

TP 6 : Fonctions récursives

I Introduction

Les fonctions récursives ont plusieurs intérêts :

- leur écriture gagne souvent en lisibilité (une fois l'habitude prise!) par rapport à l'utilisation de for ou de while ;
- elles permettent de résoudre des problèmes d'habitude quasiment irrésolvables avec l'utilisation de for ou while.

Pour s'assurer qu'une fonction récursive se termine, il est coutume de placer dans les premières lignes une condition d'arrêt.

Par exemple cette fonction ne se termine jamais :

```
def f(x) :
    x = 2 * x
    f(x)
    return x
```

```
f(3)
    x = 2 * 3
        x = 2 * 6
            x = 2 * 12
                ...
```

```
>>> f(3)
RecursionError: maximum recursion depth exceeded
```

II Suites définies par récurrence

II.1 Factorielle de n

Soit $n \in \mathbb{N}$. La factorielle de n est définie par récurrence par :

$$0! = 1, \quad n! = n \times (n-1)! \quad \forall n \in \mathbb{N}^*.$$

Compléter les algorithmes suivants :

Version itérative.

```
def fact(n: int) -> int:
    p = 1
    for i in range(          ):
        p = p * i
    return p
```

Version récursive.

```
def fact_rec(n: int) -> int:
    if n == 0:
        return 1
    else:
        return
```

```
>>> fact_rec(50)
30414093201713378043612608166064768844377641568960512000000000000
```

Exercice 1

Soit u la suite définie par $u_0 = 6$ et $u_{n+1} = 4u_n - 7$. Ecrire une fonction récursive qui prend en paramètre n et renvoie u_n . On donnera aussi une version itérative.

Exercice 2

Soit u la suite définie par $u_0 = 6$ et $u_{n+1} = 4u_n - 7n$. Ecrire une fonction récursive qui prend en paramètre n et renvoie u_n .

II.2 Suite de Fibonacci

La suite de Fibonacci $(f_n)_{n \in \mathbb{N}}$ est définie par récurrence par :

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad \forall n \in \mathbb{N}, n \geq 2.$$

Compléter les algorithmes suivants :

Version itérative.

```
def fibonacci(n: int) -> int:
    u = 0
    v = 1
    for i in range( ):
        w = v # on sauvegarde v
        v = v + u
        u = w
    return v
```

Version récursive.

```
def fib(n: int) -> int:
    if n == 0:
        return
    elif n == 1:
        return
    else:
        return
```

```
>>> fibonacci(10)
55
>>> fib(10)
55
```

```
>>> fibonacci(40)
102334155
>>> fibonacci(100)
354224848179261915075
```

⇒ Essayez !

```
>>> fib(40)
```

II.3 Coefficient binomial

Rappelons que pour tous $n, p \in \mathbb{N}$, on a la relation de Pascal suivante :

$$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}.$$

Cette relation nous permet de construire par récurrence tous les coefficients binomiaux en sachant que $\binom{k}{k} = 1$ pour tout $k \in \mathbb{N}$.

II.3.1 Version itérative du calcul de $\binom{n}{p}$

```
def ligne_suivante(L: list) -> list:
    ''' construit la ligne qui suit L
    dans le triangle de Pascal '''
    LL = []
    LL.append(1)
    for k in range(1, len(L)):
        LL.append(L[k-1] + L[k])
    LL.append(1)
    return LL
```

```
def pascal_it(n: int, p: int) -> int:
    ''' renvoie le coeff binomial
    p parmi n '''
    L = [1, 1]
    for i in range(n-1):
        L = ligne_suivante(L)
    return L[p]
```

Compléter la version récursive :

```
def pascal(n: int, p: int) -> int:
    ''' renvoie le coeff binomial p parmi n '''
    if p > n:
        return
    elif p == 0 or p == n:
        return
    return
```

```
>>> pascal(10, 6)
210
```

III Sur les chaînes de caractères

III.1 Compter le nombre d'un caractère d'une chaîne

Compléter l'algorithme version récursive suivant :

Version itérative.

```
def compte_a(chaine: str) -> int:
    c = 0
    for lettre in chaine:
        if lettre == 'a':
            c = c + 1
    return c
```

Version récursive.

```
def compte_a_rec(chaine: str) -> int:
    if len(chaine) == 1:
        if chaine[0] == 'a':
            return 1
        else:
            return 0
    else:
        if chaine[0] == 'a':
            return
        else:
            return
```

```
>>> compte_a_rec('barbapapa')
4
```

III.2 Figures alphanumériques

⇒ Essayez !

```
def rectangle(n: int) -> None:
    if n > 0:
        print("*" * 5)
        rectangle(n-1)
```

```
>>> rectangle(5)
```

1. Ecrire une fonction récursive `triangle_bas(n)` qui étant donné n un entier naturel, produit le schéma suivant (la première ligne contient n étoiles) :

```
*****
****
***
**
*
```

2. Ecrire une fonction récursive `triangle_haut(n)` qui étant donné n un entier naturel, produit le schéma suivant (la dernière ligne contient n étoiles) :

```
*
**
***
****
*****
```

3. Ecrire une fonction récursive `figure(n)` qui étant donné n un entier naturel, produit le schéma suivant (la première ligne contient n étoiles) :

```
****
***
**
*
*
**
***
****
```

III.3 Modifier une chaîne de caractères

Déterminer l'action de cette fonction sur une chaîne de caractères :

```
def mystere(ch: str) -> None:
    if len(ch) > 0 :
        mystere(ch[1:])
        print(ch[0], end='')      # le end='' permet de ne pas sauter à la ligne
                                # à la fin du print
```

```
>>> mystere('bonjour')
```

IV Sur les listes

IV.1 Doublons

Ecrire un algorithme qui supprime les doublons d'une liste donnée en paramètre.

Version itérative

```
def doublons(L:list) -> list:
```

Version récursive

```
def doublons_rec(L:list) -> list:
    if len(L) <= 1:
        return L
    if L[0] != L[1]:
        return [L[0]] + doublons_rec(L[1:])
    del L[1]
    return doublons_rec(L)
```

IV.2 Une utilisation de pop

Rappel : l'instruction `L.pop(i)` supprime l'élément d'indice `i` de la liste `L`.

Décrire - avant de la tester sur l'ordinateur - l'action de la fonction `pp` ci-dessous sur une liste non vide `L` :

```
def pp(L: list):
    if len(L) == 1:
        return L[0]
    elif L[0] < L[1]:
        L.pop(1)
    else:
        L.pop(0)
    return pp(L)
```

IV.3 Recherche par dichotomie

On dispose d'une liste ordonnée (par ordre croissant) `L` et d'une valeur `e` dont il faut tester la présence dans la liste `L`.

Version itérative.

```
def dichotomie(L: list, e: int) -> int:  
    deb = 0  
    fin = len(L) - 1  
    milieu = (deb + fin) // 2  
    while L[milieu] != e:  
        if L[milieu] > e:  
            fin = milieu - 1  
        else:  
            deb = milieu + 1  
    return (deb + fin) // 2
```

Version récursive.

```
def rech_dicho_rec(L: list, e: int, deb: int, fin: int):  
    if deb == fin :  
        return L[deb] == e  
    mi = (deb + fin) // 2
```