TP 8
Tri

I Fonctions préliminaires

On reprend les fonctions suivantes du TP 6 (algorithmes sur les listes) car ce sont les bases extrêmement importantes. Toutes les listes considérées sont constituées de nombres (entiers ou flottants).

Exercice 1. Écrire une fonction `maximum(L)` qui retourne le maximum de la liste `L`.
Méthode : dans une boucle en itérant sur l'indice `i`, une variable `M` contient le maximum des `i` premiers termes de la liste.

Remarque 1. On peut traduire ce programme par une preuve mathématiques par récurrence du théorème suivant : tout ensemble fini et non-vide à n éléments $\{x_1, \dots, x_n\} \subset \mathbb{R}$ admet un maximum.

- Pour $n \geqslant 1$, notons $P(n)$ la proposition « toute partie finie A de $\mathbb{R}$ à n éléments admet un maximum ». On démarre à $n = 1$ car pour $n = 0$ c'est une partie vide et la partie vide n'a pas de maximum (elle est majorée, mais ne contient aucun élément).
- Initialisation : pour $n = 1$ alors $A = \{x_1\}$ et donc x_1 est le maximum de A.
- Hérédité : soit $n \in \mathbb{N}$, supposons $P(n)$. Soit $A = \{x_1, \dots, x_{n+1}\}$ une partie de $\mathbb{R}$ à $n + 1$ éléments. Formons alors $B = \{x_1, \dots, x_n\}$: c'est bien une partie de $\mathbb{R}$ à n éléments et donc par l'hypothèse de récurrence B admet un maximum M, qui est l'un des éléments x_i pour $1 \leqslant i \leqslant n$. Alors le maximum de A sera le plus grand des deux nombres entre M et x_{n+1} : c'est bien un élément de A, et il est plus grand que tous les éléments de A.

Exercice 2. Écrire une fonction `deuxieme_max(L)` qui retourne le *deuxième maximum* de `L`, c'est à dire l'avant dernier élément si `L` était rangée en ordre croissant.
Méthode : la liste doit avoir au moins deux éléments, et en parcourant la liste un tuple `(M, m)` contient le premier et le deuxième maximum « jusque là » comme dans la fonction précédente.

Exercice 3. Écrire une fonction `croissante(L)` qui teste (= retourne `True` ou `False`) si la liste est rangée par ordre croissant.
Méthode : à quelle condition une liste n'est-elle pas croissante ?

II Algorithmes de tri

Le but est d'écrire de différentes façons des fonctions qui vont prendre comme argument une liste de nombres entiers et trier la liste par ordre croissant. Les règles sont les suivantes :

1. La fonction prend en argument une liste `L` et rien de plus.
2. Elle utilise nécessairement des itérations sur les indices, compare des éléments `L[i]` et `L[j]`.
3. Éventuellement elle échange des éléments.
4. À la fin la liste doit être triée (mais on n'a pas besoin d'utiliser la fonction `croissante()` : la fonction termine quand on a compris qu'elle terminait, et pas parce que la fonction `croissante()` l'a dit) et la fonction retourne `L`.

Ainsi il est **interdit** d'utiliser les fonctions Python pour insérer ou supprimer en milieu de liste, ainsi que évidemment les fonctions déjà prêtes de tri. On n'a même pas besoin de `append` et `pop`, ni même des sélections d'intervalles d'éléments.
Pour diverses raisons expliquées en annexe, nous écrirons nos fonctions de la façon suivante :

```
def tri(L):
    # les opérations effectuées ici modifient L directement
    # pas besoin de faire M = L
    return L
```

et normalement à la fin `L` est triée, la fonction modifie la liste elle-même. On rappelle qu'on peut faire des tests en écrivant directement dans le mode script nos propres exemples de test (sinon, c'est long à recopier) :

```
##
def tri(L):
    ...
```

```
test = [5, 1, 3]
tri(test)
print(test)
##
```

et si on est courageux on peut aussi faire des tests sur des listes aléatoires. Attention alors à bien sauvegarder la liste dans une variable, et à l'afficher avant et après le tri (sinon, comment savoir si cela a réussi ?) :

```
##
def tri(L):
    ...

from random import randint
test = [randint(1, 100) for _ in range(10)]
print(test)
tri(test)
print(test)
##
```

Passons à l'étude d'un premier exemple.

II.1 Tri à bulles

C'est le plus simple à programmer. On itère sur les indices, dans l'ordre, et on compare deux éléments successifs. Soit ils sont bien dans l'ordre croissant. Soit non, et on les échange. On répète ce processus plusieurs fois, en repartant du tout début. Combien de fois ? À vous de voir... l'idée est que les plus grands éléments remontent peu à peu à la fin de la liste, comme des bulles qui remontent à la surface ; la fin de la liste est donc de plus en plus triée.

Exercice 4. Écrire la fonction `tri_bulle(L)`.

II.2 Tri par sélection

On part des *indices*, et on sélectionne l'élément qui doit aller à cet indice. Ainsi on cherche d'abord le minimum de `L` (son indice, pas sa valeur) puis on le place en premier (on l'échange avec l'élément d'indice 0, donc). Puis sur la liste des $n-1$ éléments restants, on cherche encore le minimum, et on le place en deuxième position dans la liste. Ainsi dans une boucle les `i` premiers éléments de la liste sont bien triés et on sélectionne à chaque fois l'élément devant aller à la place `i+1`.

