Mathématiques spéciales le 6 Octobre 2017

TP n°2 - Algorithme de tris - Corrigé

1. Définitions

Le **tri** est un problème fondamental en algorithmique. en général, le but de tels algorithmes est clair : ils permettent de préparer les données passées en argument afin de pouvoir rechercher facilement une information parmi celle-ci.

Ainsi, dans des données triées, il est aisé de :

1. chercher si un élément est présent ou pas ;
2. détecter les doublons ;
3. trouver une information associée à une donnée (par exemple son indice dans une liste)
4. chercher le n-ième plus grand élément d'un liste de nombres.

Exercice 1.

Écrire deux fonctions qui prennent pour argument une liste (non triée) de nombres qui réalisent respectivement les points 1) et 2). et donner leur complexité respectives.

Cet algorithme est un algorithme naïf mais sa complexité est linéaire (ce qui est tout de même bien).

```
def present(L,n):
    for i in range(len(L)):
        if L[i]==n:
            return True
    return False
```

Pour le deuxième point voici un algorithme de complexité quadratique :

```
def doublon(L):
    l=len(L)
    D=[]
    for i in range(l):
        for j in range(i+1,l):
            if L[j]==L[i]:
                D+=[[i,j]]
    return D
```

Considérons un tableau de données $L = [L_0, ..., L_{n-1}]$ d'éléments d'un ensemble E muni d'un relation d'ordre $\preccurlyeq$.

Un **algorithme de tri** a pour but de prendre ce tableau L en argument et de le retourner trié selon l'ordre $\preccurlyeq$ sur E. Pour ce faire, nous ne nous autorisons que deux opérations de base :

- Comparer deux éléments L_i et L_j à l'aide de la relation d'ordre $\preccurlyeq$.
- Permuter deux éléments L_i et L_j du tableau.

Donc si $L' = [L'_0, ..., L'_{n-1}]$ est la tableau résultat d'un algorithme de tri, il existe une permutation σ de $[\![0, n-1]\!]$ tel que, pour tout $i, j \in [\![0, n-1]\!]$,

$$L'_i = L_{\sigma(i)} \quad \text{et} \quad L'_i \preccurlyeq L'_j.$$

Ainsi pour montrer qu'un algorithme tire un tableau, il suffit de montrer que :

- qu'il applique une permutation au tableau ;
- que le tableau retourné est trié.

En fait, l'ensemble E auquel appartiennent les données du tableau n'est pas toujours muni d'un ordre total. En général, sur E, on ne dispose que d'un pré-ordre total $\preccurlyeq$, c'est-à-dire :

Pré-ordre total. On dit que $\preccurlyeq$ est un **pré-ordre total sur** E si :
- *(relation totale)* : pour tous $x, y \in E$, $x \preccurlyeq y$ ou $y \preccurlyeq x$;
- *(relation réflexive)* : pour tout $x \in E$, $x \preccurlyeq x$;
- *(relation transitive)* : pour tous $x, y, z \in E$, si $x \preccurlyeq y$ et $y \preccurlyeq z$ alors $x \preccurlyeq z$.

Mais il peut y avoir deux éléments x et y tels que $x \preccurlyeq y$ et $y \preccurlyeq x$ sans que $x = y$ (un pré-ordre n'est pas anti-symétrique).

Exercice 2. *Exemple de pré-ordre total*

Montrer que $\preccurlyeq$ est un pré-ordre total sur $\mathbb{N}^2$ où

$$(x, x') \preccurlyeq (y, y') \iff x \leq y.$$

Deux éléments x et y de l'ensemble des données E sont dit **équivalents** si $x \preccurlyeq y$ et $y \preccurlyeq x$.

Définition 1. *Algorithme de tri stable*

Soit E un ensemble de données muni d'un pré-ordre. On dit qu'un algorithme de tri est **stable** si, pour tout tableau $L = [L_0, ..., L_{n-1}]$ d'éléments de E, il préserve l'ordre des éléments équivalents i.e. si L_i et L_j sont équivalents et si $i < j$ alors dans le tableau trié par l'algorithme L_i sera encore placé avant L_j.

