TP 5 - Récursivité

Le but de ce TP est d'introduire une nouvelle démarche algorithmique dont la description mène à la répétition d'une même règle.

I Définition

Pour faire simple, une fonction récursive consiste juste en une fonction qui s'appelle elle-même. Prenons un exemple classique : le calcul de la factorielle.

On définit, pour $n \geqslant 0$, $n! = \prod_{i=1}^{n} i$. Informatiquement, on peut donc définir la factorielle comme :

Factorielle non récursive

```
def fact(n) :
    assert n>=0
    f=1
    for i in range(1,n+1) :
        f=f*i
    return f
```

Une autre définition mathématique classique de la factorielle se fait par récurrence :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

En reprenant quasiment mot pour mot cette dernière définition, on obtient la fonction Python suivante

Factorielle récursive

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

qui fonctionne tout aussi bien ! On distingue clairement deux cas dans cette fonction :
— le cas $n = 0$, appelé cas terminal ;
— le cas $n > 0$, qui produit un appel récursif à la fonction fact_rec.

II D'autres exemples

Algorithme d'Euclide Soient a et b deux entiers naturels, avec $a > 0$. Si $b \neq 0$, on a la relation $PGCD(a, b) = PGCD(b, r)$, où r est le reste dans la division euclidienne de a par b. L'algorithme usuel de calcul du $PGCD$ consiste donc à faire des divisions euclidiennes :

PGCD non récursif

```
def PGCD(a,b) :
    while b>0 :
        a,b=b,a%b
    return a
```

La version récursive repose sur le même principe :

PGCD récursif

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

Calcul de puissance On possède deux méthodes "classiques" pour calculer x^n, un algorithme d'exponentiation naïf (faisant usage d'une boucle for, calculant toutes les puissances de x entre 1 et n), et un algorithme d'exponentiation rapide (consistant à remarquer que $x^n = (x^{n//2})^2 * x^{n\%2}$). Les deux admettent des équivalents récursifs :

Exponentielle classique récursive

```
def expo(x,n) :
    if n==0 :
        return 1
    else :
        return x*expo(x,n-1)
```

Exponentiation rapide récursive

```
def expo_rapide(x,n) :
    if n==0 :
        return 1
    else :
        y=expo_rapide(x,n//2)
        if n%2==0 :
            return y*y
        else :
            return y*y*x
```

Maintenant à vous d'essayer :

Question 1. (Suite définie par une relation de récurrence)
On considère la suite (dite des babyloniens) définie par la récurrence suivante :

$$\begin{cases} u_0 = 3 \\ \forall n \in \mathbb{N},\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{5}{u_n}\right) \end{cases}$$

Ecrire une fonction récursive `suite(n)` qui prend en entrée un entier $n \in \mathbb{N}$ et renvoie la valeur de u_n.

Indication : Rien n'empêche une fonction récursive de se rappeler plusieurs fois.

III Récursivité sur les listes

On considère une liste d'entiers `L`. On rappelle que le *slicing* permet de créer une liste à partir d'une autre liste. Par exemple, avec $L = [5, 3, 2, 7, 1, 2, 6, 8]$, la commande $L[2:5]$ représente la liste $[2, 7, 1]$; $L[:4]$ représente $[5, 3, 2, 7]$ et $L[4:]$ représente $[1, 2, 6, 8]$.

Question 2. Ecrire une fonction récursive `somme` qui renvoie la somme des éléments de la liste `L`.

Question 3. Ecrire une fonction récursive `produit` qui renvoie le produit des éléments de la liste `L`.

Question 4. Ecrire une fonction récursive `maximum` qui renvoie l'élément maximal de la liste `L`, supposée non vide.

IV Récursivité sur les chaînes de caractères

IV.A Premières fonctions

Question 5. Ecrire une fonction récursive `estPresent` qui prend en argument une chaîne de caractères `s`, non vide, et une lettre `l`, et qui renvoie un booléen indiquant si la chaîne `s` contient la lettre `l`.

