Algorithmes de Tris

T.P. II

■ ■ ■

Partie I : Présentation des algorithmes

Dans ce T.P., nous proposons d'implémenter différents algorithmes de tri. Nous considérerons des listes de nombres entiers dont les éléments seront triés par ordre croissant. Pour chacun de ces tris, nous évaluerons sa complexité en nombre de comparaisons.

Le *tri par sélection* consiste à échanger le plus petit élément de la liste avec celui en première position, puis à recommencer avec les éléments restants.

10, **1**, 5, 19, 3, 3
1, 10, 5, 19, **3**, 3
1, 3, 5, 19, 10, **3**
1, 3, 3, 19, 10, **5**
1, 3, 3, 5, **10**, 19
1, 3, 3, 5, 10, 19

Le *tri par insertion* consiste à garder le début de la liste triée et à y insérer successivement, à leur place, les éléments restants.

10, **1**, 5, 19, 3, 3
1, 10, 5, 19, 3, 3
1, 10, **5**, 19, 3, 3
1, **5**, 10, 19, 3, 3
1, 5, 10, **19**, 3, 3
1, 5, 10, 19, **3**, 3
1, 5, 10, **3**, 19, 3
1, 5, **3**, 10, 19, 3
1, **3**, 5, 10, 19, 3
1, 3, 5, 10, 19, **3**
1, 3, 5, 10, **3**, 19
1, 3, 5, **3**, 10, 19
1, 3, **3**, 5, 10, 19

Dans l'algorithme de *tri rapide* (inventé par C. Hoare en 1961), on choisit un pivot (dans cet exercice le premier élément de la liste) et on sépare la liste en trois sous-listes : une première liste `l_inf` avec les éléments strictement inférieurs au pivot, une deuxième `l_egal` avec les éléments égaux au pivot et une troisième `l_sup` avec les éléments qui lui sont sont strictement supérieurs. L'algorithme est récursif et est ensuite appelé sur les sous-listes `l_inf` et `l_sup`.

{**10**, 1, 5, 19, 3, 3}
{1, 5, 3, 3},{**10**}, {19}
{**1**, 5, 3, 3}, {10}, {19}
{}, {**1**}, {5, 3, 3}, {10}, {19}
{}, {1}, {**5**, 3, 3}, {10}, {19}
{}, {1}, {3, 3}, {**5**}, {}, {10}, {19}
{}, {1}, {**3**, 3}, {5}, {}, {10}, {19}
{}, {1}, {}, {3, 3}, {}, {5}, {}, {10}, {19}
1, 3, 3, 5, 10, 19

Dans l'algorithme de *tri fusion* (inventé par Neumann en 1945), pour trier une liste de taille n, on la décompose en deux sous-listes, une de taille $\lfloor \frac{n}{2} \rfloor$ et l'autre de taille $\lceil \frac{n}{2} \rceil$. On trie récursivement ces deux listes, puis on les fusionne. On veille à ce que la fusion s'effectue avec une complexité linéaire.

10, 1, 5, 19, 3, 3
{10, 1, 5}, {19, 3, 3}
{{10, 1}, {5}}, {{19, 3}, {3}}
{{{10}, {1}},{5}}, {{{19},{3}},{3}}
{{1, 10}, {5}}, {{3, 19},{3}}
{1, 5, 10}, {3, 3, 19}
1, 3, 3, 5, 10, 19

Des simulations de ces différents tris peuvent être visualisées à l'adresse suivante :
https://www.toptal.com/developers/sorting-algorithms

Partie II : Implémentations

1. Écrire une fonction `recherche_triee(l, e)` qui prend en argument une liste **triée** `l` et un élément `e` et qui renvoie `True` si l'élément est présent dans `l` et `False` sinon. Votre algorithme devra être de complexité logarithmique en la taille de la liste.

2. Écrire une fonction `echange` qui prend en argument une liste et deux entiers `i` et `j`, la modifie en échangeant ses éléments d'indices `i` et `j`.
Les listes sont des objets **mutables**. *Ainsi, l'instruction* `return` *n'est pas nécessaire dans cette fonction. La fonction modifie la liste et son résultat est de type* `None`.

3. Tri par sélection.

a) Écrire une fonction `mini` qui prend en argument une liste `liste` et un entier naturel `i` et retourne le couple constitué du minimum des éléments de l'ensemble {`liste[i]`, ..., `liste[len(liste)-1]`} ainsi que l'indice de sa première apparition.

b) En utilisant les fonctions `mini` et `echange`, écrire une fonction `tri_selection` qui trie la liste passée en argument (en la modifiant) selon la méthode de tri par sélection.

c) Étudier la complexité de `tri_selection` dans le meilleur et dans le pire des cas.
On n'oubliera pas le coût des appels à la fonction `mini`.

