INFORMATIQUE

TD n°6

Algorithmes de tri

Disposer d'une liste délà 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 $O(\ln(n))$, alors qu'elle est en $O(n)$ si la liste n'est pas triée — voir cours de première année.

On présente ici les trois algorithmes de tri (dans l'ordre croissant) d'une liste `l` qui figurent au programme. Dans tout le TD `l` est une liste de nombres *disctincts*.

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.
De manière équivalente ce tri s'effectue avec une main qui tient toutes les cartes, la partie triée étant à gauche et la partie non encore triée à droite ; alors l'autre main prend les cartes de la partie droite une par une et les insère dans la partie gauche.

1.1 Boucle extérieure

On fait varier `i` de `1` à `n-1`. Pour chaque valeur de `i` où la liste `[l[0],...,l[i-1]]` est déjà triée, on modifie `l` en insérant `l[i]` à sa place, c'est-à-dire de telle sorte que la liste `[l[0],...,l[i]]` soit triée dans l'ordre croissant ; le reste de la liste n'est pas modifié.

Q1 Compléter les valeurs successives de la liste `l` après le `i`-ème traitement si `l=[5,2,8,3,0,1,7]`.

départ : `l=[5,2,8,3,0,1,7]`

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

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

`i=3`

`i=4`

`i=5`

`i=6`

1.2 Boucle intérieure : insertion d'un élément

Pour `i` fixé, pour insérer `l[i]` à sa place dans `[l[0],...,l[i-1]]`, 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]` (durant tout le processus `l[j]` contient la valeur intiale de `l[i]`). *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]`

Q2 Écrire les valeurs successives de la liste `l` après le `j`-ème traitement 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]`.

`j=5`

`j=4`

`j=3`

`j=2`

Que se passe-t-il pour `j=1` ?

1.3 Complexité.

Q3 Calculer la complexité (nombre de comparaisons effectuées) dans le pire des cas puis dans le meilleur des cas pour l'insertion de `l[i]` dans la liste `[l[0],...,l[i-1]]`?
Refaire les calculs en comptant cette fois-ci le nombre de permutations effectuées.

Q4 Calculer la complexité dans le pire des cas et dans le meilleur des cas du tri par insertion .

Q5 Écrire une procédure `tri_insertion(l)` qui modifie `l` en place pour effectuant un tri par insertion.

2 Tri rapide

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:]` strict. inférieurs au pivot (dans le même ordre que dans `l`) ;
  - `l2` est la liste des éléments de `l[1:]` strict. supérieurs 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)`.

Q6 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.

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

Deuxième niveau :

Troisième niveau :

Q7 Écrire une fonction `partition(l)` qui renvoie `[l1,l2]`.

Q8 Écrire une fonction `tri_rapide(l)` qui renvoie la liste `l` triée par tri rapide.

3 Tri par fusion

3.1 Fusion de deux listes triées

Q9 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` (répétés autant de fois qu'ils apparaissent dans `l1` et `l2`) triée dans l'ordre croissant.
Par exemple : `fusion([1,3,5],[2,3,8,9])` → `[1,2,3,3,5,8,9]`
Quelle est la complexité de cette fonction `fusion` ?

3.2 Tri par fusion proprement dit

Le tri par fusion est un algorithme récursif fondé sur le principe de dichotomie (stratégie « diviser pour régner ») et l'algorithme de fusion précédent.

- 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:]`.
- On fusionne les résultats de ces appels récursifs et on renvoie le résultat de cette fusion.

Q10 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`. Indiquer le résultat de l'appel récursif.

Départ : `[5,1,8,3,0,2,7,4]`
(Résultat)
Premier niveau :
(Résultat)
Deuxième niveau :
(Résultat)
Troisième niveau :

Q11 Écrire une fonction `tri_fusion(l)` qui renvoie la liste `l` triée par tri par fusion.

Q12 Quelle est la complexité du tri par fusion (dans le cas d'une liste de longueur $n = 2^p$) ?

4 Complément : complexité en espace et tri « 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, dans des variables différentes, et ce de nombreuses fois lors du tri. Au contraire, l'espace mémoire est économisé lorsque l'on effectue le tri « en place », c'est-à-dire en effectuant uniquement des modification successives de la liste à trier au départ, sans faire de copies (globales ou partielles). Il faut donc bien distinguer les types d'opérations effectuées :

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

Q13 Identifier pour chacun des trois algorithmes de tri précédents, s'ils ont été programmés « en place » ou non.

