TP 8
Tri

Dans ce TP nous nous intéressons au problème du tri des listes. Il s'agit tout simplement de se donner une liste de nombres et d'étudier différentes méthodes pour les ranger par ordre croissant, en échangeant des éléments entre eux. Par exemple, trier la liste `L = [3, 4, 3, 4, 1, 1, 9, 3]` doit donner `[1, 1, 3, 3, 3, 4, 4, 9]`.

I Préliminaires

Les fonctions de cette section ne seront pas utilisées tel quel par la suite. Il s'agit cependant d'un échauffement et de bases à bien comprendre.
Le but final est d'aboutir à une liste triée, ce qui est la même chose qu'une liste rangée par ordre croissant.

Exercice 1. *Révisions*
Écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste `L` est bien rangée en ordre croissant, et `False` sinon.

Puis nous aurons besoin de diverses variantes pour obtenir le maximum d'une liste `L` — ou bien pour obtenir l'indice `i` tel que le maximum de `L` est `L[i]`. L'idée est de parcourir la liste en maintenant une variable M qui contient le maximum de la liste « jusque là ». Attention car le maximum d'une liste vide n'est pas défini, vraiment pas ! Le programme est particulièrement proche de la preuve mathématique, qu'on pourra lire avant d'écrire la fonction. C'est aussi la méthode qu'on appliquerait naturellement, en lisant la liste dans l'ordre et en gardant en tête le nombre le plus grand qu'on a rencontré.

Exercice 2. Écrire une fonction `maximum(L)` qui renvoie le maximum de la liste `L`.

Théorème 1. *Tout sous-ensemble fini et non-vide $A \subset \mathbb{R}$ admet un maximum.*

Démonstration. Démontrons par récurrence sur n la proposition $\mathcal{P}(n)$: « tout sous-ensemble fini à n éléments $A \subset \mathbb{R}$ admet un maximum ». Comme la partie A ne peut pas être vide, la récurrence porte sur $n \in \mathbb{N}^*$.

- Initialisation : pour $n = 1$, soit A une partie de $\mathbb{R}$ à un seul élément, alors on peut écrire $A = \{x_1\}$ avec $x_1 \in \mathbb{R}$ et x_1 est le maximum de A.
- Hérédité : soit $n \in \mathbb{N}^*$, supposons $\mathcal{P}(n)$. Montrons $\mathcal{P}(n+1)$. Soit une partie $A \subset \mathbb{R}$ à $n+1$ éléments, écrivons $A = \{x_1, \dots, x_n, x_{n+1}\}$. Formons la partie $B = \{x_1, \dots, x_n\}$, c'est une partie de $\mathbb{R}$ non-vide à n éléments. On applique alors l'hypothèse de récurrence à B, qui admet donc un maximum M, qui est un des éléments parmi $x_1, \dots, x_n$. Mais ensuite :
  1. Ou bien $x_{n+1} \geqslant M$. Alors x_{n+1} est plus grand que lui-même et que tous les $x_1, \dots, x_n$, donc x_{n+1} est le maximum de A.
  2. Ou bien $x_{n+1} \leqslant M$, et donc M est plus grand que tous les $x_1, \dots, x_n$ et aussi que x_{n+1} et donc M est le maximum de A.

  En conclusion la partie A admet bien un maximum, et ceci démontre $\mathcal{P}(n+1)$.

□

Pour la suite il est important de comprendre qu'une fonction qui reçoit une liste en argument va modifier la liste, comme expliqué dans l'annexe § III. La fonction suivante n'est en elle-même pas très intéressante mais illustre la façon dont seront écrites les fonctions pour la suite.

Exercice 3. Écrire une fonction `ordonne(L, i, j)` qui « ordonne » les éléments d'indices `i` et `j` ; c'est à dire les échange si `i < j` mais qu'on n'a pas `L[i]` $\leqslant$ `L[j]`, ou bien si `j < i` mais qu'on n'a pas `L[j]` $\leqslant$ `L[i]`. Ainsi à la fin les éléments d'indices `i` et `j` seront rangés par ordre croissant.

II Les algorithmes de tri

Nous démarrons maintenant les algorithmes de tri. Le but est, à chaque fois, d'écrire une fonction qui prend comme argument une liste de nombres (on les supposera entiers) de longueur n et trie la liste par ordre croissant. La fonction utilise diverses comparaisons entre des éléments d'indices `i` et `j` et éventuellement les échange, et à la fin la liste doit être triée. Normalement nous n'avons pas besoin d'utiliser la fonction `est_croissante`, car on peut savoir à l'avance qu'après un certain nombre de comparaisons la liste sera triée. Nous n'avons pas non plus besoin de `ordonne` car on ré-écrit directement les comparaisons et échanges dont on a besoin dans le corps de la fonction. Enfin on a toujours besoin d'une certaine forme de double boucle, pour parcourir la liste plusieurs fois.
On n'utilise donc pas 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 de `pop`, ni même des sélections de tranches d'éléments.
Il est encouragé de rajouter des instructions `print(L)` dans les boucles, pour voir la liste évoluer au fur et à mesure du tri.

