MPSI - ITC - TD 8
Introduction à la récursivité

L'exemple classique que l'on prend pour illustrer la récursivité est le calcul de $n!$.
En mathématiques, la factorielle sur les entiers peut être définie par récurrence : $0! = 1$ et $\forall n \in \mathbb{N}, (n+1)! = n! \times (n+1)$
On peut implémenter la factorielle sur les entiers avec une boucle `for` comme suit :

```
def fact_boucle(n):
    resultat = 1
    for i in range(1,n+1):
        resultat = resultat * i
    return resultat
```

Main on peut implémenter la factorielle d'une façon beaucoup plus proche à sa définition mathématique :

```
def fact_recursive(n):
    if n==0:
        return 1
    else:
        return n*fact_recursive(n-1)
```

En effet, une fonction peut s'appeler elle-même, on parle alors de fonction **récursive**.
On décrit ci-après comment se déroule l'évaluation de l'appel `fact_recursive(4)`.
Je demande $4!$:

- 4 est différent de 0 - Je cherche $4 * 3!$ – je dois calculer $3!$
  - 3 est différent de 0 - Je cherche $3 * 2!$ – je dois calculer $2!$
    - 2 est différent de 0 - Je cherche $2 * 1!$ – je dois calculer $1!$
      - 1 est différent de 0 - Je cherche $1 * 0!$ – je dois calculer $0!$
        0 est égal à 0 - Je retourne 1
      - Je calcule $1 * 0! = 1 * 1 = 1$, je retourne 1
    - Je calcule $2 * 1! = 2 * 1 = 2$, je retourne 2
  - Je calcule $3 * 2! = 3 * 2 = 6$, je retourne 6
- Je calcule $4 * 3! = 4 * 6 = 12$, je retourne 12

(on pourra utiliser *python-tutor* pour visualiser cette exécution étape par étape).
D'une manière générale, les suites définies par récurrence se prêtent bien à l'utilisation de la récursivité. Voici l'exemple de la suite de Fibonacci :

```
def fibonacci(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

On remarquera que l'on peut faire plusieurs appels récursifs au sein du même calcul. Cependant, on verra l'année prochaine que cela peut avoir des conséquences très limitantes sur la complexité de la fonction en question.

Définition : Une fonction est dite *récursive* si elle s'appelle elle même. On parle alors d'appels récursifs. On leur oppose les fonctions dites *itératives*.

Les trois principes généraux : Si une fonction récursive est pure (au sens où elle n'a pas d'effets de bord tels que modifier une variable globale), alors pour s'arrêter sur toutes les entrées elle doit respecter les 3 principes suivant :

1. Elle doit contenir un ou plusieurs cas d'arrêts (des cas où elle n'a pas besoin d'appel récursif pour mener ses calculs à bout), sinon elle boucle indéfiniment.
2. Les valeurs en argument de la fonction lors d'un appel récursif doivent être différentes de celles de l'appel en cours, car sinon la fonction s'exécute de manière identique à chaque appel et donc boucle indéfiniment.
3. Après un nombre fini d'appels, la ou les valeurs passées en argument doivent permettre de valider la condition d'arrêt.

Remarques : Dans le cas de la fonction `fact_recursive` de la page précédente, les calculs ne peuvent pas être effectués avant d'obtenir les valeurs renvoyées par les appels récursifs, ce qui sollicite la mémoire de la machine pour stocker les instructions en attente. On parle de *récursivité profonde*.
Ce n'est pas le cas pour la fonction qui suit : on parle alors de *récursivité terminale*.

```
def affiche_jusqu_a_dix(k):
    if k<10:
        print(k)
        affiche_jusqu_a_dix(k+1)
```

Définition On dit qu'un appel récursif est en **position terminale** s'il est directement renvoyé par la fonction, sans calculs supplémentaires.

Définition On dit d'une fonction récursive qu'elle est **terminale** si tous ses appels récursifs sont en position terminale.

Stratégie « Diviser pour régner » On applique souvent un algorithme récursif dans le cas d'une stratégie, dite « diviser pour régner » qui consiste en les quatre éléments suivants :

— Si le problème à traiter est suffisamment simple, on le traite directement (**cas d'arrêt**).
— Sinon, on divise le en sous-problèmes plus petits (étape **diviser**),
— on résout les sous-problèmes par des appels récursifs (étape **régner**)
— et on recompose les sous-solution en une solution globale (étape **rassembler**)

Exercice 1 (Échauffement)

On considère la fonction suivante :

```
def pair(n):
    while n>0:
        n=n-2
    return n==0
```

(1) Que fait cette fonction ?
(2) Écrire une version récursive terminale de cette fonction.
(3) Écrire une fonction itérative `compte_a_rebours` qui prend en argument un entier n et renvoie une liste représentant le compte à rebours de n à 0. Par exemple, `compte_a_rebours(5)` doit renvoyer `[5, 4, 3, 2, 1]`.
(4) Faire la version récursive du compte à rebours. Est-elle terminale ?

Exercice 2 (La multiplication)

(5) Écrire une fonction itérative puis récursive qui prend en entrée un flottant x et un entier n et renvoie x^n en utilisant la définition : $x^0 = 1$ et $\forall n \in \mathbb{N}$, $x^{n+1} = x \times x^n$ et en n'utilisant pas les puissances bien entendu !
(6) Faire de même en utilisant la définition : $x^0 = 1$ et $\forall n \in \mathbb{N}$, si n est pair, avec $n = 2k$ alors $x^n = (x^k)^2$ et si n est impair, avec $n = 2k + 1$ alors $x^n = x(x^k)^2$
(7) Faire de même en utilisant la définition : $x^0 = 1$ et $\forall n \in \mathbb{N}$, si n est pair, avec $n = 2k$ alors $x^n = (x^2)^k$ et si n est impair, avec $n = 2k + 1$ alors $x^n = x(x^2)^k$
(8) Comparer le nombre d'appels lorsque $n = 64$ par exemple.

Exercice 3 (La suite de Syracuse)

La suite de Syracuse est définie par son premier terme u_0 et par la relation de récurrence : pour tout $n \geqslant 1$,

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & si \quad u_n \ est \ pair \\ 3u_n + 1 & sinon \end{cases}$$

(9) Écrire une fonction récursive qui prend en paramètres deux entiers n et $u0$ et renvoie le terme de rang n de la suite de Syracuse définie par le premier terme $u_0 = u0$.
(10) La conjecture de Collatz (1937) affirme que la suite finit toujours par donner un terme égal à 1. Elle devient alors périodique avec les valeurs 4, 2 puis 1. Écrire une fonction récursive `collatz` prenant en paramètre un entier naturel $u0$ et qui retourne le rang du premier terme de la suite qui est égal à 1 dans le cas où $u_0 = u0$.