Q14 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]
```

Écrire une procédure `tri_rapide2(l)` qui renvoie la liste `l` triée par tri rapide effectué « en place ». Cette procédure effectuera uniquement un appel à une fonction auxiliaire recursive.

INFORMATIQUE

Corrigé du TD n°6

Q1

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]`

Q2

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]`

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

Q3 Dans le pire des cas il faut comparer `l[i]` avec chacun des éléments de la liste `[l[0],...,l[i-1]]`, soit `i comparaisons ;` cela se produit lorsque `l[i]` est plus petit que tous les nombres `l[0],...,l[i-1]` *i.e.* plus petit que `l[0]`. Dans le meilleur des cas, lorsque `l[i]>l[i-1]`, une seule comparaison suffit. Le nombre de permutations effectuées est alors respectivement de `i` et `0`.

Q4 Le pire cas (resp. le meilleur cas) est celui où chaque boucle intérieure est dans le pire cas (resp. le meilleur cas), c'est à dire quand la liste est ordonnée dans l'ordre décroissant (resp. dans l'ordre croissant). Le nombre de compraisons effectuées est alors de :

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

En nombre de permutations on obtient :

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}=O(n^2)\text{(resp. } 0+0+0+\cdots+0=0) \quad =O(n^2).$$

Q5

```
2  def tri_insertion(l):
3      for i in range(1,len(l)): #boucle exterieure
4          j=i
5          while j>=1 and l[j-1]>l[j]: #boucle interieure
6              l[j-1],l[j]=l[j],l[j-1]  #permutation de l[j-1] et l[j]
7              j=j-1
8      #pas de return car c'est une procedure
```

Q6

Départ : `[5,1,8,3,0,2,7,4,6]`
Premier niveau : `[1,3,0,2,4]` `[5]` `[8,7,6]`
Deuxième niveau : `[0]` `[1]` `[3,2,4]` `[7,6]` `[8]` `[]`
Troisième niveau : `[2]` `[3]` `[4]` `[6]` `[7]` `[]`

Q7

```
def partition(l):
    pivot=l[0]
    l1,l2=[],[]
    for x in l[1:]:
        if x < pivot:
            l1.append(x)
        else:
            l2.append(x)
    return [l1,l2]
```

Q8

```
def tri_rapide(l):
    if len(l)<=1:
        return l
    else:
        [l1,l2]=partition(l)
        return tri_rapide(l1)+[l[0]]+tri_rapide(l2)
```

Q9

```
def fusion(l1,l2):
    n1,n2=len(l1),len(l2)
    i1,i2=0,0
    l=[]
    while i1<n1 and i2<n2:
        if l1[i1]<l2[i2]:
            l.append(l1[i1])
            i1=i1+1
        else:
            l.append(l2[i2])
            i2=i2+1
    while i1<n1: #traitement des éléments restants de l1
        l.append(l1[i1])
        i1=i1+1
    while i2<n2: #traitement des éléments restants de l2
        l.append(l2[i2])
        i2=i2+1
    return l

#variante récursive, mais avec de nombreuses recopies de listes
def fusion(l1,l2):
    if len(l1)==0:
        return l2
    elif len(l2)==0:
        return l1
    elif l1[0]<l2[0]:
        return [l1[0]]+fusion(l1[1:],l2)
```

```
57      else:
58          return [l2[0]]+fusion(l1,l2[1:])
```

Complexité : il y a autant d'itérations que le total des nombres d'éléments des deux listes (en faisant la somme des nombres d'itération de `while` et de `for` dans la variante)).

Q10

Départ :								[5,1,8,3,0,2,7,4]							
(Résultat)								([0,1,2,3,4,7,8])							
Premier niveau :				[5,1,8,3]								[0,2,7,4]			
(Résultat)				([1,3,5,8])								([0,2,4,7])			
Deuxième niveau :	[5,1]				[8,3]				[0,2]				[4,7]		
(Résultat)	([1,5])				([3,8])				([0,2])				([4,7])		
Troisième niveau :	[5]	[1]			[8]	[3]			[0]	[2]			[4]	[7]	

Q11

```
61  def tri_fusion(l):
62      n=len(l)
63      if n<=1:
64          return l
65      else:
66          return fusion(tri_fusion(l[0:n//2]),tri_fusion(l[n//2:n]))
```

Q12 Lorsque $n = 2^p$, comme dans l'exemple pour $p = 3$, il y a p niveaux d'appels récursifs pour se ramener à des listes de longueur 1. À chaque niveau d'appel, la complexité totale des fusions effectuée est égale à n, d'où une complexité totale de $np = \dfrac{n \ln n}{\ln 2} = O(n \ln n)$.

Q13 Le tri par insertion a été effectué en place, mais pas les tris rapide et par fusion. Le tri par fusion se prête mal à une version en place, car les listes à fusionner ne sont a priori par formées à partir d'élements adjacents.

Q14 *Question difficile*

```
69  def partition_en_place(l,debut,fin):
70      pivot=debut
71      for i in range(debut,fin):
72          if l[i] < l[pivot]:
73              l[pivot],l[pivot+1],l [pivot+2:i+1]=l[i],l[pivot],l[pivot +1:i]
74              pivot=pivot+1
75      return pivot
76
77  def tri_aux(l,debut,fin):
78      if fin-debut>1:
79          pivot=partition_en_place(l,debut,fin)
80          print(pivot,l)
81          tri_aux(l,debut,pivot)
82          tri_aux(l,pivot+1,fin)
83
84  def tri_rapide_en_place(l):
85      tri_aux(l,0,len(l))
```