Nous allons tout d'abord nous intéresser aux algorithmes de tri élémentaires : le tri par sélection et le tri par insertion.

2. Le tri par sélection - *selection sort*

a. Description

Le tri par sélection correspond à un algorithme de tri parmi les plus simples et les plus naturels :

Principe du tri par sélection

- on cherche le plus petit élément du tableau ;
- on l'échange avec le premier élément du tableau ;
- on recommence le processus avec le sous-tableau restant (i.e. le tableau moins le premier élément).

On remarque que cet algorithme nécessite une fonction qui permet de trouver le plus petit élément d'un tableau à partir d'un indice donné. Programmons cette algorithme :

Exercice 3.

1. Écrire une fonction `indice_minimum(L,k)` qui prend pour arguments une liste `L` de nombres et un indice `k` et qui renvoie l'indice du minimum des éléments `L[k]`,...,`L[n-1]` (où `n` est la longueur du tableau).
2. Écrire une fonction `tri_selection(L)` qui prend pour argument une liste `L` de nombres et qui réalise l'algorithme décrit plus haut (et qui bien sûr utilise la fonction `indice_minimum` !).

```
def indice_minimum(L,k):
    i,m = k,L[k]
    for j in range(k+1,len(L)):
        if L[j]<m:
            i,m= j,L[j]
    return i

def tri_selection(L):
    for k in range(len(L)-1):
        i = indice_minimum(L,k)
        if i!= k:
            L[i],L[k] = L[k],L[i]
    return L
```

Exercice 4.

Considérons un tableau `L` contenant des chaînes de caractères de deux lettres de a à z, la première en majuscule et la deuxième en minuscule ; par exemple `L=['Bg','Dd','Ui']`.

1. En utilisant la tri par sélection, écrire deux fonctions : la première permet de trier un tableau `L` comme précédemment selon l'ordre de la deuxième lettre (en minuscule) et

la deuxième permet de trier un même tableau `L` selon l'ordre de la première lettre (en majuscule).

2. Que pensez vous de la stabilité de ces algorithmes ? Et du tri par insertion en général ?

On remarque qu'il suffit simplement de redéfinir la fonction qui donne l'indice du minimum selon l'ordre que l'on a : la fonction d'échange restera strictement la même (en changeant bien sur la fonction à l'intérieur) !

```
1   def indice_minilettre2(L,k): #retourne "l'indice du plus petit mot" selon la deuxième
        lettre à partir de l'indice k
2    i,m = k,L[k][1]
3    for j in range(k+1,len(L)):
4        if L[j][1]<m:
5            i,m= j,L[j][1]
6    return i
7
8   def indice_minilettre1(L,k): #retourne "l'indice du plus petit mot" selon la première
        lettre à partir de l'indice k
9    i,m = k,L[k][0]
10   for j in range(k+1,len(L)):
11       if L[j][0]<m:
12           i,m= j,L[j][1]
13   return i
```

Concernant la stabilité, on remarque que le tri par sélection (quand on permute le premier des minimums équivalents) n'échange pas deux éléments équivalents. C'est donc un algorithme stable.

b. Complexité du tri par sélection

Regardons la complexité temporelle en terme de comparaisons.

Exercice 5. *Complexité du tri par sélection*

Déterminer la complexité temporelle dans le pire des cas de l'algorithme de tri par sélection en terme de comparaisons.

Correction.

La fonction `indice_minimum` possède un coût en terme de comparaison égal à $n-1-k$ donc `tri_selection` à un coût total de

$$\sum_{i=0}^{n} -2(n-1-j) = \frac{n(n-1)}{2} = O(n^2).$$

Donc la complexité du tri par sélection est quadratique (bof, bof !)

