TP 11 - Récursivité

I. La récursivité

La récursivité est le caractère d'un algorithme qui s'appelle lui-même.

- Un algorithme récursif comporte ces trois aspects :
  - une des instructions appelle l'algorithme lui-même
  - un cas simple est traité, souvent une condition initiale ou une condition d'arrêt
  - les paramètres d'entrées successifs sont tous de même forme et tendent à réduire leur complexité pour se ramener au cas simple.
- Cette approche a des avantages, elle permet :
  - une écriture lisible voire plus courte d'un algorithme
  - une gestion allégée des variables.*
- Cependant, elle comporte des inconvénients :
  - il est parfois difficile de montrer que l'algorithme s'arrête en temps fini
  - on perd la maîtrise l'espace mémoire utilisé
  - l'efficacité est souvent moins bonne.

Exemple : l'algorithme d'Euclide pour la calcul du pgcd

Rappel des notations : $a, b \in \mathbb{Z}$

$$\exists!(q,r) \in \mathbb{Z}; a = qb + r \text{ et } 0 \leq r < |b|$$

Alors $PGCD(a,b) = PGCD(b,r)$
On réitère ; le dernier reste non nul est le pgcd.

Attention ! En Python, le reste de la division euclidienne est compris entre 0 et b ($\neq b$), ce n'est pas la même convention qu'en mathématique.

Euclide itératif

```
def pgcd_ite(a,b):
    a,b=abs(a),abs(b)
    while b!=0:
        a,b=b,a%b
    return a
```

Euclide récursif

```
def pgcd_rec(a,b):
    if b==0:
        return a
    else:
        return pgcd_rec(b,abs(a%b))
```

Exercice 1

1. Faire un tableau de suivi des variables sur l'instance de votre choix.
2. Modifier pgcd_rec pour retourner la profondeur des appels.

Exemple : Calcul du n-ième terme de la suite de Fibonacci :

$$\begin{cases} u_0 = 0 \quad u_1 = 1 \\ \forall n \geq 2,\ u_n = u_{n-1} + u_{n-2} \end{cases}$$

Suite de Fibonacci

```
def fibonacci(n):
    assert type(n)==int and n>=0,
                ' n, un entier naturel'
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

⇛ Dénombrement des appels : soit $\mathscr{C}(n)$ le nombre d'appels récursifs avec l 'entrée n :

$$\mathscr{C}(n) = \mathscr{C}(n-1) + \mathscr{C}(n-2) \qquad \left(\text{rappel : } \varphi = \frac{1+\sqrt{5}}{2}\right)$$

On retrouve la suite de Fibonacci et donc $\mathscr{C}(n) = \mathcal{O}(\varphi^n)$.

- De plus, chaque appel nécessite de faire une copie des variables d'entrées : création de variables locales et donc utilisation de la mémoire.

Ici, le recours au récursif n'est pas avantageux ; un algorithme itératif est très efficace tant sur le plan temporel (nombres d'appels) que sur plan spatial (utilisation de la mémoire).

```
def fibonacci_ite(n):
    if n==0:
        return 0
    else:
        u,v=0,1
        for i in range(2,n+1):
            u,v=v,u+v
        return v
```

➜ Comparaison temporelle :

n	Itératif	Récursif
20	1.9 e-05	2.8 e-02
30	3.2 e-05	5.3
31	3.1 e-05	8.6
32	4.8 e-05	14.0
33	2.8 e-05	24.4
100	4.5 e-05	
1000	5.2 e-04	

Attention ! En Python, le nombre d'appels est limité (à 1000) et un message d'erreur peut interrompre le travail : `RuntimeError: maximum recursion depth exceeded`.

Remarque : Il existe des solutions pour linéariser ce type de situation, par exemple en créant une *table d'appels*. La fonction consulte la table pour savoir si cette valeur a déjà été calculée, sinon elle lance son calcul.

II. Exercices

Exercice 2 Que fait la fonction suivante :

```
def f(n):
    print(n*'+')
    if n>0:
        f(n-1)
```

Exercice 3 *Factorielle* Écrire le script d'une fonction `factorielle(n)` qui calcule $n!$.

Exercice 4 Proposer une approche récursive pour chaque cas suivant :

1. La suite suivante converge vers $\sqrt{5}$:

$$\begin{cases} u_0 = 2 \\ \forall n \in \mathbb{N},\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{5}{u_n}\right) \end{cases}$$

Écrire le script d'une fonction `r5(n)` qui retourne le n-ième terme de cette suite.

2. Calcul de $\displaystyle\sum_{k=1}^{n} \frac{1}{k}$

3. Calcul de $\binom{n}{k}$ par la formule : pour $p > 0$

$$\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$$

4. Considérons les suites (u_n) et (v_n) par $v_0 = 1$, $u_0 = 2$ et pour tout $n \in \mathbb{N}$:

$$u_{n+1} = \frac{u_n + v_n}{2} \qquad v_{n+1} = \frac{2u_n v_n}{u_n + v_n}$$

Créer les fonctions `u(n)` et `v(n)` qui calcule u_n et v_n.

Remarque : La récursivité est parfois croisée dans le sens où une première fonction appelle une autre fonction et vice-versa.

Exercice 5 *Tri par sélection* Proposer une fonction récursive mettant en oeuvre le tri par sélection suivant :

Tri par sélection

```
def pmin(L):
    ...

def tri_selection(L):
    LL=[]
    for i in range(len(L)):
        p=pmin(L)
        LL.append(L.pop(p))
    return LL
```

Exercice 6 *Dichotomie* **Prog**

1. Donner une version récursive de `dicho_f(f,a,b,p)` qui retourne une valeur approchée à la précision p d'une racine de f sur $[a,b]$ vérifiant $f(a)f(b) \leq 0$ par la méthode de dichotomie.
2. Donner une version récursive de `dicho_L(x,L` qui teste si l'élément x est dans le liste ordonnée `L` par la méthode de dichotomie.

Exercice 7 Proposer une fonction récursive qui retourne le maximum d'une liste de nombres.