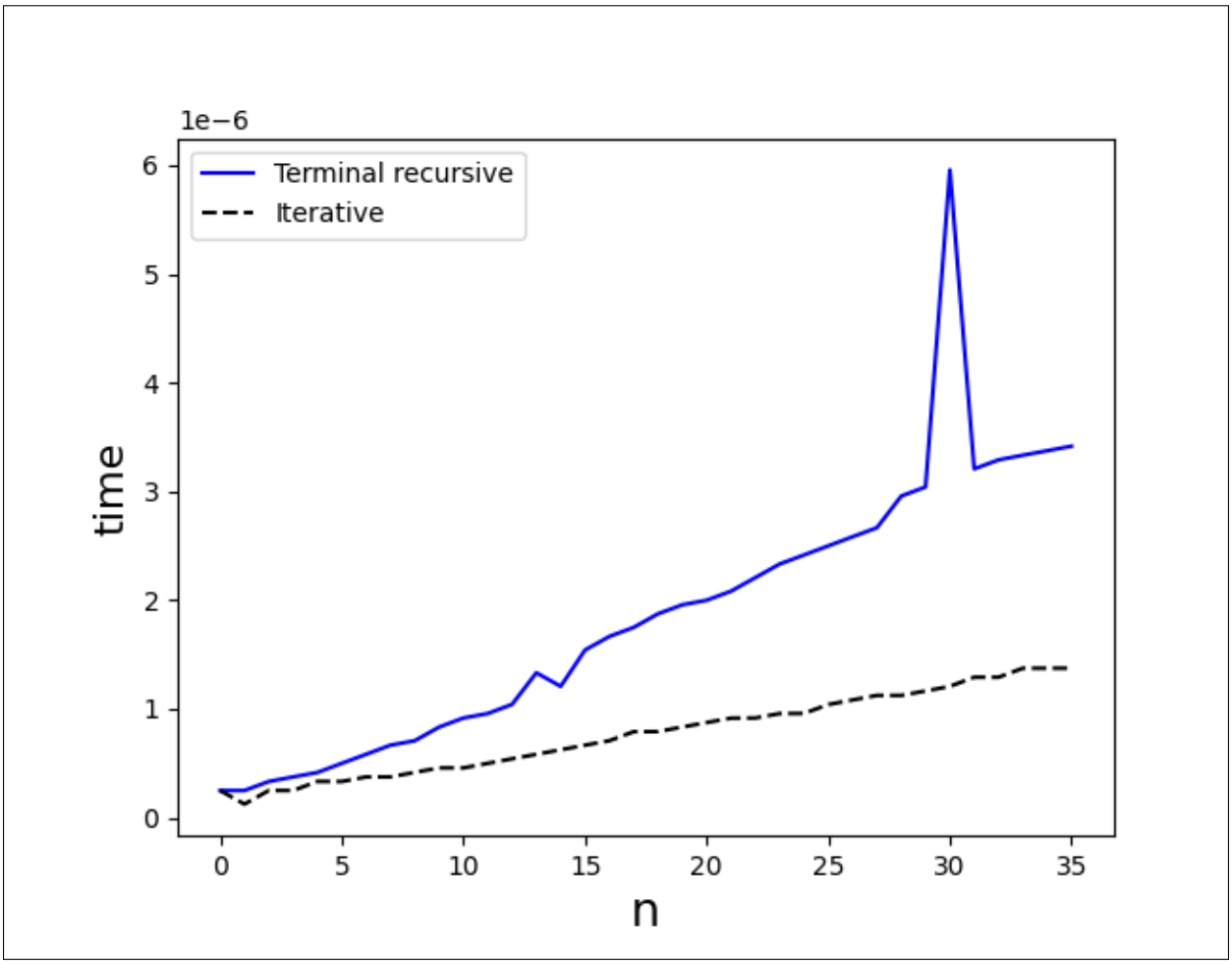
Code 3 – Fibonacci à gogo

```
import time


def rec_fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return rec_fib(n - 1) + rec_fib(n - 2)


def call_stack_rec_fib(n):
    print(f"{'-' * n}> Called with n = {n}")
    if n == 0:
        print("Stop condition")
        return 0
    elif n == 1:
        print("Stop condition")
        return 1
    else:
        result = call_stack_rec_fib(n - 1) + call_stack_rec_fib(n - 2)
        print(f"{'-' * n}>Returning --> {result}")
        return result


def term_rec_fib(n, u0=0, u1=1):
    if n == 0:
        return u0
    elif n == 1:
        return u1
    else:
        return term_rec_fib(n - 1, u1, u0 + u1)


def call_stack_term_rec_fib(n, u0=0, u1=1):
    print(f"{'-' * n}> Called with n = {n}")
    if n == 0:
```

```
        print("Stop condition")
        return u0
    elif n == 1:
        print("Stop condition")
        return u1
    else:
        result = call_stack_term_rec_fib(n - 1, u1, u0 + u1)
        print(f"{'-' * n}>Returning --> {result}")
        return result


def ite_fib(n):
    u0 = 0
    u1 = 1
    if n == 0: return u0
    if n == 1: return u1
    while n > 1:
        u0, u1 = u1, u0 + u1
        n = n - 1
    return u1


def fibonacci_timing():
    N_MAX = 36
    results = []
    for i in range(N_MAX):
        results.append([])
        for method in [term_rec_fib, ite_fib, rec_fib]:
            tic = time.perf_counter()
            method(i)
            toc = time.perf_counter()
            results[i].append(toc - tic)
        print(f"#{i} -> {results}")

    term_rec = [results[i][0] for i in range(N_MAX)]
    ite = [results[i][1] for i in range(N_MAX)]
    rec = [results[i][2] for i in range(N_MAX)]

    from matplotlib import pyplot as plt

    plt.figure()
    plt.plot(rec, color='cyan', label='Multiple recursive')
    plt.plot(term_rec, color='blue', label='Terminal recursive')
    plt.plot(ite, '--', color='black', label='Iterative')
    plt.xlabel('n', fontsize=18)
    plt.ylabel('time', fontsize=16)
    plt.legend()
    plt.show()


if __name__ == "__main__":
    fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
        987, 1597, 2584, 4181, 6765, 10946,
                 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229,
                     832040, 1346269, 2178309, 3524578]
```

```
92      print(len(fibonacci))
93      for i in range(len(fibonacci)):
94          assert rec_fib(i) == fibonacci[i]
95          assert term_rec_fib(i) == fibonacci[i]
96          assert ite_fib(i) == fibonacci[i]
97
98      call_stack_rec_fib(6)  # to see redundant calls of u_(n-p)
99      # rec_fib(42) # very long
100     # rec_fib(1200) # RecursionError: maximum recursion depth exceeded in
            comparison
101     call_stack_term_rec_fib(6)
102     fibonacci_timing()
```




