chapitre 8

Les Tris

Table des matières

Le problème de départ est simple : on veut écrire une fonction qui prend en entrée une liste et qui renvoie une liste contenant exactement les mêmes éléments (avec le même nombre de répétitions pour les éléments apparaissant plusieurs fois) mais triée par ordre croissant.

Ainsi, si l'on appelle notre fonction `tri`, on voudrait que `tri([3, 2, 5, 1, 3])` renvoie `[1, 2, 3, 3, 5]`.

I Tri d'une liste : les commandes existantes en Python

1.1 Tri sur place ou non

Une fonction de tri peut avoir deux comportements différents : soit elle crée une nouvelle liste triée qu'elle renvoie, soit elle modifie la liste qu'on lui a passé en argument (et dans ce cas là elle ne renvoie rien).

La fonction `sorted` de python est du premier type :

```
>>> t = [5,4,7,2,3,2,5,3]
>>> u = sorted(t)
>>> u
[2, 2, 3, 3, 4, 5, 5, 7]
>>> t
[5, 4, 7, 2, 3, 2, 5, 3]
```

Python fournit aussi une méthode qui a l'autre comportement : un appel à `t.sort()` modifie la liste `t` (en la triant), et ne renvoie rien. La syntaxe est assez différente de celle de `sorted`[1], mais similaire à celle de `append`.

Remarque : la différence entre `sorted(t)` et `t.sort()` est la même qu'entre `t + [x]` et `t.append(x)`.

```
>>> t = [5,4,7,2,3,2,5,3]
>>> t.sort()
>>> t
[2, 2, 3, 3, 4, 5, 5, 7]
```

1.2 Ordre croissant ou décroissant, données non numériques

Par défaut, Python trie par ordre croissant, mais l'on peut rajouter `reverse = True` comme argument de `sort` ou `sorted` pour préciser que l'on souhaite trier par ordre décroissant :

```
>>> t = [5,4,7,2,3,2,5,3]
>>> sorted(t, reverse = True)
[7, 5, 5, 4, 3, 3, 2, 2]
```

On peut trier des listes qui contiennent autre chose que des nombres : Python ordonne par exemple les chaînes de caractère par ordre alphabétique, les couples par ordre lexicographique...

```
>>> t = ["bonjour", "bon", "mauvais", "z", "B"]
>>> sorted(t)
['B', 'bon', 'bonjour', 'mauvais', 'z']
>>> t = [(1, 2), (0, 10), (1, 3), (2, 0)]
>>> sorted(t)
[(0, 10), (1, 2), (1, 3), (2, 0)]
```

1.3 Ordre personnalisé

Parfois, l'ordre choisi par défaut par Python n'est pas celui que l'on veut utiliser. Imaginons que l'on dispose d'une liste de triplets (s, x, y), où s est une chaîne de caractères (par un exemple un nom) et x et y deux valeurs numériques : `t = [("Bob", 10, 18), ("Alex", 3, 24), ("Max", 10, 12), ("Bob", 10, 12)]`

Par défaut, Python triera d'abord suivant s, puis à s égal suivant x et finalement suivant y :

```
>>> sorted(t)
[('Alex', 3, 24), ('Bob', 10, 12), ('Bob', 10, 18), ('Max', 10, 12)]
```

On peut souhaiter utiliser un autre ordre, par exemple classer suivant $x + y$. Dans ce cas, il faut utiliser un autre argument optionnel de `sorted`. Si l'on appelle `sorted(t, key = f)`, alors au lieu de comparer les éléments de t pour les classer , Python comparera les images des éléments par la fonction f. Ainsi, si l'on souhaite trie par valeurs croissantes de $x + y$, on procédera ainsi :

1. on dit que `sorted` est une fonction alors que `sort` est une méthode

```
>>> def f(triplet): return triplet[1] + triplet[2]
>>> sorted(t, key = f)
[('Max', 10, 12), ('Bob', 10, 12), ('Alex', 3, 24), ('Bob', 10, 18)]
```

Si l'on souhaite trier suivant les valeurs de x, on fera :

```
>>> def f(triplet): return triplet[1]
>>> sorted(t, key = f)
[('Alex', 3, 24), ('Bob', 10, 18), ('Max', 10, 12), ('Bob', 10, 12)]
```

II Algorithmes de tris

Il existe beaucoup d'algorithmes de tri. Vous pouvez aller voir sur la page Wikipédia consacrée aux tris les différentes sortes d'algorithmes. Une des choses qui différencie tous ces algorithmes c'est le nombre d'opérations réalisées (et donc le temps d'exécution) pour trier une liste. Vous pouvez aller voir sur le site Sorting Algorithms Animations la différence de vitesse d'exécution entre certains de ces algorithmes.

Nous allons dans ce qui suit découvrir deux algorithmes de tri, le tri par insertion et le tri à bulles.

2.1 Tri par insertion

Principe général. Si l'on dispose d'un tableau de taille n dont les k premiers éléments sont triés, on peut déterminer la place que doit occuper le $k+1$-ème élément dans les k premiers et l'y insérer. On répète cette opération jusqu'à $k = n$ et le tableau est alors trié.

