Algorithmes de tri

Disposer d'une liste déjà triée est par exemple intéressant lorsque l'on a besoin de rechercher si un élément est présent dans une liste de n éléments puisque cette recherche est alors plus rapide - en $\mathcal{O}(\log(n))$, alors qu'elle est en $\mathcal{O}(n)$ si la liste n'est pas triée - cf. TD n°2 et cours de première année.

Dans tout ce TD, on note `L` ou `M` une liste de nombres (pas forcément distincts) que l'on veut trier dans l'ordre croissant (au sens large).

On présente ici les trois algorithmes de tri qui figurent au programme. Puis on évalue leur efficacité respective. Enfin on s'intéresse à la possibilité de programmer ces algorithmes *en place*.

I Présentation de trois algorithmes de tri

I.1 Tri par insertion

C'est le tri que l'on utilise souvent pour trier un jeu de carte : une main prend les cartes une par une et les insère à leur place parmi les cartes déjà triées tenues dans l'autre main.

Q1 Écrire une fonction `insertion(M, x)` qui prend en argument une liste `M` de nombres triés dans l'ordre croissant et un nombre `x` et qui renvoie la liste triée dans l'ordre croissant, formée de `x` et des éléments de `M`.
Par exemple, `insertion([1,3,6,8,9], 4)`→`[1,3,4,6,8,9]`

Q2 En déduire une fonction `tri_insertion(L)` qui renvoie la liste formée des éléments de `L` triés dans l'ordre croissant. On créera une nouvelle liste.

Q3 Écrire une version récursive de la fonction précédente (qui utilise la fonction `insertion`).

I.2 Tri rapide « quicksort »

Le tri rapide est un algorithme **récursif** :

- Si la liste est vide ou de longueur 1, elle est triée.
- Sinon, on choisit un élément de la liste `L`, appelé pivot (nous choisissons dans la suite le premier élément de la liste comme pivot).
- On partitionne la liste `L` en trois listes `L1`, `[L[0]]` et `L2` de la manière suivante :
  - `L1` est la liste des éléments de `L[1:]` strictement inférieurs au pivot (dans le même ordre que dans `L`) ;
  - `L2` est la liste des éléments de `L[1:]` supérieurs ou égal au pivot (dans le même ordre que dans `L`).
- On appelle récursivement le tri sur les listes `L1` et `L2`
  et on renvoie la liste `tri_rapide(L1) + [L[0]] + tri_rapide(L2)`.

Q4 On se propose de trier la liste `[5,1,8,3,0,2,7,4,6]`.
Décrire sous forme d'arbre les différents appels récursifs effectués, puis former l'arbre inversé des résultats des différents appels récursifs effectuées jusqu'au résultat final.

Départ : `[5,1,8,3,0,2,7,4,6]`

niveau 1 : `[1,3,0,2,4]` `[5]` `[8,7,6]`

niveau 2 :

niveau 3 :

Résultats niveau 2 :

Résultats niveau 1 :

Résultat final :

Q5 Écrire une fonction `partition(L)` qui renvoie `L1, L2`.

Q6 Écrire une fonction `tri_rapide(L)` qui renvoie la liste `L` triée par tri rapide.

I.3 Tri par fusion

Le tri par fusion est un algorithme récursif fondé sur le principe de dichotomie (stratégie « diviser pour régner ») et sur la « fusion » de deux listes déjà triées :

- Si la liste est vide ou de longueur 1, elle est triée.
- Sinon, on appelle récursivement le tri sur les listes `L1=L[:n//2]` et `L2=L[n//2:]`.
- Les résultats de ces appels récursifs sont utilisés pour former une liste triée plus grande, par fusion.

Q7 Écrire une fonction `fusion(L1,L2)` d'arguments deux listes `L1` et `L2`, chacune triée dans l'ordre croissant, et qui renvoie la liste `L` formée des éléments de `L1` et `L2` triée dans l'ordre croissant. *Par exemple :* `fusion([1,4,5],[2,3,8,9])` $\rightarrow$ `[1,2,3,4,5,8,9]`

Q8 On se propose de trier la liste `[5,1,8,3,0,2,7,4,6]`. Écrire l'arbre des appels récursifs en indiquant `L1` et `L2`, puis former l'arbre inversé des résultats des différents appels récursifs effectuées, ceci jusqu'au résultat final.