1e−6
Terminal recursive
Iterative
time
n
0
1
2
3
4
5
6
0
5
10
15
20
25
30
35

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. ne 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 principale ainsi que une méthode pour afficher les tours correctement sur la console sont fournies.

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

```
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(start, aux, target):
    size = max(max(start) if start else 0, max(aux) if aux else 0, max(target) if
        target else 0)
    space = " " * (2 * size + 1)
    print()
    pole = (" " * size + "|" + " " * size)
    print(f"     {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(f"#{k} : {s}{a}{t}")


def init_game(n):
    # TODO
    return [], [], []


def hanoi(n, start, aux, target):
    # TODO
    pass


if __name__ == "__main__":
    n = 4
    s, a, t = init_game(n)
    print(s, a, t)
    show_game(s, a, t)
    hanoi(n, s, a, t)
    show_game(s, a, t)
```

C1. Compléter la fonction `init_game` afin de créer une configuration initiale pour le jeu. 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 récursive `hanoi` afin de résoudre le jeu. 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.

C3. Cet algorithme est-il à récursivité simple ou multiple ?

C4. 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 ?

C5. 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 hanoi(n, start, aux, end):
        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.

C6. L'ordinateur peut-il résoudre le jeu pour une tour de 64 étages [2] ?

Solution : Si un déplacement s'effectue en une microseconde, cela demanderait aujourd'hui plusieurs centaines de milliers d'années à un ordinateur standard…

Solution : On peut écrire $u_{n+1} = 2u_n + 1$ ce qui donne, après étude de la suite, une forme explicite $u_n = 2^n - 1$.

Code 5 – Tour de Hanoi

```
global moves
moves = 0

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(start, aux, target):
    size = max(max(start) if start else 0, max(aux) if aux else 0, max(target)
        if target else 0)
    space = " " * (2 * size + 1)
    print()
    pole = (" " * size + "|" + " " * size)
    print(f"     {pole * 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.

```
    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(f"#{k} : {s}{a}{t}")


def init_game(n):
    start = [i for i in range(n, 0, -1)]
    aux = []
    target = []
    return start, aux, target


def hanoi(n, start, aux, target):
    global moves   # mandatory because moves is modified below
    if n == 1:
        target.append(start.pop())
        moves += 1
    else:
        hanoi(n - 1, start, target, aux)
        hanoi(1, start, aux, target)
        hanoi(n - 1, aux, start, target)


if __name__ == "__main__":
    moves = 0
    n = 4
    s, a, t = init_game(n)
    print(s, a, t)
    show_game(s, a, t)
    hanoi(n, s, a, t)
    show_game(s, a, t)
    print(moves)

    for i in range(1, 23):
        moves = 0
        s, a, t = init_game(i)
        hanoi(i, s, a, t)
        ui = 2 ** i - 1
        assert ui == moves
```