2.1.1 Par échanges successifs

Plusieurs méthodes sont possibles pour insérer le $k+1$-ème élément à la bonne place, mais la plus simple est sans doute de procéder par échanges successifs d'éléments adjacents :

9	6	1	4	8	⟶	6	9	1	4	8									
6	9	1	4	8	⟶	6	1	9	4	8	⟶	1	6	9	4	8			
1	6	9	4	8	⟶	1	6	4	9	8	⟶	1	4	6	9	8			
1	4	6	9	8	⟶	1	4	6	8	9									

Cette méthode conduit naturellement à modifier l'argument de `tri_insertion(t)`, mais l'on peut bien sûr commencer par faire une copie si l'on préfère renvoyer une nouvelle liste.

Exercice 8.1 Compléter les lignes qui suivent à fin de programmer le tri par insertion tel qu'il a été décrit plus haut.

```
1 def tri_insertion(t):
2     for i in range(1,len(t)):
3         ...
4         while ...
5             ...
6             ...
```

Remarque : il n'est pas utile ici de mettre un `return` puisque la liste est modifiée sur place.

2.1.2 Une autre approche

Cette fois on va procéder comme la plupart des personnes font pour trier leur cartes : on prend les cartes mélangées une à une sur la table, et on forme une main en insérant chaque carte à sa place.

Étape	Liste initiale	Liste construite
0	9 6 1 4 8	liste vide
1	9 6 1 4 8	9
2	9 6 1 4 8	6 9
3	9 6 1 4 8	1 6 9
4	9 6 1 4 8	1 4 6 9
5	9 6 1 4 8	1 4 6 8 9

Exercice 8.2

1. Si `t = [3, 1, 5]`, que fait l'instruction `t[1:1] = [7]` ou `t.insert(1,7)` ?
2. Écrire une fonction `place(x, t)` qui prend en arguments une valeur x et une liste t supposée triée par ordre croissant et qui renvoie l'indice auquel il faut insérer x dans t si l'on veut que la liste obtenue reste croissante. Par exemple, on doit avoir :
   - `place(3, [8, 12, 15]) = 0`
   - `place(8, [8, 12, 15]) = 0` ou `place(8, [8, 12, 15]) = 1` (au choix)
   - `place(10, [8, 12, 15]) = 1`
   - `place(14, [8, 12, 15]) = 2`
   - `place(17, [8, 12, 15]) = 3`
3. Écrire une fonction `tri_insertion_2(t)` qui renvoie une nouvelle version de t, triée par insertion. On commencera par créer une liste vide dans laquelle on insérera les éléments de t au fur et à mesure en utilisant la fonction `place`.

2.2 Applications

Exercice 8.3 *Médiane* Écrire une fonction qui détermine la médiane des éléments d'une liste.

Exercice 8.4 *Recherche d'un élément*

1. *Le cas d'une liste quelconque*

   On redonne ici un algorithme vu lors d'un TP précédent qui permet de rechercher la présence ou non d'un élément dans une liste.

```
def appartient(liste,element):
    for x in liste:
        if x == element:
            return True
    return False
```

   Combien faudra-t-il de tests au maximum pour tester la présence d'un élément dans une liste contenant un milliard d'éléments ?

2. *Le cas d'une liste triée*

   Nous souhaitons accélérer le processus. Nous supposons maintenant que *la liste est triée*. Cette hypothèse supplémentaire va nous permettre d'aller beaucoup plus vite (imaginez que vous cherchez un mot dans le dictionnaire ou un nom dans l'annuaire : inutile de lire tous les mots ou tous les noms pour trouver celui que vous cherchez).

   En se basant sur les exemples suivants (les cases en gris foncé sont les cases éliminées de la recherche), écrire une fonction permettant de tester la présence ou l'absence d'un élément dans une liste *triée*.

   - Recherche de l'élément 44 dans la liste `[2,12,17,25,33,35,44,54,77,91]` :

Indice	0	1	2	3	4	5	6	7	8	9	10
Étape 1	2	12	17	25	33	35	44	54	77	91	
	g					m					d
Étape 2	2	12	17	25	33	35	44	54	77	91	
							g		m		d
Étape 3	2	12	17	25	33	35	44	54	77	91	
							g	m	d		
Étape 4	2	12	17	25	33	35	44	54	77	91	
							g,m	d			

- Recherche de l'élément 22 dans la liste `[2,12,17,25,33,35,44,54,77,91]` :

Indice	0	1	2	3	4	5	6	7	8	9	10
Étape 1	2	12	17	25	33	35	44	54	77	91	
	g					m					d
Étape 2	2	12	17	25	33	35	44	54	77	91	
	g		m			d					
Étape 3	2	12	17	25	33	35	44	54	77	91	
				g	m	d					
Étape 4	2	12	17	25	33	35	44	54	77	91	
				g,m	d						
Étape 5	2	12	17	25	33	35	44	54	77	91	
				g,d							

Combien faudra-t-il de tests au maximum pour tester la présence d'un élément dans une liste contenant un milliard d'éléments ?

2.3 Tri à bulles

Principe général. On parcourt le tableau, et on compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié. On arrête alors l'algorithme.

9	6	1	4	8	⟶	6	9	1	4	8	⟶	6	1	9	4	8	⟶	6	1	4	9	8	⟶	6	1	4	8	9

Le premier passage est fini

⟶	1	6	4	8	9	⟶	1	4	6	8	9

Le deuxième passage est fini.
Le troisième passage ne change rien, le tri est fini.

Exercice 8.5 Écrire une fonction `tri_bulles(t)` qui trie la liste t par ordre croissant en la modifiant par la méthode du tri à bulles.