Départ : `[5,1,8,3,0,2,7,4]`

niveau 1 :

niveau 2 :

niveau 3 :

Résultats niveau 2 :

Résultats niveau 1 :

Résultat final :

Q9 Écrire une fonction `tri_fusion(L)` qui renvoie la liste `L` triée par tri par fusion.

II Complexité en temps

La complexité des tris est mesurée en fonction de la longueur n de la liste `L` à trier.

Q10 Combien de comparaisons effectue-t-on dans le pire (resp. le meilleur) des cas de la fonction `insertion(L,x)`.
En déduire la complexité dans le pire des cas de la fonction `tri_insertion(L)`.

Q11 Combien de comparaisons effectue-t-on dans le pire des cas de la fonction `fusion(L1,L2)`, en fonction de n (qui est la somme des longueurs des deux listes `L1` et `L2`).

Q12 Montrer que, pour une liste `L` de longueur n, le nombre p d'appels récursifs effectués lors d'un tri par fusion vérifie $\log_2 n \leq p < \log_2 n + 1$ où $\log_2 n = \frac{ln(n)}{ln(2)}$.
On note log *et non* $\log_2$ *en informatique.*
En déduire la complexité dans le pire des cas de la fonction `tri_fusion(L)`.
On peut montrer qu'« en moyenne » le tri rapide a une complexité de l'ordre de $\mathcal{O}(n\log(n))$.

III Amélioration de la complexité en espace : tris en place

Lorsque l'on veut trier une liste très longue, outre la question de la rapidité du tri, se pose la question de l'utilisation de l'espace mémoire. La mémoire est fortement sollicitée lorsque les données à trier sont recopiées, globalement ou en partie, de nombreuses fois. Au contraire, l'espace mémoire est économisé lorsque l'on effectue le tri « en place », c'est-à-dire en effectuant uniquement des modifications successives de la liste à trier au départ, sans faire de copies.
Il faut donc bien distinguer les types d'opérations effectuées :

Opérations en place (variable `L` modifiée)	*Opérations pas en place (variable `L` recopiée)*
`L.append(...)`	`L = L + [...]`
`L.pop()`	`L = L[:-1]`
`L[i:j] = ...`	`M = L[i:j]`

III.1 Tri par insertion en place

Le tri par insertion peut se faire en place. Pour cela on opère par exemple par permutations d'éléments consécutifs de la liste pour faire en sorte que les premiers éléments de la liste soient triés, et l'on fait augmenter le nombre d'éléments concernés.

Q13 On part de la liste `L` qui vaut `[5,2,8,3,0,1,7]`. Compléter la liste des valeurs successives de `L` de sorte qu'à l'étape `i` les éléments `0` à `i` soient les mêmes que ceux du départ, mais sont triés dans l'ordre croissant, les autres éléments étant inchangés (valeur et position) :

départ : `[5,2,8,3,0,1,7]`

`i=1` `[2,5,8,3,0,1,7]`

`i=2` `[2,5,8,3,0,1,7]`

`i=3`

`i=4`

`i=5`

`i=6`

Q14 Pour i fixé, pour insérer `L[`i`]` à sa place dans `[L[0],`$\cdots$`,L[`$i-1$`]]` sans écraser d'élément et sans avoir à stocker les valeurs de tous ces éléments, on stocke `L[`i`]` dans `L[`j`]` (avec $j = i$), et on procède par permutations successives de deux éléments contigus `L[`$j-1$`]`, `L[`j`]` pour j allant de i à 0, tant que `L[`j`] < L[`$j-1$`]`. *Par exemple,* si $i = 4$:

`[2,4,6,9,3,11,1]`→`[2,4,6,3,9,11,1]`→`[2,4,3,6,9,11,1]`→`[2,3,4,6,9,11,1]`

Écrire les valeurs successives de la liste `L` après le j-ème traitement (avec $j \leq 5$) lorsque $i = 5$ et que l'on part de la liste `[0,2,3,5,8,1,7]` pour obtenir la liste `[0,1,2,3,5,8,7]`.

départ à $i = 5$ **:** `[0,2,3,5,8,1,7]`

arrivée à $i = 6$ **:** `[0,1,2,3,5,8,7]`

Q15 Écrire une procédure `insertion_en_place(L, i)` qui prend en argument la liste `L` dont on suppose les éléments `L[0], ..., L[`$i-1$`]` triés dans l'ordre croissant, et qui modifie cette liste pour que les éléments `L[0], ..., L[`i`]` soient triés, ceci sans modifier la valeur ni la place des éléments `L[`$i+1$`], ..., L[`$n-1$`]`.

