Tris

Alix Munier-Kordon et Maryse Pelletier

LIP6
Université P. et M. Curie
Paris

Module 2I003 Algorithmique Elémentaire

Plan du cours

Tris de comparaison et stabilité

Definition

Un algorithme de tri est dit de comparaison si il compare les éléments deux à deux pour les trier.

Definition

Un algorithme de tri est dit stable si il n'inverse pas l'ordre de deux éléments de même clef.

Autres caractéristiques

- Complexité pire-cas, meilleur cas, moyenne.
- Difficulté algorithmique.
  le tri par insertion, le tri par sélection et le tri à bulles sont dits *simples* car faciles à comprendre et à programmer.
- Structure linéaire utilisée : tableau, listes simplement ou doublement chaînées, fichiers.
  Certains tris (par exemple le tri par insertion ou le Quicksort) s'adaptent facilement pour trier des listes chaînées.

Complexité minimale d'un tri de comparaison

Theorem

Tout algorithme de tri de comparaison est de complexité d'au minimum $\mathcal{O}(n \log n)$ *dans le pire des cas.*

Impossible d'espérer un tri de comparaison de complexité inférieure à $\mathcal{O}(n \log n)$!

Principe du tri par insertion

Soit n le nombre d'éléments du tableau à trier.

1. Au début de l'étape $j \in \{1, \cdots, n-1\}$, $tab[0 \cdots j-1]$ est trié;
2. on insère alors $tab[j]$ à sa place dans $tab[0 \cdots j-1]$;
3. $tab[0 \cdots j]$ est alors un tableau trié.

Algorithme de tri par insertion

```
def insertionSort(tab):
    j = 1
    n = len(tab)
    while j != n:
        # inserer tab[j] dans tab[0...j-1]
        # a sa place
        insertionElem(tab, j)
        j = j + 1
```

Principe de insertionElem

En entrée, $j \in \{1 \cdots, n-1\}$ et $tab[0 \cdots j-1]$ est un tableau trié. insertionElem insère la valeur $tab[j]$ à sa place dans $tab[0 \cdots j-1]$.

1. On sauvegarde $tmp = tab[j]$;
2. on parcourt le tableau $tab[0 \cdots j-1]$ en décalant chaque élément d'une case vers la droite;
3. on s'arrête dès que l'on a trouvé la place de tmp.

Fonction insertionElem

```
def insertionElem(tab, j):
    tmp = tab[j]
    i = j-1
    while i > -1 and tab[i] > tmp:
        tab[i+1] = tab[i]
        i = i - 1
    tab[i+1] = tmp
```

Exécution de insertionElem(tab, 4)

$j = 4 \qquad tmp = tab[j] = 5 \qquad i = j - 1 = 3$

$i = 3$	1	3	6	7	5		$7 > 5$
$i = 2$	1	3	6	7	7		$6 > 5$
$i = 1$	1	3	6	6	7		$3 \leq 5$
	1	3	5	6	7		$tab[i + 1] = tmp$

Complexité et stabilité du tri par insertion

- Le tri par insertion est stable ;
- Dans le meilleur des cas, le tableau est déja trié : Complexité de $\Omega(n)$;
- Dans le pire des cas, le tableau est en ordre décroissant : Complexité de $\mathcal{O}(n^2)$.

Complexité et stabilité des tris simples pour des tableaux

Tri	Complexité	Stable	
Insertion	$\Omega(n)/\mathcal{O}(n^2)$	Oui	–
Bulles	$\Theta(n^2)$	Oui	Voir TD 4
Sélection	$\Theta(n^2)$	Oui	Voir TD4

Principe du Quicksort sur les listes

Soient L une liste non vide et $x_1 = L[0]$. x_1 est appelé le “Pivot” ;

1. Eclater les éléments de $L \setminus \{x_1\}$ en deux sous-listes $L1$ et $L2$ telles que :

$$\forall y \in L1, y < x_1 \text{ et } \forall y \in L2, y \geq x_1 .$$

2. Si L est vide, retourner vide. Sinon, retourner la liste

$$L' = Quicksort(L1).(x_1).Quicksort(L2).$$

La notation $\ell.\ell'$ désigne la liste constituée de la concaténation des listes ℓ et ℓ'.

