

## 10 Tri par fusion et tris élémentaires

### 10.1 Tri

On a un fichier d'éléments avec **clés comparables** — on veut les ranger selon l'ordre des 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 ( <i>selection sort</i> )	oui	oui
tri par insertion ( <i>insertion sort</i> )	oui	oui
tri par fusion ( <i>Mergesort</i> )	oui	oui
tri par tas ( <i>Heapsort</i> )	oui	non
tri rapide ( <i>Quicksort</i> )	oui	non

### 10.2 Fusion de deux tableaux triés.

**Algo FUSION**( $A[0..n-1], B[0..m-1]$ ) (de type `Comparable[]`, triés)

```
F1 initialiser  $C[0..n+m-1]$  // on place le résultat dans C
F2  $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$  // i est l'indice dans A; j est l'indice dans B
F3 while  $i < n$  et  $j < m$  do
F4   if  $A[i] \leq B[j]$  then  $C[k] \leftarrow A[i]; i \leftarrow i + 1$  else  $C[k] \leftarrow B[j]; j \leftarrow j + 1$ 
F5    $k \leftarrow k + 1$ 
F6 while  $i < n$  do  $C[k] \leftarrow A[i]; i \leftarrow i + 1; k \leftarrow k + 1$ 
F7 while  $j < m$  do  $C[k] \leftarrow B[j]; j \leftarrow j + 1; k \leftarrow k + 1$ 
```

**Théorème 10.1.** L'algorithme FUSION calcule la fusion de deux tableaux triés en un temps  $\Theta(n+m)$ , avec  $n+m$  comparaisons [d'éléments entre les listes] au pire.

### 10.3 Tri par fusion

Tri par fusion (*mergesort*) utilise le principe de **diviser pour régner** dans une procédure récursive.

$W_{(fr)}$

$$\text{tri}(A[0..n-1]) = \begin{cases} A & \{n < 2\} \\ \text{fusion}(\text{tri}(A[0..\lfloor n/2 \rfloor]), \text{tri}(A[\lfloor n/2 \rfloor + 1..n-1])) & \{n \geq 2\} \end{cases}$$

**Algo** MERGESORT( $A[0..n-1], g, d$ )

// récursion pour trier le sous-tableau  $A[g..d-1]$  : appel initial avec  $g = 0, d = n$

M1 **if**  $d - g < 2$  **then return**

// cas de base : tableau vide ou un seul élément

M2  $m \leftarrow \lfloor (d + g) / 2 \rfloor$

//  $m$  est au milieu

M3 MERGESORT( $A, g, m$ )

// trier partie gauche

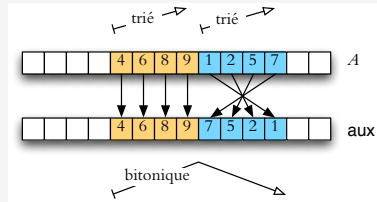
M4 MERGESORT( $A, m, d$ )

// trier partie droite

M5 FUSION( $A, g, m, d$ )

// fusion des résultats

Typiquement, on utilise un seul tableau auxiliaire pour faire la fusion. Avec un arrangement **bitonique**, on peut simplifier les conditions dans la boucle de la fusion.



**Algo** FUSION( $A[], g, m, d$ ) // fusion «en place» pour  $A[g..m-1]$  et  $A[m..d-1]$

F1 **for**  $i \leftarrow g, g+1, \dots, m-1$  **do**  $\text{aux}[i] \leftarrow A[i]$

F2 **for**  $j \leftarrow m, m+1, \dots, d-1$  **do**  $\text{aux}[m+d-1-j] \leftarrow A[j]$

F3  $i \leftarrow g; j \leftarrow d-1; k \leftarrow g$

// meilleure condition :  $i < m$

F4 **while**  $k < d$  **do**

F5 **if**  $\text{aux}[i] \leq \text{aux}[j]$  **then**  $A[k] \leftarrow \text{aux}[i]; i \leftarrow i+1; k \leftarrow k+1$