Q16 En déduire une procédure `tri_insertion_en_place(L)` qui modifie la liste `L` pour que tous ses éléments soient triés en place, par insertion.

III.2 Tri rapide en place

Une manière de traiter seulement une partie de la liste `L` « en place » lors d'un appel récursif est de spécifier des variables supplémentaires qui indiquent la portion de liste à traiter :

```
def fonction_recursive(L,debut,fin):
    # traitement (récursif) de L[debut:fin]
```

Q17 Écrire une procédure `tri_rapide_en_place(L)` qui renvoie la liste `L` triée par tri rapide effectué « en place ». Cette procédure effectuera uniquement 1 appel à une fonction auxiliaire récursive avec des paramètres bien choisis.

IV Quelques application des tris

Q18 Écrire une fonction `mediane(L)` qui renvoie la médiane des nombres de la liste `L`.

Q19 Écrire fonction `SupprimeDoublons(L)` qui renvoie la liste des éléments de `L` où l'on a supprimé ceux qui figurent en plusieurs exemplaires, dans un ordre non imposé.

Q20 Écrire `SumGroupBy(T)` qui prend en argument un tableau `T` formé de deux listes de nombres `T[0]` et `T[1]` et qui renvoie le tableau `R` formé de deux listes de nombres `R[0]` et `R[1]` dont le premier contient la liste des éléments de `T[0]` sans répétition et tel que, pour tout `i`, `R[1][i]` contient la somme de toutes les valeurs `T[1][j]` telles que `T[0][j]` est égal à `R[0][i]`.
Par exemple : `SumGroupBy([[1,2,1,2,3],[10,7,6,2,8]])`→`[[1,2,3],[16,9,8]]`

Corrigé

Q1

```
def insertion(M, x):
    '''L est supposé trié dans l'ordre croissant; on insère x dans L, éventuellement vide'''
    for i in range(len(M)):
        if x <= M[i]: # afin d'assurer la stabilité du tri
            return M[:i] + [x] + M[i:]
    return M + [x] # x doit être ajouté en fin de liste

# ou
def insertion(M, x):
    '''L est supposé trié dans l'ordre croissant; on insère x dans L, éventuellement vide'''
    i = 0
    while i < len(M) and x > M[i]:
        i = i+1
    return M[:i] + [x] + M[i:]
```

Q2

```
def tri_insertion(L):
    '''Tri insertion par ordre croissant avec nouvelle liste'''
    M = []
    for x in L:
        M = insertion(M, x) # M est toujours triée
    return M

#ou avec un parcours avec les indices

def tri_insertionB(L):
    '''Tri insertion par ordre croissant avec nouvelle liste'''
    M = []
    for i in range(len(L)):
        M = insertion(M, L[i]) # M est toujours triée
    return M
```

Q3

```
# version récursive
def tri_insertionRec(L):
    '''Tri insertion par ordre croissant'''
    if L == []:
        return []
    else:
        # par définition, tri_insertionRec(L[1:]) est toujours triée
        return insertion(tri_insertionRec(L[1:]), L[0])
```

Q4 En bleu, les paramètres de la fonction récursive.
En noir, la variable non paramètre de la fonction récursive.
En violet, les paramètres de la fonction récursive qui sont aussi résultat de l'appel.
En rouge, le résultat de l'appel.

