TP Informatique 14 : Récursivité

L'approche récursive est une méthode fondamentale en informatique. Le principe est de résoudre un problème avec une instance de taille donnée en calculant les solutions pour des instances de taille inférieure.
La récursivité permet de résoudre certains problèmes d'une manière très rapide, alors que si l'on devait les résoudre de manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.
La définition formelle d'une fonction récursive n'est pas une chose aisée, en voici une définition très simple :

Une fonction est récursive si elle intervient dans sa définition.

L'objectif de cette séance est double : introduire la notion de récursivité et montrer la puissance de l'approche récursive des problèmes.

Exemple introductif : la factorielle

La fonction factorielle peut être définie de manière itérative à l'aide de la relation : $n! = \prod_{k=1}^{n} k$ et conduit au programme suivant :

```
def fact_iter (n) :
    p = .....
    for k in range (1, n+1) :
        ..............
    return p
```

La factorielle possède également une définition récursive :

$$0! = 1$$

$$n! = n.(n-1)!$$

Autrement dit, pour connaître n !, il faut connaître (n-1) !...qui lui même utilisera (n-2) ! , et ainsi de suite.
La fonction factorielle peut être définie de manière récursive de la manière suivante :

```
def fact_rec (n) :
    assert .....# assertion  on vérifie que n est positif
    if n == 0 :
        return ..... # Terminaison ou condition d'arrêt
    else :
        return ..... # appel récursif
```

Remarques importantes :

i) A propos de la **terminaison : une fonction récursive doit toujours inclure au moins un cas terminal (ou condition d'arrêt) qui ne demande pas d'appel récursif.** Une fonction récursive doit toujours finir par aboutir à un tel cas, faut de quoi la fonction part en récursivité infinie.
Dans le cas de la factorielle, comme elle fait appel à elle-même avec un paramètre diminué de 1, on aboutit forcément à 0 et la fonction donnera un résultat...à condition d'avoir un **entier n positif** (avec n négatif, la fonction ne s'arrête jamais!!).
ii) A propos de l'**assertion** : l'instruction *assert n > 0* teste si le booléen n > 0 est vérifié avant de poursuivre. Si l'expression est évaluée à False, la fonction est interrompue et un message d'erreur apparaît.
iii) Il est parfois utile de représenter un **arbre des appels récursifs** qui permet de visualiser les différents appels qui apparaissent sous forme d'une **pile.** C'est le dernier appel (terminaison) qui permet de dépiler.

Production de figures alphanumériques à l'aide de prints successifs

On considère le programme ci-dessous. Prévoir ce que l'on obtient à l'écran et le vérifier pour n = 5.

```
def figure(n):
    if n > 0 :
        print(n*"*")
        figure(n-1) # appel récursif
```

Exercice 1. Écrire une fonction récursive permettant d'afficher la matrice triangulaire inférieure d'astérisques de taille n.

Ainsi pour n = 5, le programme doit afficher :

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

Algorithme de recherche dichotomique : version récursive

On peut déterminer si un élément x est présent dans une liste L à l'aide d'un algorithme dichotomique, si l'on suppose que L est triée dans l'ordre croissant.

Pour ce faire, si L contient au moins un élément, on la coupe en deux en calculant l'indice m de l'élément situé au milieu de la liste. Puis on compare L[m] et l'élément x :