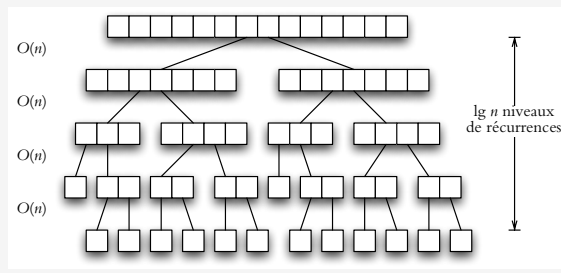
F6 **else**  $A[k] \leftarrow \text{aux}[j]; j \leftarrow j-1; k \leftarrow k+1$

## 10.4 Temps de calcul du tri par fusion

Dans la fusion, on combine les deux tableaux triés en un troisième en un temps linéaire (Théorème 9.1). Le temps de calcul s'écrit donc comme

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T_{\text{fusion}}(\lfloor n/2 \rfloor, \lceil n/2 \rceil) + O(1) \quad (10.1)$$

où  $T_{\text{fusion}}(k, m) = \Theta(k + m)$  est le temps pour fusionner deux tableaux de tailles  $k$  et  $m$ .



On a donc  $T_{\text{fusion}}(k, m) = \Theta(k + m)$  en (9.1), qui mène à la récurrence classique

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n).$$

La solution de la récurrence est  $T(n) = \Theta(n \log n)$  (voir Théorème 3A.5). On peut voir cette solution directement en dessinant l'**arbre de récurrences** : on a  $\lceil \lg n \rceil$  niveaux et  $O(n)$  travail (fusions) à chaque niveau.

## 10.5 Tris élémentaires

### Tri par sélection

Boucler  $i \leftarrow 0, \dots, n-2$ ; propriété invariante à la fin d'itération  $i$  :

$\mathcal{P}(i) = \langle A[0..i] \rangle$  contient les  $i$  éléments les plus petits, dans l'ordre correct»

**Algo** TRI-SELECTION( $A[0 \dots n-1]$ )

S1 **for**  $i \leftarrow 0, 1, \dots, n-2$  **do**

S2  $\text{minidx} \leftarrow i$  // (on cherche l'élément minimal en  $A[i+1..n-1]$ )

S3 **for**  $j \leftarrow i+1, \dots, n-1$  **do if**  $A[j] < A[\text{minidx}]$  **then**  $\text{minidx} \leftarrow j$   
// maintenant  $A[\text{minidx}] = \min\{A[i], A[i+1], \dots, A[n-1]\}$

S4 **if**  $i \neq \text{minidx}$  **then** échanger  $A[i] \leftrightarrow A[\text{minidx}]$  // maintenant  $\mathcal{P}(i)$  est vraie

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

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

★ échange d'éléments [ligne S4] :  $\leq (n-1)$  fois  $\Rightarrow$  très efficace si l'échange est beaucoup plus coûteux que la comparaison !

### Tri par insertion

Boucler  $i \leftarrow 1, \dots, n-1$  ; propriété invariante à la fin d'itération  $i$  :

$\mathcal{Q}(i)$  = «le sous-tableau  $A[0..i]$  contient les éléments originaux du même préfixe, dans l'ordre trié.»

**Algo** TRI-INSERTION( $A[0 \dots n-1]$ ) // première solution

I1 **for**  $i \leftarrow 1, \dots, n-1$  **do**

I2      $j \leftarrow i-1$  ; **while**  $j \geq 0$  et  $A[j] > A[j+1]$  **do** échanger  $A[j+1] \leftrightarrow A[j]$  ;  $j \leftarrow j-1$

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 servira comme sentinelle pour simplifier la condition d'arrêt dans la boucle interne.

★ remplacer échange par décalage.

