Informatique

TP 5

Tris récursifs

1 Le tri fusion

Une **fusion** de deux listes triées `x` et `y` est une liste triée `z` qui contient les mêmes éléments que la liste `x + y`. L'algorithme **tri fusion** ("**merge sort**" en Anglais) repose sur fait que deux listes triées peuvent être fusionnées en temps linéaire.

Question 1. *Écrire une fonction* Python *qui prend en arguments deux listes triées* `x` *et* `y`*, et qui renvoie une liste triée contenant les mêmes éléments que* `x + y`. **Le nombre d'opérations de cette fonction doit être en** $O(\texttt{len(x)} + \texttt{len(y)})$**.** Q.1

Le tri fusion procède ensuite par une stratégie « diviser pour régner ». L'idée est la suivante. À chaque itération,

(1) on coupe la liste en deux,

(2) on trie récursivement les deux sous-listes,

(3) et on fusionne les résultats.

On vise une complexité en $O(n \log_2(n))$, où n est la longueur de la liste à trier.

Attention. Une liste de longueur < 2 ne peut être découpée en deux listes non-vides. Cependant, une liste de longueur < 2 est nécessairement triée.

Question 2. *Écrire une fonction* Python `mergesort` *implémentant l'algorithme de tri fusion.* Q.2

Complexité* Nous allons maintenant raisonner de manière abstraite sur la complexité du tri fusion. Pour ce faire, notons $T(n)$ le nombre d'opérations effectuées (dans le pire cas) par `mergesort` sur une liste de longueur n. On suppose que T est une fonction croissante telle que

$$T(n) \quad = \quad T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + g(n)$$

où $g(n)$ est en $O(n)$. Notre objectif est de montrer que $T(n)$ est en $O(n \log_2 n)$.

La difficulté vient des arrondis entiers $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ apparaissant dans l'expression de $T(n)$ ci-dessus. Rappelons que pour un nombre réel $x \geq 0$,

- $\lfloor x \rfloor$ est le plus grand entier $\leq x$ (c.-à-d. la partie entière de x),
- $\lceil x \rceil$ est le plus petit entier $\geq x$.

Nous allons dans un premier temps considérer une expression ne faisant intervenir que l'arrondi $\lfloor n/2 \rfloor$.

Question* 3. *Considérons une fonction S telle que* *Q.3

$$S(n) \quad \leq \quad 2S(\lfloor n/2 \rfloor) + h(n)$$

où $h(n)$ est en $O(n)$. Montrer que $S(n)$ est en $O(n \log_2 n)$.

Nous allons maintenant ramener la fonction T ci-dessus à une fonction S comme à la Question 3. Pour cela, nous effectuons un changement de variable.

Question* 4. *Donner un α tel que $S(n) := T(n + \alpha)$ satisfait l'hypothèse de la Question 3.* *Q.4

Question* 5. *Montrer que $T(n)$ est en $O(n \log_2(n))$.* *Q.5

2 Le « tri rapide »

Le « tri rapide » (**quicksort** en Anglais) est un des algorithmes de tri les plus utilisés en pratique. C'est un algorithme « diviser pour régner », dont le principe est différent de celui du tri fusion.

Considérons une liste `x`, ainsi que deux entiers `d`, `f` avec $0 \leq \texttt{d} < \texttt{f} \leq \texttt{len(x)}$. Pour le moment, appelons **pivot** de $(\texttt{x}, \texttt{d}, \texttt{f})$ l'élément `p` de `x` à la position $\texttt{f} - 1$ (c.-à-d. $\texttt{p} = \texttt{x}[\texttt{f} - 1]$).

