

10 Tris élémentaires et le tri par fusion

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 et interne. Un algorithme de **tri externe** sert à ordonner les éléments d'un fichier stocké partiellement ou entièrement en mémoire externe (disque). Il existe des algorithmes spécialisés pour trier des fichiers trop grand pour la mémoire vive : en une telle application, on veut minimiser l'accès à mémoire externe. Le **tri interne** se fait sur un fichier stocké entièrement en mémoire vive. Le plus souvent, on considère le tri d'un tableau $A[0..n-1]$; parfois, on peut adapter un tel algorithme aux listes chaînées aussi.

Temps de calcul. On caractérise le temps de calcul d'un algorithme de tri par le nombre d'opérations de **comparaison** entre éléments et d'**échange** d'éléments. En une caractérisation plus générale d'un tri de tableau, on compte le nombre d'**accès** aux cellules.

Espace de travail. Un aspect important d'efficacité est l'usage de mémoire pendant le tri. Noter que cela inclut toutes les variables locales sur la pile d'appel dans récurrences. Un algorithme *en place* utilise $O(1)$ mémoire à part du fichier d'entrée. (En particulier, un tel algorithme ne crée pas de copies du tableau à trier.)

Méthode de tri interne	liste chaînée	comparaisons	espace de travail
tri par sélection (<i>selection sort</i>)	oui	$\sim n^2/2$ (toujours)	$O(1)$
tri par insertion (<i>insertion sort</i>)	oui	$\sim n^2/2$ (pire), $\Theta(n^2)$ (moyenne), $\sim n$ (meilleur)	$O(1)$
tri par fusion (<i>Mergesort</i>)	oui	$\sim n \lg n$ (toujours)	$\Theta(n)$
tri par tas (<i>Heapsort</i>)	non	$\sim 2n \lg n$ (pire), $\Theta(n \lg n)$ (toujours)	$O(1)$
tri rapide (<i>Quicksort</i>)	non	$\sim 2n \ln n$ (moyenne), $O(n^2)$ (pire)	$O(\log n)$

10.2 Tri par sélection

Idée : séparer les éléments en un segment trié ($[0..i-1]$, $i = 0$ au début avec aucun élément) et un segment non-trié ($[i..n-1]$) ; mettre toujours l'élément minimal non-trié à la fin des éléments triés.

$W_{(fr)}$

Algo TRI-SELECTION($A[0..n-1]$)

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

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

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 $A[i] \leftrightarrow A[\text{minidx}]$

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 !

10.3 Tri par insertion

W^(fr)

Idée : séparer les éléments en un segment trié ($[0..i - 1]$, $i = 0$ au début avec aucun élément) et un segment non-trié ($[i..n - 1]$) ; insérer toujours l'élément $A[i]$ parmi les éléments triés.

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

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 $A[j + 1] \leftarrow a$

10.4 Fusion de deux tableaux triés.

On peut fusionner deux listes (implantées comme tableaux ou listes chaînées), par la récurrence

$$\text{fusion}(A, B) = \begin{cases} B & \text{si } A \text{ est vide} \\ A & \text{si } B \text{ est vide} \\ A[0] \odot \text{fusion}(A[1..], B) & \text{si } A[0] \leq B[0] \\ B[0] \odot \text{fusion}(A, B[1..]) & \text{si } B[0] < A[0] \end{cases}$$

où \odot dénote la concaténation.

Dans le cas de deux tableaux, une solution itérative utilise deux indices parcourant les deux listes.

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** $C[k] \leftarrow A[i]; i \leftarrow i + 1$

F5 **else** $C[k] \leftarrow B[j]; j \leftarrow j + 1$

F6 $k \leftarrow k + 1$

F7 **while** $i < n$ **do** $C[k] \leftarrow A[i]; i \leftarrow i + 1; k \leftarrow k + 1$

F8 **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 $n + m - 1$ comparaisons d'éléments au pire.

10.5 Tri par fusion

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

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

Algo MERGESORT($A[0..n-1], g, d$) // appel initial avec $g = 0, d = n$

// récursion pour trier le sous-tableau $A[g..d-1]$

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

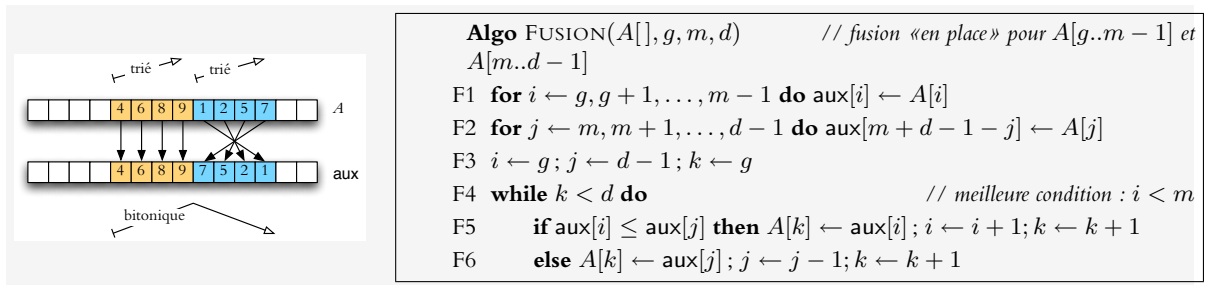
M5 FUSION(A, g, m, d) // fusion des résultats

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 implanter 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.

Gestion d'espace auxiliaire

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.

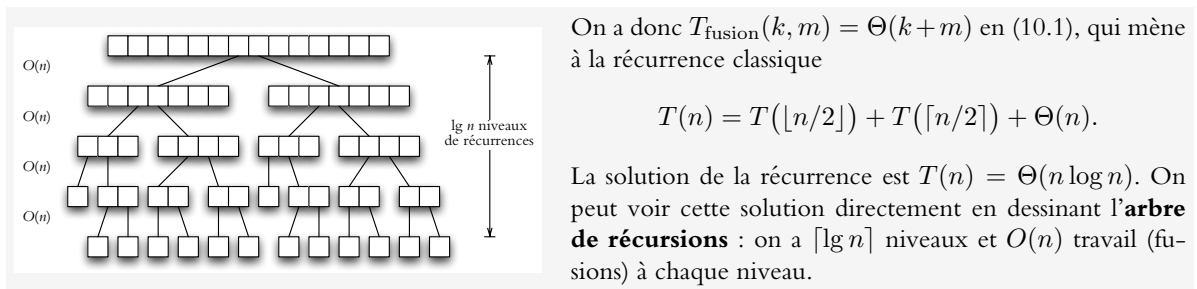


10.6 Temps de calcul du tri par fusion

Dans la fusion, on combine les deux tableaux triés en un troisième en un temps linéaire (Théorème 10.1). 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 .



10.7 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 sur 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. ► Adapter l'algorithme de FUSION de §10.4 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. ♠

Exercice 10.2. ► Montrer comment implanter tri par insertion sur une liste chaînée. ♠