Départ : [5,1,8,3,0,2,7,4,6]
niveau 1 : [1,3,0,2,4] [5] [8,7,6]
niveau 2 : [0] [1] [3,2,4] [7,6] [8] []
niveau 3 : [2] [3] [4] [6] [7] []
Résultat niveau 2 : [2,3,4] [6,7]
Résultat niveau 1 : [0,1,2,3,4] [6,7,8]
Résultat final : [0,1,2,3,4,5,6,7,8]

Q5

```
60 def partition(L):
61     '''renvoie, pour L supposée non vide, les listes L1 < L[0] et L2 > L
       [0]'''
62     L1, L2 = [], []
63     for i in range(1, len(L)): # attention à bien partir de 1
64         if L[i] < L[0]: # afin d'assurer la stabilité du tri
65             L1.append(L[i])
66         else:
67             L2.append(L[i])
68     return L1, L2
```

Q6

```
71 def tri_rapide(L):
72     if len(L) <= 1: # on diminue le nombre d'appels en s'arrêtant aux
       listes de longueur 1
73         return L
74     else:
75         L1, L2 = partition(L)
76         return tri_rapide(L1) + [L[0]] + tri_rapide(L2)
77
78 L = [5, 1, 8, 3, 0, 2, 7, 4, 6]
79 assert tri_rapide(L) == [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Q7 Version sans sentinelles :

```
82 def fusion(L1, L2):
83     M = [] # liste contenant la liste fusionnée
84     i, j = 0, 0
85     n1, n2 = len(L1), len(L2)
86     # on initialise la liste par comparaison de min(n1,n2) éléments
87     while i < n1 and j < n2:
88         if L1[i] < L2[j]:
89             M.append(L1[i])
90             i += 1
91         else:
92             M.append(L2[j])
93             j += 1
94     # on complète la liste avec les éléments non traités, ce qui é
       quivaut à faire de façon performante M+L1[i:]+L2[j:]
95     if i == n1 :
96         for k in range(j, n2):
97             M.append(L2[k])
98     else:
99         for k in range(i, n1):
100            M.append(L1[k])
101    return M
```

Version avec sentinelles :

```
104 from numpy import inf
105
106 # variante itérative avec sentinelles
107 def fusion(L1, L2):
108     '''fusion avec sentinelles'''
109     L1.append(inf) # ajout d'une sentinelle
110     L2.append(inf) # ajout d'une sentinelle
111     M = [] # liste contenant la liste fusionnée
112     i, j = 0, 0
113     while i < len(L1) and j < len(L2):
114         if L1[i] < L2[j]:
115             M.append(L1[i])
116             i += 1
117         else:
118             M.append(L2[j])
119             j += 1
120     M.pop() # on enlève la sentinelle
121     return M
```

Version récursive :

```
124 # variante récursive, mais avec de trop nombreuses recopies de listes
125 def fusionRec(L1, L2):
126     if len(L1) == 0:
127         return L2
128     elif len(L2) == 0:
129         return L1
130     elif L1[0] < L2[0]:
131         return [L1[0]] + fusion(L1[1:], L2)
132     else:
133         return [L2[0]] + fusion(L1, L2[1:])
```

Q8 L1 en rouge et L2 en noir :

Départ :			[5,1,8,3,0,2,7,4]				
niveau 1 :		[5,1,8,3]				[0,2,7,4]	
niveau 2 :	[5,1]		[8,3]		[0,2]		[7,4]
niveau 3 :	[5] [1]		[8] [3]		[0] [2]		[7] [4]
Résultat niveau 2 :	[1,5]		[3,8]		[0,2]		[4,7]
Résultat niveau 1 :		[1,3,5,8]				[0,2,4,7]	
Résultat final			[0,1,2,3,4,5,7,8]				

Q9

```
138 def tri_fusion(L):
139     if len(L) <= 1: # on diminue le nombre d'appels en s'arrêtant aux
        listes de longueur 1
140         return L
141     else:
142         n = len(L)
143         return fusion(tri_fusion(L[:n//2]), tri_fusion(L[n//2:]))
144
145 L = [5, 1, 8, 3, 0, 2, 7, 4, 6]
146 assert tri_fusion(L) == [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Q10 Considérons la fonction la fonction `insertion(L,x)`. Dans le pire des cas, il faut comparer `x` avec chacun des éléments de la liste `L`, soit `n` comparaisons ; cela se produit lorsque `x` est plus grand que tous les nombres de `L`, càd. plus grand que `L[len(L)-1]`. Dans le meilleur des cas, lorsque `x<L[0]`, une seule comparaison suffit. On conclut que la complexité de la fonction `insertion(L,x)` est de $O(n)$ dans le pire des cas et de $O(1)$ dans le meilleur des cas.