3. Le tri par insertion - *insertion sort*

a. Description

Le tri par insertion est de nouveau un algorithme simple et naturel. Voici son principe : on se donne une liste `L` en argument et on procède par en utilisant un invariant de boucle :

> — La liste contenant seulement le premier élément de la liste `L` est triée ;
> — Soit $1 \leq i <$`len(L)`-1. On suppose la sous-liste des i premiers éléments de `L` triée. Alors on place le $(i+1)$ème élément de `L` dans la sous-liste des i premiers en le faisant glisser vers la gauche et en s'arrêtant :
> • soit si on arrive en première position ;
> • soit si l'élément à gauche est inférieur ou égal.

On remarque que l'on a besoin pour le tri par insertion d'une fonction qui permet d'insérer à sa place l'élément `L[k]` dans la partie du tableau `L[0:j]` (les j premiers éléments de `L` - de 0 à $j-1$) qui est supposé trié par ordre croissant.

Programmons !

Exercice 6.

1. Écrire une fonction `insertion(L,k)` qui prend pour arguments une liste `L` de nombres et un indice `k` et qui place l'élément `L[k]` au bon endroit dans la partie du tableau `L[0:j]` (qui est supposée triée bien-sûr).
2. Écrire une fonction `tri_insertion(L)` qui prend pour argument une liste `L` de nombres et qui réalise l'algorithme décrit plus haut (et qui bien sûr utilise la fonction `insertion` !).

```
def insertion(L,k):
    i=k
    while i>0 and L[i]<L[i-1]:
        L[i],L[i-1]=L[i-1],L[i]
        i-=1
    return L


def tri_insertion(L):
    for k in range(1,len(L)):
        L=insertion(L,k)
    return L
```

Question 1.

Cet algorithme est-il stable ?

b. Complexité du tri par insertion

Exercice 7.

Calculer le coût dans le pire des cas du tri par insertion en terme de comparaisons et en déduire la complexité du tri par insertion dans le pire des cas.

Correction.

Le coût est de la fonction `insertion` dans le pire des cas est égal à k comparaisons donc le coût de `tri_insertion` est

$$\sum_{i=1}^{n} -1k = \frac{n(n-1)}{2} = O(n^2).$$

Donc la complexité du tri par insertion est quadratique !

Détaillons maintenant des algorithmes de tris plus élaborés qui adoptent une démarche de type "diviser pour régner" :

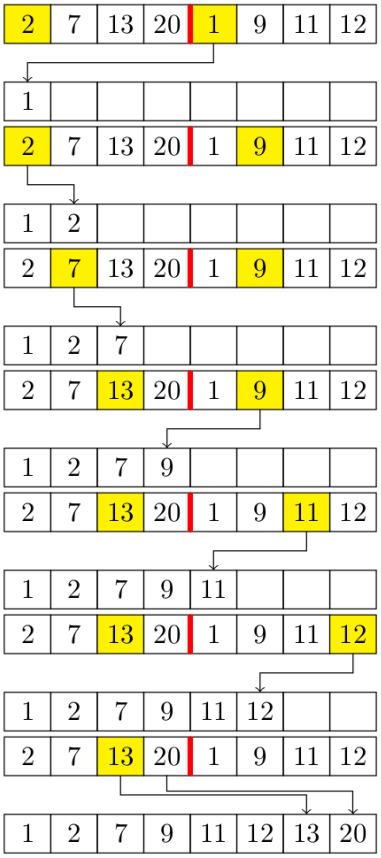
4. Le tri fusion - *merge sort*

a. Description

Comme annoncé précédemment, le tri fusion est un algorithme récursif qui adopte la technique du "diviser pour régner" ; voici la description d'une étape de la récursion du tri fusion :

- **Diviser :** Pour trier une liste de taille n, on la découpe en deux listes de taille $\frac{n}{2}$.
- **Régner :** On trie les deux moitiés de la liste (en appliquant récursivement le tri fusion).
- **Fusion :** On fusionne (en triant) les deux listes pour obtenir la liste triée.
  Par mesure de simplicité, on décompose cette dernière étape en deux :
  - on fusionne (en triant) les deux listes dans une liste temporaire ;
  - on recopie la liste temporaire dans la liste initiale pour obtenir la liste triée.

