TP Python 8 - Récursivité

Les fonctions demandées dans les exercices suivants ne doivent utiliser ni boucle `for`, ni boucle `while` (sauf à l'exercice 7).

Attention lorsqu'on programme une fonction récursive à ne pas oublier le cas d'arrêt.

Exercice 1 Écrire une fonction récursive `somme_harmonique` qui, recevant un entier naturel non nul n, renvoie la valeur de $H_n = \sum_{k=1}^{n} \frac{1}{k}$.

```
>>> somme_harmonique(1)
1
>>> somme_harmonique(2)
1.5
>>> somme_harmonique(100)
5.187377517639621
```

Indication : utiliser le fait que $H_n = H_{n-1} + \frac{1}{n}$.

Exercice 2 On rappelle la définition de la suite de Syracuse vue au TP 2. Un entier naturel n non nul est donné. S'il est pair, on le divise par 2. S'il est impair, on le multiplie par 3 et on ajoute 1. Puis on recommence : si le nombre obtenu est pair, on le divise par 2, et s'il est impair on le multiplie par 3 et on ajoute 1. On continue ainsi jusqu'à ce qu'on arrive à 1.

Écrire une fonction récursive `syracuse` qui, recevant un entier naturel non nul n, lui applique l'algorithme et affiche la suite des nombres obtenus jusqu'à ce qu'on arrive à 1.

```
>>> syracuse(3)
3 10 5 16 8 4 2 1
```

Exercice 3 Écrire une fonction récursive `bonjour` qui, recevant un entier naturel n, affiche n fois la chaîne `'Bonjour'`.

```
>>> bonjour(4)
Bonjour
Bonjour
Bonjour
Bonjour
```

Exercice 4 Écrire deux fonctions récursives `etoiles1` et `etoiles2` qui, recevant un entier naturel n, affichent un triangle d'astérisques comme dans les exemples suivants.

```
>>> etoiles1(5)
*****
****
***
**
*
```

```
>>> etoiles2(5)
*
**
***
****
*****
```

Exercice 5 La suite de Fibonacci est définie par $F_1 = 1$, $F_2 = 1$ et, pour tout $n \in \mathbb{N}^*$, $F_{n+2} = F_{n+1} + F_n$.

1) Écrire une fonction récursive `fibonacci` qui, recevant un entier naturel n non nul, renvoie la valeur de F_n.

```
>>> fibonacci(10)
55
```

2) Combien d'appels récursifs sont exécutés pour calculer `fibonacci(3)` ? Pour calculer `fibonacci(4)` ? Pour calculer `fibonacci(5)` ?

3) Calculer `fibonacci(20)`, `fibonacci(30)` et `fibonacci(40)`.

4) Pour expliquer le phénomène précédent, on va introduire une variable globale `N` qui compte le nombre d'appels effectués. Modifier la fonction `fibonacci` de telle sorte qu'elle ajoute 1 à `N` lorsqu'elle est exécutée (ne pas oublier d'ajouter `global N`).

```
>>> N = 0
>>> fibonacci(10)
55
>>> N
109
```

Combien d'appels récursifs sont exécutés pour calculer `fibonacci(20)`, `fibonacci(30)`, `fibonacci(40)` ? Ne pas oublier de remettre `N` à zéro à chaque fois.

5) Pour éviter l'explosion du nombre d'appels récursifs, on peut mémoriser les résultats des calculs au moment où ils sont effectués : c'est ce qu'on appelle la **mémoïsation**.

Dans le cas de la suite de Fibonacci, on va définir un dictionnaire (qui sera une variable globale) dans lequel on va stocker les valeurs de la suite.

```
d = {1: 1, 2: 1}
```

Récrire la fonction récursive `fibonacci` de telle sorte que si `d[n]` existe, `fibonacci(n)` renvoie `d[n]`, et sinon elle calcule sa valeur et la stocke dans le dictionnaire avant de la renvoyer.

```
>>> fibonacci(10)
55
>>> d
{1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
>>> fibonacci(100)  # doit être instantané
354224848179261915075
```

Exercice 6 Écrire une fonction récursive `binomial` qui, recevant deux entiers n et p, renvoie l'entier $\binom{n}{p}$. On utilisera la mémoïsation comme dans l'exercice précédent.

```
>>> binomial(100, 50)
100891344545564193334812497256
```

Exercice 7 Le produit cartésien des listes $L_1, \ldots, L_n$ est la liste contenant toutes les listes de la forme $[x_1, \ldots, x_n]$ où x_1 est dans L_1, …, x_n est dans L_n. Par exemple :

– Le produit cartésien des listes `['a', 'b']` et `[1, 2, 3]` est

`[['a', 1], ['a', 2], ['a', 3], ['b', 1], ['b', 2], ['b', 3]]`.

– Le produit cartésien des listes `['a', 'b']`, `[1, 2, 3]` et `['x', 'y']` est

```
[['a', 1, 'x'], ['a', 1, 'y'], ['a', 2, 'x'], ['a', 2, 'y'], ['a', 3, 'x'],
 ['a', 3, 'y'], ['b', 1, 'x'], ['b', 1, 'y'], ['b', 2, 'x'], ['b', 2, 'y'],
 ['b', 3, 'x'], ['b', 3, 'y']].
```

1) S'il n'y a que deux listes, on peut facilement construire leur produit cartésien à l'aide de deux boucles imbriquées. Écrire une fonction `produit_cartesien_2` qui, recevant deux listes, renvoie leur produit cartésien.

```
>>> produit_cartesien_2(['a', 'b'], [1, 2, 3])
[['a', 1], ['a', 2], ['a', 3], ['b', 1], ['b', 2], ['b', 3]]
```

2) Pour un nombre quelconque de listes, on ne peut plus utiliser la méthode précédente. On peut procéder de la manière récursive suivante. Soient $L_1, \ldots, L_n$ des listes.

– On calcule le produit cartésien P des listes $L_1, \ldots, L_{n-1}$.

– On construit à l'aide de deux boucles imbriquées la liste des $[x_1, \ldots, x_n]$ où $[x_1, \ldots, x_{n-1}]$ est un élément de P et x_n un élément de L_n.

Écrire une fonction `produit_cartesien` qui, recevant une liste de listes, renvoie leur produit cartésien.

```
>>> produit_cartesien([])
[[]]
>>> produit_cartesien([['a', 'b'], [1, 2, 3]])
[['a', 1], ['a', 2], ['a', 3], ['b', 1], ['b', 2], ['b', 3]]
>>> produit_cartesien([['a', 'b'], [1, 2, 3], ['x', 'y']])
[['a', 1, 'x'], ['a', 1, 'y'], ['a', 2, 'x'], ['a', 2, 'y'], ['a', 3, 'x'],
 ['a', 3, 'y'], ['b', 1, 'x'], ['b', 1, 'y'], ['b', 2, 'x'], ['b', 2, 'y'],
 ['b', 3, 'x'], ['b', 3, 'y']]
```