Question 6. *Écrire une fonction* Python `partition`$(\texttt{x}, \texttt{d}, \texttt{f})$*, où* $(\texttt{x}, \texttt{d}, \texttt{f})$ *sont comme ci-dessus, et qui* **modifie** `x` *de manière à ce que pour un certain* $\texttt{i} \in \{\texttt{d}, \ldots, \texttt{f} - 1\}$*,* Q.6

- $\texttt{x}[\texttt{d} : \texttt{i}]$ *ne contienne que des éléments* $\leq \texttt{p}$*,*
- $\texttt{x}[\texttt{i}] = \texttt{p}$*,*
- $\texttt{x}[\texttt{i} + 1, \texttt{f}]$ *ne contienne que des éléments* $> \texttt{p}$*.*

Un appel à `partition`$(\texttt{x}, \texttt{d}, \texttt{f})$ *doit renvoyer* `i`. **Le nombre d'opérations doit être en** $O(\texttt{f} - \texttt{d})$**.**

Question 7.** *Démontrez que votre fonction* `partition` *de la Question 6 est correcte.* **Q.7

Le tri rapide procède ensuite récursivement. À chaque itération, en partant de $(\texttt{x}, \texttt{d}, \texttt{f})$ comme ci-dessus,

- si $\texttt{d} = \texttt{f}$, alors on s'arrête ($\texttt{x}[\texttt{d} : \texttt{f}]$ est vide).
- sinon,
  (1) on choisit un pivot dans $\texttt{x}[\texttt{d} : \texttt{f}]$ et on le place en position $\texttt{f} - 1$,
  (2) on partitionne $(\texttt{x}, \texttt{d}, \texttt{f})$, et on utilise la position `i` ainsi obtenue pour recommencer récursivement avec $(\texttt{x}, \texttt{d}, \texttt{i})$ et $(\texttt{x}, \texttt{i} + 1, \texttt{f})$.

Il existe de nombreuses possibilités pour le choix du pivot, qui peuvent grandement impacter l'efficacité du tri rapide. On se contentera de la solution la plus simple : toujours choisir $\texttt{x}[\texttt{f} - 1]$ comme pivot pour $(\texttt{x}, \texttt{d}, \texttt{f})$.

Question 8. *Écrire une fonction* Python `quicksort` *implémentant le tri rapide.* Q.8

Complexité. Dans le pire cas, la complexité du tri rapide est en $O(n^2)$ (où n est la longueur de la liste à trier). Par ailleurs, si (la position de) chaque pivot est choisi(e) aléatoirement, alors la complexité « en moyenne » du tri rapide est en $O(n \log_2 n)$.

Bien que ces complexités théoriques laissent penser que le tri rapide est moins efficace que le tri fusion, en pratique le tri rapide est dans bien des cas le plus ... rapide.

3 Stabilité*

Lorsque l'on trie une liste de valeurs numériques, le traitement des éléments de même valeur est peu important. Cependant, en pratique il est courant d'avoir à trier des listes de valeurs structurées.

Exemple 3.1. *Considérons par exemple une liste* `x` *dont les éléments sont tous des paires de la forme* `(nom,age)`*, où* `nom` *est une chaine de caractères et* `age` *est un entier. Supposons que* `x` *soit triée par ordre alphabétique sur* `nom`*. On souhaite trier* `x` *par ordre croissant selon* `age`*, avec la contrainte suivante :*

— *les éléments de même* `age` *doivent rester dans le même ordre selon leur* `nom`*.*

Ainsi, si `x` *est la liste*

$$[(\texttt{"Dupont"}, 25)\ ,\ (\texttt{"Durand"}, 20)\ ,\ (\texttt{"Durand"}, 25)\ ,\ (\texttt{"Martin"}, 20)\ ,\ (\texttt{"Martin"}, 15)]$$

alors on veut obtenir la liste

$$[(\texttt{"Martin"}, 15)\ ,\ (\texttt{"Durand"}, 20)\ ,\ (\texttt{"Martin"}, 20)\ ,\ (\texttt{"Dupont"}, 25)\ ,\ (\texttt{"Durand"}, 25)]$$

Un algorithme de tri est dit **stable** lorsque l'ordre relatif des éléments de même clef n'est pas modifié. Dans l'Exemple 3.1, dire qu'on « souhaite trier la liste `x` par ordre croissant selon `age` » revient à dire qu'on souhaite trier la liste `x` selon la clef `age`.

