Récursivité

1 - Récursivité simple

a) La fonction factorielle

En informatique, les fonctions **récursives** sont des fonctions dont le calcul nécessite d'invoquer la fonction elle-même. Pour bien comprendre comment fonctionne une fonction récursive, en préambule, on peut tester la fonction factoriel ci-dessous

```
1  def fact(n):
2      print("Entree dans f("+ str(n) +")")
3      if n == 0:
4          res = 1
5      else:
6          res = n * fact(n - 1)
7      print("Sortie de f("+ str(n) +") renvoie " + str(res))
8      return res
```

2 - Calculs sur une liste

Ex. 1 :

Q 1 - Proposer une fonction `somme(L)` qui renvoie la somme des éléments d'une liste L. de manière récursive.

Q 2 - Proposer une fonction `Min(L)` qui renvoie le minimum des éléments d'une liste L. de manière récursive.

Pour les guerriers :

Q 3 - Pour une liste sans structure identifiée par exemple : `L = [[1],[3,4],[6,7,8],[5] ]`, proposer une fonction `somme(L)` qui renvoie la somme de tous les éléments. On utilisera la fonction `isinstance(L,list)` qui renvoie `True` si `L` est une liste et `False` sinon.

Q 4 - proposer une fonction non récursive équivalente à la précédente.

Q 5 - Écrire une fonction `permutation(L)` , sous forme récursive, permettant d'obtenir l'ensemble des permutations possibles d'un n-uplet de `L=[e0,e1,...,en]`.

Par exemple, `permutation([1,2,3])` doit retourner [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

3 - Affichage de triangles

Ex. 2 :

Écrire une fonction récursives `triangle(n,aff)` qui prend un paramètre entier `n` et un accumulateur `aff` qui permet d'afficher un triangle comme indiqué ci-dessous.

```
1  >>> triangle(6,'*')
2  *
3  **
4  ***
5  ****
6  *****
7  ******
```

4 - La fonction puissance

La fonction `puissance_iter(x, n)` définie ci-dessous renvoie x^n :

```
1  def puissance_iter (x, n):
2      res = 1
3      if n == 0:
4          return res
5      else:
6          for i in range(n):
7              res =  res * x
8          return res
```

AM, JB, NG, SR

Ex. 3 : *Puissance*

Q 1 - Réécrire cette fonction de manière récursive.
On rappelle qu'il est possible d'utiliser l'exponentiation rapide pour calculer la puissance d'un nombre :

— si n est pair alors : $x^n = \left(x^2\right)^{n//2}$

— si n est impair alors : $x^n = x \times x^{n-1} = x \times \left(x^2\right)^{n//2}$

Q 2 - Écrire un fonction récursive utilsant l'exponentiation rapide.
Q 3 - Comparer les complexités. On pourra intégrer un compteur d'appels de fonction.

a) Autres exemples usuels

Ex. 4 : *Combinaisons*

Écrire une fonction récursive `C(n,p)` qui associe à deux entiers n et p la valeur de $\binom{n}{p}$
On utilisera la formule du triangle. On prendra le temps de réfléchir au(x) point(s) d'appui de la récursivité.

Ex. 5 : *Test de parité*

Écrire deux fonctions `pair(N)` et `impair(N)` prenant en argument un entier N, renvoyant `True` ou `False` selon la parité (l'imparité) de N.
On pourra remarquer qu'un nombre N est pair si $N-1$ est impair et qu'il est impair si $N-1$ est pair.

5 - De mauvaises fonctions récursives

Ex. 6 :

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$u_0 = 2 \text{ et } u_{n+1} = \frac{1}{u_n} + u_n + 1$$

1. Que dire de cette fonction ?

```
1  def eval_u(n):
2      if n == 0:
3          return 2
4      else:
5          return 1 / eval_u(n-1) + eval_u(n-1) + 1
```

2. Ecrire une fonction `eval_u_2(n)` qui fait le même calcul en réduisant le nombre d'appels récursifs.
3. Ajouter un compteur global dans chaque fonction et tester le nombre d'appels récursifs pour chaque fonction lorsque $n = 50$.

Ex. 7 : *Suite de Fibonacci*

La suite de Fibonacci est définie par :

$$u_0 = 0,\ u_1 = 1 \text{ et } u_{n+2} = u_{n+1} + u_n$$

Q 1 - Pour commencer, proposer une fonction itérative `fibo_iter(n)`.
Q 2 - Proposer une version récursive `fibo_rec(n)`.
Q 3 - En s'appuyant sur le code donné en préambule, tracer l'arbre des appels de la fonction récursive.
Q 4 - En déduire la complexité.

6 - Récursivité avec listes et chaînes de caractères

Ex. 8 : *Recherche d'occurences dans une liste*

Q 1 - Écrire une fonction récursive `nb_occurences(L, n)` renvoyant le nombre d'occurences n de la liste L.
Si `Liste=[1, 0, 5, 0, 2]`, `nb_occurences(Liste, 0)` doit renvoyer 2.
Indice : Le nombre d'occurences de la liste est égal au nombre d'occurences de la liste privée de son premier terme plus 1 ou 0 en fonction de la valeur de ce terme.

Ex. 9 : *Palindrome*

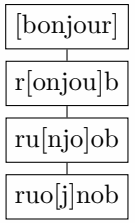
Q 1 - Écrire une fonction récursive `palindrome(s)` renvoyant `True` si la chaîne de caractères `s` est un palindrome (i.e. pouvant être lue indifféremment de la gauche vers la droite ou de la droite vers la gauche).
Les chaînes de caractères "ici", "kayak", "été" sont des palindromes.

Indice : La chaîne de caractères sera un palindrome si le premier et le dernier caractère sont identiques et si le reste de la chaîne de caractères (chaîne privée de son premier et dernier caractère) est un palindrome.

Ex. 10 : *Retourner une chaîne de caractères*

On souhaite réaliser une fonction récursive `retourner(s, g, d)` qui prend en arguments la chaîne de caractères `s` et ses indices minimal/maximal `g` et `d`, et qui réécrit cette chaîne à l'envers. Si la chaîne de caractères est *"bonjour"*, la fonction `retourner(s,0,6)` renvoie, *"ruojnob"*.

Le principe algorithmique est décrit ci-dessous et illustre les différentes permutations à réaliser au rythme des appels récursifs :



Q 1 - Écrire la fonction récursive `retourner(s, g, d)`.