Le pire cas (resp. le meilleur cas) du tri par insertion est celui où chaque insertion est dans le pire cas (resp. le meilleur cas), c'est à dire quand la liste est ordonnée dans l'ordre croissant (resp. dans l'ordre décroissant). Le nombre de comparaisons effectuées est alors de :

$$1+2+\cdots+n=\frac{n(n+1)}{2}=\mathcal{O}(n^2) \quad (\text{resp. } 1+1+1+\cdots+1=n=\mathcal{O}(n)) \quad .$$

On conclut que la complexité de la fonction `tri_insertion(L)` est de $\boxed{\mathcal{O}(n^2) \text{ dans le pire des cas}}$ et de $\boxed{\mathcal{O}(n) \text{ dans le meilleur des cas.}}$

Q11 Le pire des cas est celui ou les deux listes sont épuisées presque en même temps. Il y a alors $n-1$ comparaisons des éléments des deux listes. Par exemple,
`L1 = [1, 4, 5, 8, 9,...,` $n-1$`]` et
`L2 = [2, 3, 6, 7, 10,...,` n`]`

Q12 *Question difficile.*
Notons qu'il existe un entier naturel non nul p tel que $2^{p-1}<n\leq 2^p$.
À chaque itération, le nombre de listes est multiplié par 2. Après p itérations, il y a donc 2^p listes. Chacune d'elle a alors au plus 1 élément. *En effet, dans le cas contraire, où* 2^p-1 *listes ont* 1 *élément et* 1 *liste a* 2 *éléments, il y aurait* $(2^p-1)\times 1+1\times 2=2^p+1>2^p$ *éléments dans la liste de départ, absurde car* $n\leq 2^p$.
Le nombre total d'appels récursif ne peut donc pas excéder p.
Or, $2^{p-1}<n\leq 2^p \iff \log_2 n\leq p<\log_2 n+1$ où $\log_2 n=\frac{ln(n)}{ln(2)}$.

À chaque niveau d'appels, dans le pire des cas, la fusion parcourt n éléments, donc la complexité totale des fusions effectuées est égale a
$\mathcal{O}(np)=\mathcal{O}\left(\dfrac{n\ln(n)}{\ln 2}\right)=\mathcal{O}(n\log(n))$.

On conclut que la complexité de la fonction `tri_fusion(L)` est de $\boxed{\mathcal{O}(n\log(n)) \text{ dans le pire des cas}}$.

Q13 départ : `l=[5,2,8,3,0,1,7]`

`i=1 [2,5,8,3,0,1,7]`

`i=2 [2,5,8,3,0,1,7]`

`i=3 [2,3,5,8,0,1,7]`

`i=4 [0,2,3,5,8,1,7]`

`i=5 [0,1,2,3,5,8,7]`

`i=6 [0,1,2,3,5,7,8]`

Q14 départ : `[0,2,3,5,8,1,7]`

`j=5 [0,2,3,5,1,8,7]`

`j=4 [0,2,3,1,5,8,7]`

`j=3 [0,2,1,3,5,8,7]`

`j=2 [0,1,2,3,5,8,7]`

`j=1` , la boucle s'arrête (boucle `while`).

Q15

```
159 def insertion_en_place(L, i):
160     '''on suppose i>=1'''
161     j=i
162     while j > 0 and L[j] < L[j-1]: # boucle intérieure
163         L[j-1], L[j] = L[j], L[j-1]  # permutation de L[j-1] et L[j]
164         j = j-1
165     # pas de return car c'est une procédure
```

Q16

```
168 def tri_insertion_en_place(L):
169     for i in range(1, len(L)): # boucle extérieure
170         insertion_en_place(L, i)
```

Q17 Le principe adopté ici pour la partition en place à gauche et à droite d'un pivot est le suivant.
On choisit un pivot, par exemple `L[`*debut*`]`.
Soit $i \in [\![debut+1, fin-1]\!]$. De *debut* à $i-1$ inclus, on a la situation suivante :

- À gauche du pivot se trouvent les éléments plus petits que le pivot (notés "-" dans la liste ci-dessous).
- À droite du pivot jusqu'à $i-1$ inclus se trouvent les éléments plus grands que le pivot (notés "+" dans la liste ci-dessous).