II.1 Tri à bulles

C'est le plus simple des tris à programmer. L'algorithme du tri à bulles se décrit ainsi :

1. On parcourt la liste, tout simplement.
2. Si deux éléments consécutifs ne sont pas rangés dans l'ordre croissant, on les échange.
3. On répète ce processus n fois (en fait $n-1$ fois suffisent : pourquoi ?) en repartant à chaque fois du tout début de la liste.

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 de l'eau.

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

II.2 Tri par sélection

Il s'agit du deuxième plus simple des tris, et il est important d'avoir bien compris la fonction `maximum`.

1. On cherche **l'indice** du minimum de `L`, et par un échange on place le minimum à l'indice 0.
2. Puis on cherche l'indice du minimum de la liste, à partir de l'indice 1 (ce sera donc le deuxième plus petit), et de même par un échange on le place à l'indice 1.
3. On continue ce procédé jusqu'à arriver à la fin de la liste.

Ainsi, on **sélectionne** directement les éléments, uns par uns, pour les mettre à leur place, en partant du plus petit.

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

II.3 Tri par insertion

C'est celui qu'on fait le plus naturellement, par exemple quand on trie un jeu de cartes, en **insérant** uns par uns les éléments chacun à leur place. Cependant, malgré le nom, nous écrivons cette fonction sans utiliser des insertions dans une liste, ce qui nécessite à la place de faire des décalages d'éléments vers la droite.
L'algorithme peut donc se décrire ainsi :

1. On parcourt la liste simplement.
2. À chaque élément rencontré `x` d'indice `i`, on repart du début de la liste pour trouver l'indice `j` (avec $j \leqslant i$) où doit être placé `x`.
3. Pour insérer `x` à sa place on doit décaler vers la droite tous les éléments d'indice k avec $j \leqslant k < i$.
4. On répète tout cela jusqu'à avoir inséré le dernier élément de la liste.

Attention à l'ordre des opérations pour tout décaler vers la droite en échangeant des éléments deux par deux !

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

II.4 Tri stupide

Pour s'amuser uniquement, car ce tri est totalement inefficace. Le tri stupide fonctionne ainsi (là, nous pouvons ré-utiliser les fonctions de la section préliminaire) :

1. On choisit deux indices de la liste au hasard i et j.
2. Si les éléments d'indice i et j ne sont pas rangés dans l'ordre croissant, alors on les échange.
3. On répète *tant que* la liste n'est pas triée.

Pour choisir les indices au hasard, on a besoin de la bibliothèque `random` et de sa fonction `randint()` — quoique, ici `randrange(n)` est plus pertinent, qui donne un nombre au hasard tout comme `randint` mais avec une syntaxe similaire à `range`. Ainsi `randrange(n)`, aussi `randrange(0, n)`, est la même chose que `randint(0, n-1)`.

Exercice 7. Écrire la fonction `tri_stupide(L)`.

Observer aussi comme la fonction est nettement de plus en plus lente quand la longueur de la liste augmente, à un point où elle ne se termine même plus en un temps raisonnable. Quand la liste est très grande et presque triée, il devient très improbable de tomber pile sur deux éléments restant à échanger, et la boucle continue de nombreuses fois en attendant.

II.5 Tri par comptage

Il ne s'agit pas d'un tri au même sens que les précédents.
On suppose qu'on a une liste `L` contenant **uniquement des nombres entiers** entre 0 et un certain entier N (au sens large, bornes incluses). Au lieu de trier directement, nous allons d'abord compter combien de fois apparait chaque nombre, puis nous allons reconstruire la liste. Après coup, on n'a pas besoin de se donner N car il s'agit du maximum de la liste.

Exercice 8.
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 renvoie une liste `C` où `C[x]` est le nombre d'éléments de `L` qui sont égaux à `x`.
   (C'est la fonction qui s'appelait `compte_tout` dans le TP 6 : Algorithmes sur les listes.)
2. En déduire une fonction `tri_comptage(L)` qui utilise la fonction précédente et reconstruit une liste `M` qui contient les même éléments que `L` mais triée.