Exercice 5. Écrire la fonction `tri_selection(L)`.

II.3 Tri par insertion

C'est celui qu'on fait le plus naturellement quand on peut *insérer* les objets uns par uns précisément là où ils doivent être. Cependant, malgré le nom, nous n'allons rien insérer du tout : ce sont seulement des échanges d'éléments dans la liste.
Ainsi on parcourt la liste, et à chaque élément rencontré `x` d'indice `i`, on repart du début de la liste pour trouver l'indice `j` où doit être placé `x`, puis on l'insère à sa place en décalant tous les termes d'indices entre `j` et `i` d'un cran vers la droite. C'est donc le début de la liste qui est de plus en plus trié. Au départ l'élément d'indice 0 reste à sa place, puis on prend l'élément d'indice 1 et on l'insère (soit on l'échange avec celui d'indice 0, soit il reste à sa place) puis on s'occupe de celui d'indice 2 et ainsi de suite. Attention à l'ordre des opérations !

Exercice 6. Écrire la fonction `tri_insertion(L)`.

II.4 Tri par comptage

Ce n'est pas vraiment un tri.
On suppose qu'on a une liste `L` contenant **uniquement des nombres entiers** entre 0 et un entier N (au sens large, bornes incluses). Alors plutôt que de trier les éléments on se pose la question de combien de fois on trouve chaque nombre. À partir de cette donnée il sera alors facile de reconstituer la liste triée.

Exercice 7.
1. Écrire la fonction `compte(L, N)` qui prend en argument une liste `L` et un entier `N`, en supposant que les éléments x de `L` vérifient tous $0 \leqslant x \leqslant N$, et qui retourne une liste `C` où `C[i]` est le nombre d'élements de `L` égaux à `i`.
2. Puis écrire une fonction `tri_comptage(L, N)` qui utilise la fonction précédente et retourne une liste triée `M` reconstruite à partir de la donnée précédente.

Cela a cependant quelques applications :

Exercice 8. En utilisant les fonctions précédentes, écrire la fonction `mediane(L, N)` qui calcule la médiane.

On remarque facilement que la somme des éléments de `C` est égal à la longueur de `L`...

II.5 Tri stupide

On choisit deux indices au hasard, et on compare les éléments. Et éventuellement on les échange. On répète cela beaucoup de fois. Pour savoir quand on s'arrête, on utilise éventuellement la fonction `croissante()` ou alors on fixe un nombre `N` de répétitions passé en argument à la fonction. On a évidemment besoin de la bibliothèque `random` et da sa fonction `randint()` (ou même `randrange()` qui dans ce contexte est un peu plus claire).

Exercice 9. Écrire la fonction `tri_stupide(L)`.

III Annexe : un problème ennuyeux (ou pas)

Comparer les morceaux de programmes suivants (on peut essayer en mode interactif, pour voir le résultat pas à pas) :

```
x = 3
y = x
x = 12
print(y)
```

et

```
L = [1, 3, 5]
M = L
L[0] = 25
print(M)
```

et aussi

```
s = "Bonjour"
t = s
s[0] = "b"
print(t)
```

Que se passe-t-il ?
On dira que dans le cas des variables de type `int`, chaque variable **contient** une valeur, et lors de l'affectation `y = x` la valeur de `x` est **copiée** dans `y`. Mais dans le cas des listes, ce n'est pas **toute la liste** qui est copiée mais un « moyen d'accéder à la liste » (on dit une **référence**) et ainsi après l'instruction `M = L` alors `M` devient **une référence à la *même* liste** que `L`. En effet, recopier une liste est une opération beaucoup plus lourde (la liste pourrait avoir des milliers d'éléments) que de recopier un seul nombre ! Quant aux chaines de caractère, ce sont des objets **immutables** que l'on ne peut de toute façon pas modifier : la question de recopier ne se pose même pas.
En cas de besoin de copie, on peut utiliser la méthode `L.copy()` :

```
L = [1, 3, 5]
M = L.copy()
L[0] = 25
print(M)
# M est bien une liste différente de L, obtenue par copie
```

Remarque 2. La fonction Python `id()` permet de connaitre *l'identité unique* d'une variable, un nombre se comportant comme un identifiant unique associé au contenu de la variable. Ainsi dans les cas précédents

```
L = [1, 3, 5]
M = L
print(id(L))
print(id(M)) # ce sont bien deux références sur la même liste
N = L.copy()
print(id(N)) # c'est une autre liste, avec le même contenu
```

Cela a toute son importance dans les questions sur les fonctions car pour la même raison, si une fonction prend en argument une liste `L` alors elle peut modifier la liste `L`. Par exemple on écrira ainsi la fonction qui double tous les éléments de `L` :

```
def double(L):
    for i in range(len(L)):
        L[i] = L[i] * 2
    return L
```

et si `L = [1, 3, 5]` alors il suffit d'écrire `double(L)`, sans avoir besoin d'écrire `L = double(L)`.
. . . en fait cela marche même sans `return L` à la fin mais si on ne le met pas on ne peut pas écrire simplement quelque chose comme `double([1, 3, 5])` (ceci modifie la liste mais la perd ensuite : rien ne s'affiche), et même si on écrit juste `double(L)` alors `L` est modifiée mais rien n'est affiché en retour. Donc à la fois on modifie `L` sans copie, et à la fois on retourne `L`.