L'opération principale de cet algorithme est l'étape de fusion de deux sous-listes. Pour mieux comprendre son principe, étudions le déroulement d'une fusion de deux sous-listes triées sur un exemple concret : les cases jaunes indiquent les éléments de chaque sous-liste qui sont comparés à chaque étape et les flèches indiquent le placement du plus petit des deux dans la liste temporaire qui sert à la fusion.



Exercice 8.

1. Écrire une fonction `fusion(L,i,j,k)` qui prend pour arguments une liste `L` de nombres et trois indices i, j, k tels que $i \leq j \leq k$. Cette fonction doit fusionner (au sens vu plus haut) les deux sous-listes `L[i:j]` et `L[j:k]` dans une liste temporaire et la recopier dans la liste initiale.
2. Écrire une fonction `tri_fusion(L,i,k)` qui prend pour argument une liste `L` de nombres et deux indices `i`,`k` et qui réalise l'algorithme récursif décrit plus haut sur la liste `L[i:k]` (et qui bien sûr utilise la fonction `fusion(L,i,j,k)` !).

```
def fusion(L,i,j,k):
    temp = [None]*(k-i)
    x,y=i,j
    for t in range(k-i):
        if ((y<k and L[y]<L[x]) or x==j):
            temp[t] = L[y]
            y+=1
        else:
            temp[t] = L[x]
            x+=1
    L[i:k]=temp
```

```
12
13 def tri_fusion(L,i,k):
14     if i<k-1:
15         j=(i+k)//2
16         tri_fusion(L,i,j)
17         tri_fusion(L,j,k)
18         fusion(L,i,j,k)
19     return L
20
21 # Tri fusion comportant une sous-fonction afin d'éviter l'appel des indices de début et
       fin dans la fonction de départ
22
23 def Tri_Fusion(L):
24     def rec_tri_fusion(i,k):
25         if i<k-1:
26             j=(i+k)//2
27             tri_fusion(L,i,j)
28             tri_fusion(L,j,k)
29             fusion(L,i,j,k)
30     rec_tri_fusion(0,len(L))
31     return L
```

b. Complexité du tri fusion

Le coût est de la fonction `fusion` dans le pire des cas est égal à $k-i=n$ comparaisons pour une liste de taille n donc le coût de `tri_fusion` est donné par la relation de récurrence $c(0)=1$, $c(n)=2c(n/2)+n+1$.

La suite (u_p) de terme général $u_p=c(2^p)$ vérifie :

$$u_p = 2u_{p-1} + 2^p + 1,$$

Donc $\frac{u_p}{2^p} - \frac{u_{p-1}}{2^{p-1}} = 1 + \frac{1}{2^p}$. En sommant de $p=1$ à k, on obtient :

$$\frac{u_k}{2^k} - 1 = k + (1 - \frac{1}{2^k})$$

Et donc $u_k = O(k2^k)$.
Ainsi, pour $n=2^k$, $k2^k = n\log_2(n)$, d'où $c(n)=u_k=O(k2^k)=O(n\log(n))$. Et on admettra que cette formule reste vraie pour n quelconque :

$$c(n) = 0(n\log(n)).$$

Donc la complexité du tri fusion dans le pire des cas est quasi-linéaire (on dit également semi-logarithmique).

5. Le tri rapide - *quick sort*

a. Description

Comme le tri fusion, le tri rapide adopte la démarche "diviser pour régner" pour trier une liste. Voyons le fonctionnement de cet algorithme :

— **Pivot :** On choisit un pivot (un élément de la liste).
— **Partitionnement :** On place tous les éléments de la liste plus petits que le pivot, à gauche du pivot et tous les éléments plus grands, à droite du pivot.
— **Récursion :** On trie les parties à gauche et à droite du pivot.