Question 6. Ecrire une fonction récursive `nbOccurences` qui prend en argument une chaîne de caractères `s` et une lettre `l`, et qui renvoie le nombre d'occurrences de la lettre `l` dans la chaîne `s`.

IV.B Palindromes

On appelle *palindrome* un mot qui peut se lire dans les deux sens. Par exemple, "kayak" est un palindrome mais pas "bonjour".

Question 7. Ecrire une fonction récursive `estPalindrome` qui renvoie `True` si la chaîne de caractères `s` est un palindrome et `False` sinon.

IV.C Anagrammes

On considère une chaîne de caractères `s`, on souhaite créer une liste contenant tous les anagrammes distincts de la chaîne `s`. C'est-à-dire tous les mots obtenus en mélangeant les lettres de la chaîne `s`. Par exemple, les anagrammes de "bac" sont "abc", "acb", "bac", "bca", "cab" et "cba", et les anagrammes de "ici" sont "ici", "iic" et "cii".

Question 8. Ecrire une fonction récursive `anagrammes` qui prend en argument une chaîne de caractères `s` et qui renvoie la liste des anagrammes distincts de cette chaîne.

V La suite de Fibonacci

Question 9. On définit la suite de Fibonacci par $F_0 = F_1 = 1$ et pour tout $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$.

1. Ecrire une fonction itérative `fiboit` qui prend en argument un entier `n` et qui renvoie le terme F_n de la suite de Fibonacci correspondant.
2. Ecrire une fonction récursive `fiborec` qui prend en argument un entier `n` et qui renvoie le terme F_n de la suite de Fibonacci correspondant.
3. Comparer la rapidité d'exécution de ces fonctions sur des valeurs de n de plus en plus grande. Par exemple $n = 10$, $n = 20$, $n = 30$ et $n = 45$.

Le dernier point met en évidence un problème sur lequel on reviendra au second semestre. En attendant, on peut résoudre ce problème en rusant :

Question 10.

1. Ecrire une fonction récursive `fibo_term` qui prend en entrée trois entiers a, b et n et qui renvoie F_{n+m} lorsque $a = F_m$ et $b = F_{m+1}$. Cette fonction devra, en comptant tout ses appels récursifs, n'effectuer que n additions.
2. En déduire une fonction `fiborec2(n)` calculant les termes de la suite de Fibonacci plus rapidement que `fiborec(n)`.

VI Plus difficile

Question 11. (Recherche dichotomique dans une liste triée)
Ecrire une fonction récursive `recherche_dicho` prenant en entrée une liste ℓ de nombres triée par ordre croissant et un nombre a et qui renvoie `True` si a appartient à la liste ℓ, `False` sinon. Cette fonction devra utiliser l'algorithme de recherche dichotomique, c'est à dire qu'elle comparera a à l'élément situé au milieu de la liste et si ce dernier est différent, la fonction cherchera a soit dans la sous liste de droite soit dans la sous liste de gauche.

Indication : On rappelle que l'instruction `L[i:j]`, pour une liste `L`, construit la sous liste des éléments de `L` depuis celui d'indice i (inclus) jusqu'à celui d'indice j (exclu).