La stabilité est une propriété importante pour certaines applications. Certains algorithmes de tri sont stables, d'autres non.

Remarque. Dans ce qui suit, on s'intéresse au tri fusion et au tri rapide, mais aussi aux tris du TP2 (tri par propagation, tri par sélection, et tri par insertion). [1]

3.1 Trier des données structurées

Les tris de ce TP et du TP2 supposent que l'on dispose des opérations <, <=, > et >= sur les données à trier. Lorsqu'on cherche à trier des données structurées (comme dans l'Exemple 3.1), ces opérations ne sont en général pas directement disponibles.

Nous allons voir deux méthodes pour pallier à cette difficulté : une première méthode, générale, et une seconde méthode (adoptée pour le tri primitif de Python), qui fonctione dans beaucoup de situations concrètes.

3.1.1 Première méthode : ordre abstrait

La méthode la plus générale pour trier des données structurées est de paramétrer les fonctions de tri par une fonction appelée `order` (par exemple), et telle que `order(a,b)` renvoie `True` si, et seulement si, `a` est inférieur ou égal à `b` selon l'ordre envisagé.

Exemple 3.2. *Dans le cadre de l'Exemple 3.1, on peut prendre*

```
def order(a,b) :
  return a[1] <= b[1]
```

Avec cette méthode, le tri par propagation du TP2 pourrait être réecrit en :

```
def tri_bulles_abs(x,order):
    for i in range(len(x),0,-1) :
        for j in range(1,i) :
            if not order(x[j-1],x[j]) :
                x[j],x[j-1] = x[j-1],x[j]
    return None
```

Question 9. *Adapter les fonctions* `mergesort` *(Question 2) et* `quicksort` *(Question 8) de manière à ce qu'elles prennent en argument une fonction* `order` *comme ci-dessus.* Q.9

3.1.2 Seconde méthode : projections

Alors que la méthode présentée au §3.1.1 est très générale, en pratique les ordres envisagés sur des données structurées sont souvent obtenus en projetant ces données sur une de leur composante, de manière à pouvoir ensuite utiliser un opérateur de comparaison Python (parmi <, <=, > et >=). Cela peut permettre de gagner en efficacité.

1. http://perso.ens-lyon.fr/colin.riba/teaching/cpes/tp/tp02.pdf

Exemple 3.3. *Dans le cadre de l'Exemple 3.1, si on prend*

```
def proj(a) :
  return a[1]
```

alors `proj(a) <= proj(b)` *renvoie le même résultat que* `a[1] <= b[1]`.

Voici une adaptation du tri par propagation selon cette méthode.

```
def tri_bulles_proj(x,proj):
    for i in range(len(x),0,-1) :
        for j in range(1,i) :
            if proj(x[j]) < proj(x[j-1]) :
               x[j],x[j-1] = x[j-1],x[j]
    return None
```

Question 10. *Adapter les fonctions* `mergesort` *(Question 2) et* `quicksort` *(Question 8) de manière à ce qu'elles prennent en argument une fonction de projection comme ci-dessus.* Q.10

3.2 Tris stables

Nous avons prétendu plus haut que certains tris ne sont pas stables et que d'autres le sont. Vous allez maintenant le vérifier.

Question* 11. *Parmi les tris de ce TP (tri fusion, Question 2 et tri rapide, Question 8), ainsi que parmi ceux du TP2 (tri par propagation, par sélection, par insertion), lesquels sont stables ?* *Q.11

3.3 Les tris primitifs de Python et de NumPy

3.3.1 Python

Le tri primitif sur les listes Python traite les données structurées avec des fonctions de projection (appelées **key functions**), dans l'esprit de la méthode du §3.1.2.

