TP 2 - Optimisation des performances.

Décorateurs

Exercice 1

Écrivez un décorateur `comptable`, qui affiche le numéro de l'appel actuel de la fonction. Par exemple :

```
>>> @comptable
... def hello_world():
...     print("Hello, world!")
...
>>> for i in range(5):
...     hello_world()
...
[appel 1] Hello, world!
[appel 2] Hello, world!
[appel 3] Hello, world!
[appel 4] Hello, world!
[appel 5] Hello, world!
```

Exercice 2

Écrivez un décorateur `profondeur`, qui affiche la profondeur récursive d'un appel de fonction avec ses paramètres. Par exemple :

```
>>> @profondeur
... def fibo(n):
...     return n if n < 2 else fibo(n-1) + fibo(n-2)
...
>>> n = 4
>>> print("Le ", n, "-ème nombre de Fibonacci est ", fibo(n), sep="")
[appel de fibo, profondeur 1, paramètres: 4]
    [appel de fibo, profondeur 2, paramètres: 3]
        [appel de fibo, profondeur 3, paramètres: 2]
            [appel de fibo, profondeur 4, paramètres: 1]
            [appel de fibo, profondeur 4, paramètres: 0]
        [appel de fibo, profondeur 3, paramètres: 1]
    [appel de fibo, profondeur 2, paramètres: 2]
        [appel de fibo, profondeur 3, paramètres: 1]
        [appel de fibo, profondeur 3, paramètres: 0]
Le 4-ème nombre de Fibonacci est 3
```

Exercice 3

Dans un moment de distraction, votre binôme de projet a oublié que la fonction de recherche dichotomique qu'il a programmée ne peut s'appliquer qu'à une liste triée. Pour régler les bugs qu'il a ainsi introduits dans votre projet, écrivez un décorateur qui intercepte *toutes* les listes communiquées à la fonction décorée et les trie avant d'exécuter cette fonction. Par exemple :

```
>>> # avant décoration
>>> def dicho(liste, elem):
...     # ... du code ...
...
>>> liste = [4, 2, 1, 7, 5, 3, 6]; x = 3
>>> print(dicho(liste, x))
None
```

```
>>> # après décoration
>>> @trier_listes
... def dicho(liste, elem):
...     # ... du code ...
...
>>> print(dicho(liste, x))
2
```

On suppose ici que la fonction de recherche sera utilisée à la manière d'un `in` (autrement dit : on veut savoir si l'élément est dans la liste, peu importe sa position), puisque la position renvoyée est celle de l'élément dans la version triée de la liste et non dans la liste d'origine.

Programmation parallèle

Pour chacun des exercices suivants, intégrez votre réponse à un programme qui implémente également la version séquentielle de l'algorithme demandé, et qui compare les temps d'exécution sur des données de taille suffisante pour remarquer les gains.

Exercice 4 (Produit matriciel)
Implémentez une version parallèle de l'algorithme naïf pour le produit matriciel. Le module standard `itertools` contient des fonctions qui peuvent vous être utiles (mais qui ne sont pas indispensables).

Exercice 5 (Recherche approximative)
Écrivez une fonction exploitant le parallélisme pour compter le nombre d'occurrences approximatives d'un mot dans un texte. Par "occurrence approximative", on veut dire que le mot cherché et le mot trouvé, de même longueur, diffèrent en au plus k (fixé) positions. Par exemple :

```
>>> nombre_occurrences_approx("bon", "concombre", 2)
3
```

En effet, si l'on passe en revue toutes les sous-chaînes de taille `len("bon")` de `"concombre"`, on a :

```
'con'  # 1 différence  -> 1
'onc'  # 3 différences
'nco'  # 3 différences
'com'  # 2 différences -> 2
'omb'  # 3 différences
'mbr'  # 3 différences
'bre'  # 2 différences -> 3
```