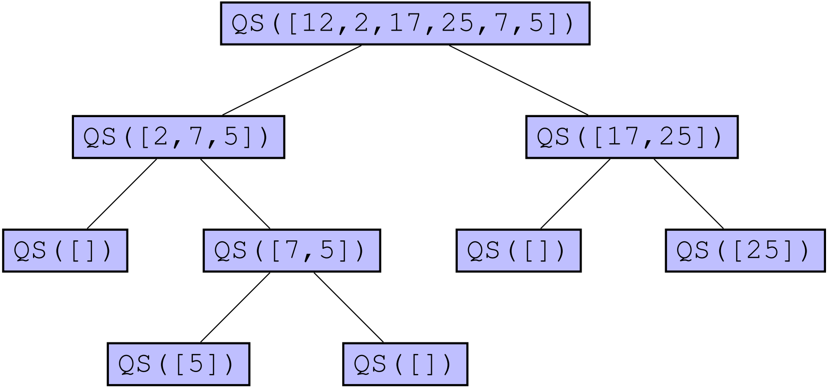
Quicksort

```
def Quicksort(L):
    if (len(L)>1):
        L1=[] ; L2=[] ; L3=[]
        L3.append(L[0])
        Eclatement(L, L1, L2)
        return Quicksort(L1)+L3+Quicksort(L2)
    return L
```

Fonction d'éclatement du Quicksort

```
def Eclatement(L, L1, L2):
        x=L[0]
        for y in L[1:]:
                if (y<x):
                        L1.append(y)
                else:
                        L2.append(y)
```

Arbre des exécutions de `Quicksort([12,2,17,25,7,5])`



Complexité et stabilité du Quicksort pour une liste circulaire doublement chaînée

1. La complexité de l'éclatement est en $\Theta(n)$;
2. Si la liste est déjà triée, la complexité est en $\mathcal{O}(n^2)$;
3. Dans le meilleur des cas, si la liste est divisée en deux à chaque appel, la complexité est en $\Omega(n \log n)$;
4. On peut démontrer (mais pas dans ce cours), que la complexité en moyenne du Quicksort est en $n \log n$.
5. Est-ce-que le Quicksort est stable ?

Les monotonies

Soit $L = (1, 5, 8, 3, 7, 2, 12, 4, 9, 15)$ une liste d'entiers à trier en ordre croissant.

Definition

Une monotonie de L est une sous-liste maximale d'éléments consécutifs en ordre croissant.

Les monotonies de L sont les sous-listes $(1, 5, 8)$, $(3, 7)$, $(2, 12)$ et $(4, 9, 15)$.

Opération de fusion de deux listes

Consiste à construire une seule liste à partir de deux listes $L1$ et $L2$ en prenant en premier de manière récursive, l'élément

$$\min(L1[0], L2[0]).$$

La fusion de $L1 = (1, 5, 8, 2, 12)$ avec $L2 = (3, 7, 4, 9, 15)$ renvoie

$$L = (1, 3, 5, 7, 4, 8, 2, 9, 12, 15)$$

```
def fusion(L1,L2):
    if (L1 == []):
        return L2
    if (L2 == []):
        return L1
    if (L1[0]<= L2[0]):
        R=fusion(L1[1: ], L2)
        R.insert(0, L1[0])
        return R
    R=fusion(L1, L2[1: ])
    R.insert(0, L2[0])
    return R
```

Evolution du nombre de monotonies

Theorem

Soit L la liste obtenue par fusion des sous-listes L1 et L2 non vides. Si m_1, m_2 et m représentent le nombre de monotonies des listes L1, L2 et L, alors $m < m_1 + m_2$.

Pour $L1 = (1, 5, 8, 2, 12)$, $m_1 = 2$.
Pour $L2 = (3, 7, 4, 9, 15)$, $m_2 = 2$.
La fusion donne la liste $L = (1, 3, 5, 7, 4, 8, 2, 9, 12, 15)$ avec $m = 3$.

Principe du tri fusion itératif

1. Eclatements/fusions en deux sous-listes selon les monotonies jusqu'à obtenir une liste triée.
2. La fonction d'éclatement de L en deux sous-listes $L1$ et $L2$ ne doit pas augmenter le nombre global de monotonies : la première monotonie de L est placée dans $L1$, la seconde dans $L2$, la troisième dans $L1$...etc...