4. Tri par insertion.

a) Écrire une fonction `tri_partiel` qui prend en argument une liste `liste` dont on suppose que `liste[0]` $< \cdots <$ `liste[i-1]` et l'entier `i` et qui insère l'élément `liste[i]` à sa place parmi les `i` premiers éléments de `liste` (en modifiant cette dernière).
Cette fonction utilisera la fonction `echange` *écrite dans le préambule uniquement avec deux indices consécutifs en faisant remonter les éléments à leur place. Le* **slicing**, *cher à Python mais qui masque des complexités, ne sera pas utilisé.*

b) Écrire une fonction `tri_insertion` qui trie la liste passée en argument (en la modifiant) selon la méthode du tri par insertion.

c) Étudier la complexité de `tri_insertion` dans le meilleur et dans le pire des cas.

5. Tri rapide.

a) Écrire une fonction `partition` qui, étant donnée une liste `liste` et la valeur d'un pivot `pivot`, renvoie le triplet de trois listes `(l_inf, l_egal, l_sup)` décrit précédemment.

b) Écrire une fonction `tri_rapide` qui trie la liste passée en argument (en la modifiant) selon l'algorithme du tri rapide. Le pivot sera choisi comme étant le premier élément de la liste.
On utilisera récursivement la fonction de tri sur les éléments inférieurs et supérieurs au pivot.

c) Discuter la complexité de la fonction `tri_rapide` dans le meilleur et dans le pire des cas.

6. Le programme précédent crée 3 nouvelles listes à chaque appel récursif, il n'est pas de complexité optimale. Cherchez à l'optimiser en effectuant le partitionnement en place. Vous pouvez également randomiser l'indice du pivot de manière à éviter les cas de complexité maximale.

7. Tri fusion.

a) Écrire une fonction `fusion` qui, étant donnée une liste `l` et deux indices `g` et `d` (on note `m = (g+d)//2`), copie les listes supposées triées `[l[g],...,l[m-1]]` et `[l[m],...,l[d-1]]` puis les fusionne en les triant et stocke le résultat en écrasant les données de `[l[g],...,l[d-1]]`.

b) Écrire une fonction `tri_fusion` qui trie la liste passée en argument (en la modifiant) selon l'algorithme du tri fusion.
Le programme précédent utilise une complexité spatiale de l'ordre de la taille de la liste initiale. KRONROD *a proposé en 1969 une adaptation qui utilise une place additionnelle constante.*

c) On suppose que la liste initiale est de taille 2^p. Écrire la relation de récurrence satisfaite par le nombre de comparaisons T_p nécessaire pour trier cette liste. En déduire une expression de T_p en fonction de 2^p.

Partie III : Pour aller plus loin...

8. Le *tri à bulles* consiste à parcourir la liste et à intervertir toutes les paires d'éléments consécutifs qui ne sont pas ordonnées. Une fois la liste parcourue, le tri repart du début de la liste. Le tri s'arrête lorsque, lors du parcours de la liste, aucun échange n'est effectué.

a) Écrire une fonction `tri_bulle` qui trie la liste `liste` selon la méthode du tri à bulles.

b) Étudier la complexité de `tri_bulle` dans le pire des cas.

9. Le *tri du postier* ne repose pas uniquement sur des comparaisons entre éléments de la liste mais suppose également que les éléments de la liste ne peuvent prendre qu'un nombre fini de valeurs, supposées être des entiers compris entre 0 et $m-1$ pour simplifier. Ce tri est appelé *tri du postier* car il peut être utilisé pour trier des enveloppes en fonction de leur adressage. On suppose que les éléments des listes passées en arguments sont des couples dont le premier élément est un entier compris entre 0 et $m-1$ (l'adresse) et le second est une chaîne de caractères (la lettre). Par exemple, vous pourrez trier par adresses croissantes :

```
[(10, "hello"), (1, "salut"), (0, "bonjour"), (9, "Buenos dias")]
```

a) Écrire une fonction `denombrement`, qui prend argument une liste d'adresses `liste` et l'entier `m`, puis renvoie une liste `occ` de taille `m` telle que pour tout entier i, `occ[`i`]` soit égal au nombre d'occurrences de i dans `liste`.
On veillera à ne parcourir la liste initiale qu'une seule fois.

b) Écrire une fonction `place` qui prend en argument une liste d'adresses `liste` et l'entier `m`, puis renvoie une liste `position` de taille `m` telle que pour tout entier i, `position[`i`]` sera la position de la première lettre adressée à i dans la liste triée issue de `liste`.
Pour les éléments absents de la la liste, l'élément correspondant dans le vecteur renvoyé pourra être choisi à votre guise.

c) Écrire une fonction `tri_postier` qui prend en arguments une liste d'adresses ainsi que l'entier `m` et renvoie cette liste triée en temps linéaire par rapport à sa longueur.
Dans un second temps, on pourra réécrire cette fonction en modifiant la liste passée en argument.

Mathématiciens

Neumann John von (28 déc. 1903 à Budapest-8 fév. 1857 à Washington).
Kronrod Alexander Semenovich (22 oct. 1921 à Moscou-6 oct. 1990 à Moscou).
Hoare Charles Antony Richard (11 jan. 1934 à Colombo-).