**Algo** TRI-INSERTION( $A[0 \dots n-1]$ ) // plus efficace

IM1 minidx  $\leftarrow 0$  ; **for**  $i \leftarrow 1, \dots, n-1$  **if**  $A[i] < A[\text{minidx}]$  **alors** minidx  $\leftarrow i$

IM2 échanger  $A[0] \leftrightarrow A[\text{minidx}]$  // sentinelle : on ne devra pas vérifier  $j \geq 0$  en IM4

IM3 **for**  $i \leftarrow 1, \dots, n-1$  **do**

IM4      $j \leftarrow i-1$  ;  $a \leftarrow A[i]$  ; **while**  $A[j] > a$  **do**  $A[j+1] \leftarrow A[j]$  ;  $j \leftarrow j-1$

IM5      $A[j+1] \leftarrow a$

### Tri hybride

En pratique, le tri par fusion performe le mieux dans une **approche hybride** : la récursion est trop coûteuse pour les petits sous-tableaux, et le tri par insertion est plus rapide. Pour cette raison, il vaut planter le tri par fusion avec un seuil  $\ell > 1$  sur les petits sous-tableaux. On passe les sous-tableaux de taille  $d-g < \ell$  en Ligne M1 directement à un tri par insertion.

## 10.6 Tri de liste chaînée

### Tris élémentaires

On peut implanter la logique des 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  $N.next$ . L'algorithme ci-dessous performe un tri par sélection.

```
Algo TRI-SELECTION-LISTE(head)           // tri par sélection pour la liste qui commence avec head
SL1   $I \leftarrow head$ 
SL2  while  $I.next \neq null$  do
SL3     $min \leftarrow I$ ;  $J \leftarrow I.next$ 
SL4    while  $J.next \neq null$  do
SL5      if  $J.next.val < min.next.val$  then  $min \leftarrow J$ 
SL6       $J \leftarrow J.next$ 
SL7    // maintenant  $min.next$  est le nœud avec  $val$  minimale dans la sous-liste après  $I$ 
SL8    if  $min \neq I$  then
SL9       $M \leftarrow min.next$ ;  $min.next \leftarrow M.next$            // suppression après  $min$ 
SL10      $M.next \leftarrow I.next$ ;  $I.next \leftarrow M$            // insertion après  $I$ 
SL11    $I \leftarrow I.next$ 
```

### Tri par fusion

Pour adapter les récurrences à des listes chaînées, il faut séparer une liste en deux moitiés — une implantation simple ajoute des éléments en alternant entre les deux «demi-listes».

```
MERGESORT-LIST( $A$ )           // tri d'une liste chaînée qui commence avec nœud  $A$ 
ML1  if  $A = null$  then return           // une liste vide est triée par défaut
ML2  initialiser sous-listes vides  $B_1 \leftarrow null$  et  $B_2 \leftarrow null$ 
ML3   $N_1 \leftarrow B_1$ ;  $N_2 \leftarrow B_2$            //  $N_i$  est le dernier nœud en  $B_i$ 
ML4  while  $A \neq null$  do
ML5     $N \leftarrow A$ ;  $A \leftarrow A.next$ ;  $N.next \leftarrow null$ 
ML6    if  $N_i = null$  then  $B_i \leftarrow N$            // premier nœud sur liste  $i$ 
ML7    else  $N_i.next \leftarrow N$            // insertion après  $N_i$ 
ML8     $N_i \leftarrow N$ ;  $i \leftarrow 3 - i$            //  $i = 1, 2, 1, 2, \dots$ 
ML9  return FUSION(MERGESORT-LIST( $B_1$ ), MERGESORT-LIST( $B_2$ ))
```

**Exercice 10.1.** ► Adaptez l'algorithme de FUSION de §9.2 à des listes chaînées pour la ligne ML9. L'algorithme prend deux listes comme arguments, dans lesquelles les éléments sont dans l'ordre croissant, et retourne une troisième liste qui comprend tous les nœuds des listes d'entrée, dans l'ordre croissant. Faites attention au cas de listes vides à l'entrée.