Remarque : le premier pivot de l'algorithme est généralement le dernier élément de la liste.

Notre algorithme de tri rapide est composé de deux fonctions principales :
— La première est la fonction de partition qui s'occupe de déplacer les éléments à droite et à gauche du pivot ; il existe de nombreuses fonctions de partitions possibles, nous n'en verrons que deux ici, l'une utilisant des listes temporaires, et l'autre qui permet un tri "en place" c'est-à-dire, directement dans la liste d'origine.
— La seconde fonction est celle qui s'occupe des appels récursifs.

Exercice 9.

1. Écrire une fonction `partition(L,i,j)` qui réalise l'opération du pivot (ici `L[j-1]`) dans la sous-liste `L[i:j]` en utilisant deux listes temporaires : une (`gauche`) qui contiendra les éléments devant se trouver à gauche du pivot, l'autre (`droite`) contenant les éléments devant se trouver à droite du pivot. On remplace enfin la liste `L[i:j]` par `gauche+[L[j-1]]+droite` et on retourne le nouvel indice du pivot dans `L` (et pas dans `L[i:j]`).
2. Décrire sur un exemple le comportement de la fonction de partition ci-dessous :

```
def partition(L,i,j):
    pivot=L[j-1]
    x=i
    for y in range(i,j-1):
        if L[y]<=pivot:
            L[x],L[y]=L[y],L[x]
            x+=1
    L[x],L[j-1]=L[j-1],L[x]
    return x
```

3. Grâce à la fonction `partition` précédente, écrire une fonction `tri_rapide(L,i,j)` qui prend pour argument une liste `L` de nombres et deux indices `i`,`j` et qui réalise l'algorithme de tri rapide décrit plus haut.

Correction.

1.
```
def partition(L,i,j):
    pivot=L[j-1]
    gauche=[]
    droite=[]
    for k in range(i,j-1):
        if L[k]<= pivot:
            gauche.append(L[k])
        else:
            droite.append(L[k])
    L[i:j]=gauche+[pivot]+droite
    return i+len(gauche)
    return L
```

2.

3.
```
def tri_rapide(L,i,j):
    if i+1<j:
        x = partition(L,i,j)
        tri_rapide(L,i,x)
        tri_rapide(L,x+1,j)
    return L
```

b. Complexité du tri rapide

Exercice 10.

1. Déterminer (avec la fonction `partition` de la question 2. de l'exercice précédent) la complexité dans le pire des cas en terme de comparaisons de l'algorithme de tri rapide.
   Indication : le pire des cas est celui où le pivot reste toujours le dernier élément de la liste i.e. la liste est déjà triée !
2. Même question si, à chaque appel de la fonction `partition`, le pivot se place au milieu de la liste en cours.

Correction.

1. La fonction `partition(L,i,j)` a un coût de $j-1-i$ comparaisons.
   Le pire des cas est celui où x est la suite $n, ..., 0$ où n est la longueur de la liste.
   Ainsi, on a, $c(0) = 0$ et pour tout $n \in \mathbb{N}^*$,
   $$c(n) = 1 + n - 1 + c(n-1) = n + c(n);$$
   d'où,
   $$c(n) - c(n-1) = n.$$

Par suite, pour tout $n \in \mathbb{N}$,

$$c(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2).$$

Donc le tri rapide est de complexité quadratique (dans le pire des cas) !

2. Si on fait l'hypothèse précédente, on voit que `tri_rapide(L,i,x)` et `tri_rapide(L,x+1,j)` ont le même coût. Ainsi, on aura :

$$c(n) = 1 + n - 1 + 2c(n/2).$$

Donc, en reprenant la résolution de la complexité du tri fusion dans le cas particulier $n = 2^p$ (et en admettant une nouvelle fois le résultat pour n quelconque), on obtient une complexité quasi-linéaire :

$$c(n) = O(n\log(n)).$$