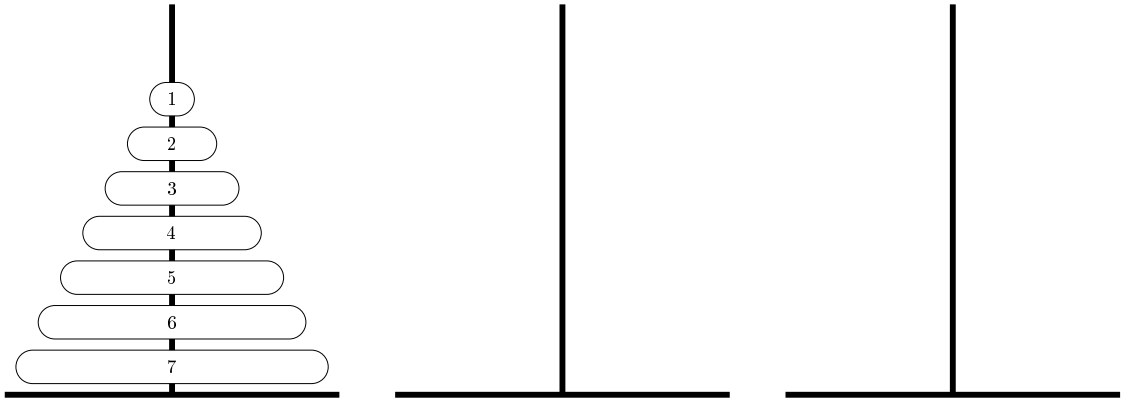
Question 12. (Algorithme d'Euclide étendu)
Ecrire une fonction récursive `pgcd_bezout(n1,n2)` prenant en paramètre deux entiers naturels n_1 et n_2 et qui renvoie le triplet suivant : $(n_1 \wedge n_2, u, v)$, où $un_1 + vn_2 = n_1 \wedge n_2$ est une relation de Bézout.

Question 13. (Suppression de doublons)
Ecrire une fonction récursive `supprime_doublon(L)` qui prend en paramètre une liste d'entiers supposée triée et qui construit une liste composée des mêmes éléments mais sans doublons.

VII Les tours de Hanoï

Nous allons voir ici un petit jeu classique dont la résolution par une stratégie récursive illustre bien la puissance de ce type de réflexion : le problème des tours de Hanoï.

On dispose de n disques troués en leur centre, numérotés de 1 à n, de diamètres croissants. On se donne également 3 piquets (indicés A, B et C). Initialement, tous les disques sont enfilés sur le premier piquet, le plus grand étant à la base et le plus petit au sommet. Comme sur la figure ci-dessous avec 7 disques :



Le but du jeu est d'amener les disques sur le piquet C tout en suivant les règles suivantes :

— On ne peut déplacer qu'un disque à la fois en le prenant du sommet d'un piquet pour l'amener au sommet d'un autre piquet. On a par exemple, sur la figure précédente, interdiction de prendre directement le disque 7 pour l'amener sur le piquet C ni prendre d'un coup toute la pile de disques et transférer sur le lieu d'arrivée.

— De plus, un disque ne doit jamais être posé sur un disque de diamètre inférieur. Encore sur notre exemple, on ne peut déplacer le disque 1 sur le piquet B puis déplacer le disque 2 par dessus le disque 1 sur le piquet B.

On cherche un algorithme nous donnant (s'ils existent) les mouvements de disques à effectuer pour résoudre ce jeu. De manière itérative, il n'est pas évident à résoudre, mais il devient excessivement simple lorsque l'on pense de manière récursive (ce qui, au passage va montrer qu'il existe une résolution pour n'importe quel nombre de disques).

Question 14. Donner une suite de mouvements permettant de résoudre le problème pour $n = 3$

Pour la résolution générale, on va complexifier légèrement les paramètres du problème dans le sens où on va devoir, en plus de se donner un nombre de disques $n \in \mathbb{N}$, préciser le rôle de chaque piquet.

On définit ainsi un problème auxiliaire dont une instance sera `Auxiliaire`(i, j, k, n) avec le caractère i désignant le piquet initial (sur lequel se trouvent les n disques au début), le caractère j désignant le piquet final (sur lequel on veut tous les disques à la fin) et le caractère k désignant le piquet intermédiaire.

Il suffira alors d'appeler `Auxiliaire`$("A","C","B", n)$ pour le jeu de Hanoï présenté précédemment.

Question 15. En utilisant la fonction `deplacement` suivante, Ecrire une fonction `Auxiliaire` résolvant le problème précédemment décrit.

Fonction déplacement

```
1 def deplacement(i,j) :
2     print(i, "->", j)
```

Question 16. En déduire une fonction `hanoi(n)` résolvant notre jeu originel.