On dispose de la méthode `list.sort()` et de la fonction `sorted`.[2] Par exemple,

```
>>> x = list(range(10))
>>> x.reverse()
>>> x
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ou encore

```
>>> x = list(range(10))
>>> x.reverse()
>>> list.sort(x)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Remarque 3.4 (Itérateurs Python)**.** *On veillera à ne pas confondre les* **listes** Python *avec les* **itérateurs** Python*. La primitive* Python `sorted` *renvoie une liste triée à partir d'un itérateur. Par exemple :*

2. https://docs.python.org/3/howto/sorting.html

```
>>> y = range(10)
>>> y
range(0, 10)
>>> list(y)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> reversed(y)
<range_iterator object at 0x1060e6240>
>>> list(reversed(y))
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sorted(reversed(y))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

L'algorithme de tri implémenté est appelé « timsort ».[3] C'est un tri **stable**.

Pour trier des données structurées, il faut fournir une projection (appelée **key function**). Commençons par voir ce qu'il se passe si on ne précise rien :

```
>>> x = [('Dupont', 25), ('Durand', 20), ('Durand', 25), ('Martin', 20), ('Martin', 15)]
>>> sorted(x)
[('Dupont', 25), ('Durand', 20), ('Durand', 25), ('Martin', 15), ('Martin', 20)]
```

On peut utiliser la fonction projection `proj` grâce à l'argument optionel `key`, en indiquant `key=proj`. Voici une première manière de le faire :

```
>>> proj = lambda a : a[1]
>>> x = [('Dupont', 25), ('Durand', 20), ('Durand', 25), ('Martin', 20), ('Martin', 15)]
>>> sorted(x,key=proj)
[('Martin', 15), ('Durand', 20), ('Martin', 20), ('Dupont', 25), ('Durand', 25)]
```

Voici une deuxième manière :

```
>>> proj = lambda a : a[1]
>>> x = [('Dupont', 25), ('Durand', 20), ('Durand', 25), ('Martin', 20), ('Martin', 15)]
>>> x.sort(key=proj)
>>> x
[('Martin', 15), ('Durand', 20), ('Martin', 20), ('Dupont', 25), ('Durand', 25)]
```

Voici enfin une troisième manière :

```
>>> proj = lambda a : a[1]
>>> x = [('Dupont', 25), ('Durand', 20), ('Durand', 25), ('Martin', 20), ('Martin', 15)]
>>> list.sort(x,key=proj)
>>> x
[('Martin', 15), ('Durand', 20), ('Martin', 20), ('Dupont', 25), ('Durand', 25)]
```

3.3.2 NumPy

La situation en NumPy est légèrement plus compliquée, du fait d'une discipline plus stricte sur les tableaux NumPy que sur les liste Python.[4]

Nous allons nous contenter de traiter le cas de tableaux contenant des paires d'entiers.

3. https://en.wikipedia.org/wiki/Timsort
4. Cette discipline est justifiée par des gains substentiels en temps et en espace de calcul.

```
>>> import numpy as np
>>> dtype = [('fst',int), ('snd',int)]
```

Par exemple

```
>>> x = [(0,3),(0,2),(0,1),(1,3),(1,2),(1,1),(2,3),(2,2),(2,1)]
>>> a = np.array(x, dtype=dtype)
>>> a
array([(0, 3), (0, 2), (0, 1), (1, 3), (1, 2), (1, 1), (2, 3), (2, 2),
       (2, 1)], dtype=[('fst', '<i8'), ('snd', '<i8')])
```

Ainsi, `a` est un tableau croissant selon sa composante `fst`. La documentation de NumPy indique comment trier un tableau selon différentes composantes. [5]

Question* 12. *Trier un tableau* `a` *comme ci-dessus selon sa composante* `fst`*, selon sa composante* `snd`*. Peut-on le faire de manière stable ?*

*Q.12

5. https://numpy.org/doc/stable/reference/generated/numpy.sort.html