Par exemple, l'éclatement de $L = (1, 5, 8, 3, 7, 2, 12, 4, 9, 15)$ permet d'obtenir les deux sous-listes $L1 = (1, 5, 8, 2, 12)$ et $L2 = (3, 7, 4, 9, 15)$.

```
def eclatement ( L, L1, L2) :
    listeL1 = True
    pred = L[0]
    L1.append(pred)
    i = 1
    while (i<len(L)):
        if (pred > L[i]):
            listeL1=not listeL1
        if listeL1:
            L1.append(L[i])
        else:
            L2.append(L[i])
        pred=L[i]
        i = i+1
```

```
»> triFusionMonotonies(L)
L = [1, 5, 8, 3, 7, 2, 12, 4, 9, 15]
L1 = [1, 5, 8, 2, 12]
L2 = [3, 7, 4, 9, 15]
L = [1, 3, 5, 7, 4, 8, 2, 9, 12, 15]
L1 = [1, 3, 5, 7, 2, 9, 12, 15]
L2 = [4, 8]
L = [1, 3, 4, 5, 7, 2, 8, 9, 12, 15]
L1 = [1, 3, 4, 5, 7]
L2 = [2, 8, 9, 12, 15]
L = [1, 2, 3, 4, 5, 7, 8, 9, 12, 15]
L1 = [1, 2, 3, 4, 5, 7, 8, 9, 12, 15]
L2 = []
L = [1, 2, 3, 4, 5, 7, 8, 9, 12, 15]
```

```
def triFusionMonotonies(L):
    if len(L)>1:
        L1 = []
        L2 = []
        eclatement(L,L1,L2)
        while (len(L2)>0):
            L=fusion(L1,L2)
            L1 = []
            L2 = []
            eclatement(L, L1, L2)
        return L1
    return L
```

Convergence et complexité du tri fusion itératif pour une liste circulaire doublement chaînée

1. Le nombre de monotonies de L décroît strictement en fonction du nombre d'itérations et L possède au plus $n = |L|$ monotonies. Donc, l'algorithme converge en au plus n itérations.
2. La fusion est en $\mathcal{O}(n)$ et l'éclatement est en $\Theta(n)$. Donc, le tri fusion est en $\mathcal{O}(n^2)$.
3. Est-ce que le tri par fusion de monotonies est stable ?
4. Peut-on utiliser le tri par fusion de monotonies pour trier un fichier ?

Tri par dénombrement

1. Supposons que les entiers $p \in L$ à trier vérifient $0 \leq p \leq maxV$.
2. On remplit, $\forall i \in \{0, \cdots, maxV\}$, $Comptage[i] =$ nombre d'occurrences de i dans L.
3. On re-construit L à partir de $Comptage$.

Tri par dénombrement

1. Soit $L = (6, 3, 2, 4, 1, 0, 5, 4, 6, 3, 2, 0, 4)$;
2. $maxV = 6$ et $Comptage[0 \cdots 6] = (2, 1, 2, 2, 3, 1, 2)$;
3. On obtient $L = (0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6)$.

```
def triDenombrement(L):
    maxV=elemMax(L)
    Comptage = [0]*(maxV+1)
    for elem in L :
        Comptage[elem]=Comptage[elem]+1
    R=[]
    for i in range(maxV+1):
        for j in range(Comptage[i]):
            R.append(i)
    return R
```

Complexité et taille mémoire du tri par dénombrement

1. Complexité si L est une liste chaînée circulaire :

$$\Theta(n + \max\{maxV, n\}) \equiv \Theta(\max\{maxV, n\}).$$

2. Taille mémoire :

$$\Theta(\max\{maxV, n\}).$$

Est-ce que le tri par dénombrement est stable ? Est-ce un tri de comparaison ?

Conlusion

1. De nombreuses façons de trier une structure linéaire. Il faut choisir en fonction de la structure et des valeurs à trier, mais aussi de la complexité recherchée.
2. Deux tris classiques restent à voir : Radix (TD 6) et Tri par Tas (cours/TD sur les arbres).