Cette méthode est notamment intéressante pour calculer la médiane d'une liste. Définissons la **médiane** d'une liste `L` comme le *plus petit* élément m de la liste tel qu'*au moins la moitié* des éléments de `L` soient *inférieurs ou égaux* à m. Ces précisions sont subtiles mais permettent d'avoir une médiane bien définie, que le nombre d'éléments soit pair ou non. Remarquons que lorsque la liste `L` est déjà triée, la médiane est immédiate à trouver… Remarquons aussi que la somme des éléments de `C` est égal à la longueur de `L`.

Exercice 9. Écrire la fonction `mediane(L)` qui calcule la médiane de la liste `L`, avec la définition ci-dessus :

1. Une première fois en triant la liste.
2. Une deuxième fois en utilisant `compte` mais sans trier la liste.

III Annexe : le problème des listes et des références

Il est important d'être conscient du problème suivant lorsqu'on manipule des listes. Comparons les trois morceaux de programmes :

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

et

```
>>> L = [1, 3, 5]
>>> M = L
>>> L[0] = 12
>>> print(M)
[12, 3, 5]
```

et aussi

```
>>> s = "Bonjour"
>>> t = s
>>> s[0] = "b"
TypeError: 'str' object does not support item assignment
```

Que se passe-t-il ?

1. Dans le premier cas, les variables de type `int` **contiennent** une valeur, et lors de l'affectation `y = x` la valeur de `x` est **copiée** dans `y`. Une modification ultérieure de `x` ne modifiera en rien `y`.
2. Dans le deuxième cas, lors de l'affectation `M = L`, ce n'est pas toute la liste qui est copiée mais un « moyen d'accéder à la liste » qu'on appellera une **référence** à `L`. Les noms de variables `L` et `M` sont donc des références à la **même** liste en mémoire et toute modification de l'une affecte l'autre. Recopier une liste en mémoire peut être une opération coûteuse car la liste peut contenir des milliers d'éléments, et donc il faut éviter de la recopier inutilement !
3. Enfin dans le troisième cas les chaines de caractères sont des objets **immuables** que l'on ne peut de toute façon pas modifier. Cela fait donc peu de différence de savoir si deux variables sont des références à la même chaine ou bien deux chaines différentes.

Ce phénomène se produit aussi lorsqu'on passe une liste en argument à une fonction :

```
def f(L):
    L[0] = 12
```

puis

```
>>> L = [1, 3, 5]
>>> f(L)
>>> print(L)
[12, 3, 5]
```

Cela peut être souhaité ou bien peut être embêtant. Les fonctions `append` et `pop` vont aussi modifier la liste passée en argument :

```
def f(L):
    L.append(12)
```

puis

```
>>> L = [1, 3, 5]
>>> f(L)
>>> print(L)
[1, 3, 5, 12]
```

On parle aussi d'**effets de bords** — la fonction a des effets en dehors de la manipulation de ses variables locales. Le mot-clé `is` permet de savoir si deux variables sont des références au même objet Python.

```
>>> L = [1, 3, 5]
>>> M = L
>>> M == L
True
>>> M is L
True
```

mais

```
>>> L = [1, 3, 5]
>>> M = [1, 3, 5]
>>> M == L
True
>>> M is L
False
```

Dans ce dernier cas les listes `L` et `M` sont **structurellement égales** (elles contiennent les mêmes éléments et sont donc des objets égaux au sens mathématique usuel du terme) mais ne sont pas des références à un même objet liste. Ainsi une modification de l'une ne va pas affecter l'autre.
Il est toujours possible d'obtenir une copie « fraiche » d'une liste, qui ne soit pas une référence mais contienne les même éléments, avec la méthode `L.copy()`.

```
>>> L = [1, 3, 5]
>>> M = L.copy()
>>> M
[1, 3, 5]
>>> M is L
False
```

Encore une fois, ce `False` nous indique qu'on peut sereinement modifier soit `L` soit `M` sans affecter l'autre.
Retenons que :

1. Les opérations `+`, `*`, et les sélections de tranches, créent à chaque fois des listes nouvelles en copiant les éléments.
2. Les affectations `L[i] = ...`, et les opérations `append` et `pop`, modifient directement les listes, même à travers des références. Attention aux effets de bords quand on les utilise sur des listes passées en argument à des fonctions !
3. L'opération `M = L.copy()` permet de créer une nouvelle liste `M`, et non pas une référence à `L`, en copiant les éléments.

Dans ce TP il y a un **choix** que les fonctions de tri modifient la liste elle-même. On pourrait aussi demander à ce que les fonctions ne modifient pas les listes, auquel cas elles devraient créer une nouvelle liste ou bien travailler dès le début avec une copie.