Schématiquement : `L = [ - , pivot, + , L[`i`], ...]`.
Quand `L[`i`]` est plus petit que le pivot,

- on récupère le pivot, à l'indice *ind_pivot*, et les éléments de $ind\_pivot+1$ à $i-1$ inclus (les "+"),
- on place `L[`i`]` là où se trouve le pivot (en `L[`*ind_pivot*`]`),
- on met le pivot à la case suivante (en $ind\_pivot+1$)
- on reprend tous les "+" pour les placer de $ind\_pivot+2$ à i inclus,
- on passe au i suivant.

Quand `L[`i`]` est plus grand que le pivot, on le laisse à sa position à droite du pivot, et on passe au i suivant.

```
173 def partition_en_place(L, debut, fin):
174     ind_pivot = debut
175     for i in range(debut+1, fin):
176         if L[i] < L[ind_pivot]:
177             L[ind_pivot] , L[ind_pivot+1] ,L[ind_pivot+2 : i+1] = L[i] ,
    L[ind_pivot] , L[ind_pivot+1 : i]
178             ind_pivot += 1
179     return ind_pivot
180
181 def tri_aux(L, debut, fin):
182     if fin - debut > 1:
183         ind_pivot = partition_en_place(L, debut, fin)
184         tri_aux(L, debut, ind_pivot)
185         tri_aux(L, ind_pivot+1, fin)
186
187 def tri_rapide_en_place(L):
188     tri_aux(L, 0, len(L))
189
190 L = [5, 1, 8, 3, 0, 2, 7, 4, 6]; tri_rapide_en_place(L)
191 assert L == [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Q18

```
194 def mediane(L):
195     M = tri_fusion(L) # meileur dans le pire des cas
196     return M[len(L)//2]
```

Q19

```
199 def SupprimeDoublons(L):
200     M = tri_fusion(L)
201     N = []
202     for i in range(len(M)-1):
203         if M[i] != M[i+1]: # ajout des éléments du tableau trié du
    premier à l'avant-dernier quand ils ne sont pas doublonnés
204             N.append(M[i])
205     N.append(M[-1]) # le dernier est toujours différent du dernier élé
    ment ajouté
206     return N
```

Q20

```
209 def insertion_couple(L, x):
210     '''L est supposé trié dans l'ordre croissant; on insère x dans L, é
    ventuellement vide'''
211     for i in range(len(L)):
212         if x[0] <= L[i][0]: # ou x < L[i]
213             return L[:i] + [x] + L[i:]
214     return L + [x] # x doit être ajouté en fin de liste
215
216
217 def tri_insertion_couple(L):
218     '''Tri insertion par ordre croissant avec nouvelle liste'''
219     M = []
220     for x in L:
221         M = insertion_couple(M, x) # M est toujours triée
222     return M
223
224 def SommeGroupe(i, TT, indice):
225     '''accumulation selon la valeur de i; l'indice permet de se déplacer
     progressivement dans TT sans redémarrer au début'''
226     somme = 0
227     for j in range(indice, len(TT)):
228         if TT[j][0] == i:
229             somme += TT[j][1]
230             indice = j
231     return somme, indice + 1
232
233 def SumGroupBy(T):
234     T0, T1 = T
235     TT = [] # Fusion de T0 et T1 triée par T0[0]
236     for i in range(len(T0)):
237         TT.append([T0[i], T1[i]])
238     TT = tri_insertion_couple(TT)
239
240     R0 = SupprimeDoublons(T0) # tableau trié sans doublons
241     R1 = []
242     indice = 0
243     for k in range(len(R0)):
244         somme, indice = SommeGroupe(R0[k], TT, indice)
245         R1.append(somme)
246     return [R0, R1]
247
248 assert SumGroupBy([[1,2,1,2,3],[10,7,6,2,8]]) == [[1,2,3],[16,9,8]]
```