TD3
Fonctions - Récursion

Exercice 1

1. Écrire une fonction récursive qui prend comme argument une valeur entière n et qui calcule la somme des n premiers entiers. Appeler cette fonction pour qu'elle calcule la somme des entiers allant de 0 à 100.

2. Transformer cette fonction afin qu'elle admette 2 arguments `debut` et `fin` et qu'elle calcule la somme des entiers variant de la valeur entière `debut` à la valeur entière `fin`. Appeler cette fonction pour calculer la somme des entiers variant de 50 à 100.

Correction

Une fonction récursive est une fonction qui s'appelle elle-même dans sa définition. Ainsi, la fonction fait un appel à elle-même, dans cet appel elle va à nouveau s'appeler elle-même, et ainsi de suite, créant une chaîne d'appels récursifs. Il est important de donner à la fonction des conditions d'arrêt qui remplissent 2 rôles :

- Donner les conditions limites du problème auquel notre programme répond ; par exemple, la somme des entiers de 0 à n vaut bien 0 quand $n = 0$, et aucune autre valeur.
- Empêcher que la chaîne d'appels récursifs soit infinie, ce qui nous retournerait l'erreur :

```
RecursionError: maximum recursion depth exceeded
```

On pensera donc toujours bien à définir pour une fonction récursive ses conditions d'arrêt et son (ou ses) appel(s) récursif(s).

1.
```python
def sommeRec(n):
    if n == 0:
        return 0
    return n + sommeRec(n-1)
print(sommeRec(100))
```

2.
```python
def sommeRecBornes(debut, fin):
    if debut == fin:
        return debut
    return fin + sommeRecBornes(debut, fin-1)
print(sommeRecBornes(50, 100))
```

Ou une autre solution équivalente :

```python
def sommeRecBornes2(debut, fin):
    if debut == fin:
        return debut
    return debut + sommeRecBornes2(debut+1, fin)
print(sommeRecBornes2(50, 100))
```

Exercice 2

La suite de Fibonacci est définie par :

$$\text{Fib}(0) = 0 \tag{1}$$
$$\text{Fib}(1) = 1 \tag{2}$$
$$\forall\, n \geq 2,\ \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \tag{3}$$

1. Programmer une fonction itérative calculant le n-ième nombre de Fibonacci.
2. Programmer une fonction récursive calculant le n-ième nombre de Fibonacci.
3. A l'aide de la bibliothèque `time` et de sa fonction `time()`, comparer les temps de calcul de ces deux fonctions pour `n=35` puis `n=70`. Que constatez-vous ?
4. Écrire et programmer une nouvelle fonction récursive ne faisant que n appels récursifs seulement. Mesurer le temps de calcul pour `n=35` puis `n=70`. Que constatez-vous ?

Correction

1. On doit au moins garder en mémoire à chaque étape les deux derniers nombres de Fibonacci calculés pour pouvoir calculer le suivant selon l'équation (3).

```
def FibIt(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    k = 2
    a, b = 0, 1
    while k <= n:
        tmp = a + b
        a = b
        b = tmp
        k += 1
    return b
```

Python permet de condenser le code dans la boucle pour écrire :

```
def FibIt2(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    k = 2
    a, b = 0, 1
    while k <= n:
        a, b = b, a+b
        k += 1
    return b
```

Enfin, une solution peut-être plus simple avec une liste :

```
def FibItList(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
```

```
    L = [0, 1]
    for k in range(2,n):
        L.append(L[-1]+L[-2])
    return L[-1]
```

2. ```
   def FibRec(n):
       if n == 0:
           return 0
       if n == 1:
           return 1
       return FibRec(n-1)+FibRec(n-2)
   ```

3. ```
   import time
   print("\t35\t\t\t70")
   t0 = time.time()
   FibIt(35)
   t1 = time.time()
   FibIt(70)
   t2 = time.time()
   print("FibIt\t"+str(t1-t0)+"\t"+str(t2-t1))
   t0 = time.time()
   FibRec(35)
   t1 = time.time()
   FibRec(70)
   t2 = time.time()
   print("FibRec\t"+str(t1-t0)+"\t"+str(t2-t1))
   ```

   Ici on a utilisé `"\t"` pour simple raison esthétique. Les `"\t"` sont des caractères spéciaux reconnus par Python pour faire des tabulations, c'est-à-dire des espaces plus grands que les espaces simples habituels. On se souvient que `"\n"` était lui un caractère spécial permettant d'effectuer un retour à la ligne. Il se peut que cela ne rende pas normalement dans le shell de Pyzo.

