IFT2015 hiver 2010 — Notes de cours

Miklós Csűrös

17 février 2010

8 Méthodes élémentaires de tri

On a un fichier d'éléments avec **clés comparables** — on veut les ranger selon l'ordre les clés. Clés comparables en Java :

```
public interface Comparable<T>
{
      int compareTo(T autre_objet);
}
```

`x.compareTo(y)` retourne une valeur négative si `x` précède `y`, ou positive si `x` suit `y`, selon l'ordre «naturel» des éléments.

Tri **externe** : fichier stocké partiellement ou entièrement en mémoire externe (disque) — accès à mémoire externe est coûteux...

Tri **interne** : tout le fichier est en mémoire (représenté par un tableau ou une liste chaînée) :

Méthode de tri interne	**tableau**	**liste chaînée**
tri par sélection (*selection sort*)	oui	oui
tri par insertion (*insertion sort*)	oui	oui
tri par fusion (*Mergesort*)	oui	oui
tri par tas (*Heapsort*)	oui	non
tri rapide (*Quicksort*)	oui	non

8.1 Tri par sélection

Idée : boucler : en itération i, on s'assure que $A[0..i]$ contient les i éléments les plus petits, triés.

Algo TRI-SELECTION($A[0 \dots n-1]$)
S1 **pour** $i \leftarrow 0, 1, \dots, n-2$ **faire**
S2 minidx $\leftarrow i$
S3 **pour** $j \leftarrow i+1, \dots, n-1$ **faire si** $A[j] < A[\text{minidx}]$ **alors** minidx $\leftarrow j$
// *maintenant* $A[\text{minidx}] = \min\{A[i], A[i+1], \dots, A[n-1]\}$
S4 **si** $i \neq$ minidx **alors** échanger $A[i] \leftrightarrow A[\text{minidx}]$

Complexité de calcul : $\Theta(n^2)$ pour tout A.

⋆ comparaison d'éléments [ligne S3] : $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$ fois ;

⋆ échange d'éléments [ligne S4] : $\leq (n-1)$ fois $\Rightarrow$ pas nécessairement une mauvaise idée si l'échange est beaucoup plus coûteux que la comparaison !

8.2 Tri par insertion

Idée : boucle $i \leftarrow 0, \ldots, n-1$; en itération i, on s'assure que le sous-tableau $A[0..i]$ est trié.

```
   Algo TRI-INSERTION(A[0 ... n − 1])
I1 pour i ← 1, ..., n − 1 faire
I2     j ← i − 1
I3     tandis que j ≥ 0 et A[j] > A[j + 1] faire
I4         échanger A[j + 1] ↔ A[j]
I5         j ← j − 1
```

Complexité — dépend de l'ordre des éléments au début :

meilleur cas (déjà trié) : $n-1$ comparaisons et aucun échange $\Rightarrow$ très utile si A est «presque trié» au début

pire cas (trié en ordre décroissant) : $\frac{n(n-1)}{2}$ comparaisons et échanges

moyen cas (permutation aléatoire) : $\Theta(n^2)$

Génie algorithmique :

- ⋆ placer le minimum en $A[0]$ — il sert comme sentinelle.
- ⋆ remplacer échange par décalage.

```
    Algo TRI-INSERTION(A[0 ... n − 1]) — modifiée
IM1 minidx ← 0 ; pour i ← 1, ..., n − 1 si A[i] < A[minidx] alors minidx ← i
IM2 échanger A[0] ↔ A[minidx]          // sentinelle : on ne devra pas vérifier j ≥ 0 en IM5
IM3 pour i ← 1, ..., n − 1 faire
IM4     j ← i − 1 ; a ← A[i]
IM5     tandis que A[j] > a faire A[j + 1] ← A[j] ; j ← j − 1
IM6     A[j + 1] ← x
```

8.3 Tri de liste chaînée

On peut implanter la logique de ces tris élémentaires avec des listes chaînées aussi. Soit L une liste chaînée, avec une sentinelle (nœud factice) head à la tête, et null à la fin. Chaque nœud N possède la valeur N.val et une référence à son successeur N.prochain. L'algorithme ci-dessous performe un tri par sélection.

```
    Algo TRI-SELECTION-LISTE(L)
SL1 I ← head
SL2 tandis que I.prochain ≠ null faire
SL3     min ← I ; J ← I.prochain
SL4     tandis que J.prochain ≠ null faire
SL5         si J.prochain.val < min.prochain.val alors min ← J
SL6     // maintenant min.prochain est le nœud avec val minimale dans la sous-liste après I
SL7     si min ≠ I alors
SL8         M ← min.prochain ; min.prochain ← M.prochain        // délétion après min
SL9         M.prochain ← I.prochain ; I.prochain ← M            // insertion après I
SL10    I ← I.prochain
```