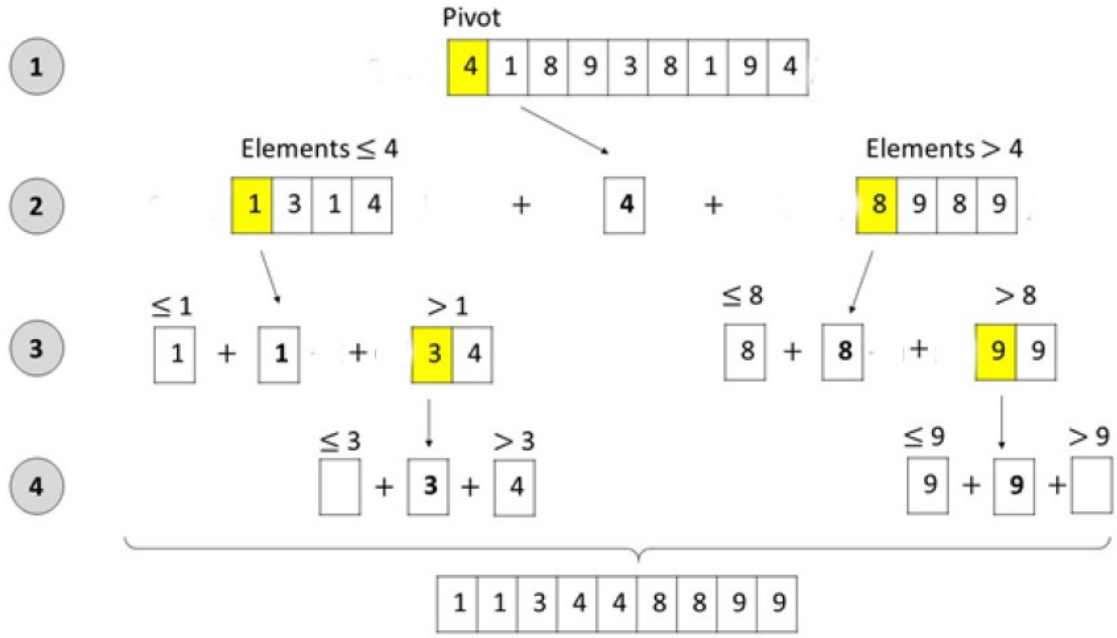
- s'ils sont égaux, on a trouvé x
- si x < L[m], comme L est triée dans l'ordre croissant, si x se trouve dans L, il se situe forcément dans la portion à gauche de L[m]
- à l'inverse, si x > L[m], alors si x se trouve dans L il se situe dans la portion à droite de L[m]

Exercice 2. Compléter la fonction récursive suivante qui implémente l'algorithme de recherche dichotomique détaillé ci-dessus. On pourra utiliser des slicings judicieusement choisis.

```
def Dichotomie_récursive (x,L):
    if..... :
        return False
    p = .....          # indice de l'élément central
    if x == L[p]:
        return .....
    elif x < L[p]:
        return Dichotomie_récursive (..... , .....)    # appel récursif
    else:
        return Dichotomie_récursive (..... , .....)    # appel récursif
```

Récursivité et algorithme de tri : le tri rapide

L'idée du tri rapide (ou Quicksort) , inventée par Tony Hoare, d'une liste L est de choisir un **pivot** (typiquement le 1er élément de la liste L[0]), de parcourir la liste L et de séparer la liste en 3 : les éléments plus petits **strictement** que le pivot, le pivot et les éléments égaux ou plus grands que le pivot.
Une solution est de créer deux sous listes Lg et Ld (qui vont contenir respectivement les éléments plus petits **strictement** que le pivot et les éléments égaux ou plus grands que le pivot), de les trier **récursivement** puis reconstituer l'ensemble par **concaténation**.



Exercice 3. Écrire une fonction **separe(L)** qui prend en argument une liste L supposée non vide.
En notant x le premier élément de L, votre fonction doit renvoyer deux listes Lg et Ld telle que Lg contient les éléments de L qui sont inférieurs ou égaux à x (à l'exception du premier élément), et Ld ceux qui sont strictement supérieurs à x.
Ainsi, separe([6, 7, 1, 6, 9, 2]) doit renvoyer ([1, 6, 2], [7, 9]).
On remarquera en particulier que seul le deuxième 6 figure dans la liste.

Exercice 4. Écrire une fonction récursive **Tri_rapide(L)** qui renvoie la liste L triée dans l'ordre croissant.