4. ```
   def Fib2Rec(n):
       return Fib2RecLoop(n,0,1)

   def Fib2RecLoop(n, a, b):
       if n == 0:
           return a
       if n == 1:
           return b
       return Fib2RecLoop(n-1,b,a+b)

   print("\t35\t\t\t70")
   t0 = time.time()
   Fib2Rec(35)
   t1 = time.time()
   Fib2Rec(70)
   t2 = time.time()
   print("Fib2Rec\t"+str(t1-t0)+"\t"+str(t2-t1))
   ```

   Cette solution fonctionne sur la même idée que notre première solution itérative ; l'idée est de garder en mémoire les deux derniers nombres de Fibonacci calculés, `a` et `b`, pour calculer le suivant qui vaut `a+b`. Les deux nouveaux derniers nombres sont donc `b` et `a+b`.

Exercice 3

Écrire une fonction récursive qui prend en argument une liste d'entiers `t` et qui calcule et affiche un entier de la façon suivante : à partir de `t`, on calcule la liste des différences de 2 éléments consécutifs de
   ```

`t` (à savoir `t[i+1]-t[i]`), et on recommence sur cette liste des différences jusqu'à obtenir une liste d'un seul élément. Par exemple, partant de la liste [3,5,10], la fonction doit afficher 3 (en passant par le calcul de la liste intermédiaire [2,5]).

Correction

```
def calcDiff(t):
    # La condition d'arret : liste de longueur 1, il n'y a plus de differences a
    faire
    if len(t) == 1:
        return t[0]

    # Calcul de la liste des differences
    diff = []
    for k in range(len(t)-1):
        diff.append(t[k+1]-t[k])

    # calcDiff() s'appelle elle-meme pour effectuer a nouveau l'operation sur la
    nouvelle liste obtenue
    return calcDiff(diff)

t = [3, 5, 10]
print(calcDiff(t))
```

Exercice 4

1. Écrire une fonction prenant un entier a et un entier positif n et qui retourne a^n de manière itérative.
2. Écrire une fonction récursive calculant cette même puissance.
3. Sachant que $a^0 = 1$ et

$$a^n = \begin{cases} (a^{(n/2)})^2 & \text{si } n \text{ est pair} \\ a \times (a^{(n-1)/2)})^2 & \text{si } n \text{ est impair} \end{cases}$$

   écrire une deuxième fonction récursive plus maline calculant cette même puissance.
4. Afficher les puissances de 2 de 2^0 à 2^{30} en appelant les deux fonctions récursives ci-dessus. On affichera également pour chaque puissance le nombre d'appels à chaque fonction, comptabilisés à l'aide d'une variable globale.

Correction

1. ```
   def powerIt(a, n):
       mult = a
       for k in range(n-1):
           mult *= a
       return mult
   ```

2. ```
   def powerRec(a, n):
       global c
       c += 1
       if n == 0:
           return 1
       return a*powerRec(a, n-1)
   ```

   C'est exactement la même idée qu'avec la somme de l'exercice 1, mais avec des multiplications.

3.
```
def power2Rec(a, n):
    global c
    c += 1
    if n == 0:
        return 1
    if n%2 == 0:
        p = power2Rec(a, n//2)
        return p*p
    else:
        p = power2Rec(a, (n-1)//2)
        return a*p*p
```

L'opérateur modulo `%` donne le reste de la division euclidienne de `x` par `y` si on écrit `x%y`.

4.
```
a, N = 2, 30
print("  Rec1")
for k in range(N+1):
    c = 0
    print(' ', powerRec(a,k), "\t", c)
print("\n  Rec2")
for k in range(N+1):
    c = 0
    print(' ', power2Rec(a,k), "\t", c)
```