12 Tris élémentaires et le tri par fusion

12.1 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 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 (*selection sort*)	oui	$\sim n^2/2$ (toujours)	$O(1)$
tri par insertion (*insertion sort*)	oui	$\sim n^2/2$ (pire), $\Theta(n^2)$ (moyenne), $\sim n$ (meilleur)	$O(1)$
tri par fusion (*Mergesort*)	oui	$\sim n \lg n$ (toujours)	$\Theta(n)$
tri par tas (*Heapsort*)	non	$\sim 2n \lg n$ (pire), $\Theta(n \log n)$ (toujours)	$O(1)$
tri rapide (*Quicksort*)	non	$\sim 2n \ln n$ (moyenne), $O(n^2)$ (pire)	$O(\log n)$

12.2 Tri par sélection

W(fr)

Idée : séparer les élements 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.

Algo TRI-SELECTION($A[0 \ldots n-1]$)
S1 **for** $i \leftarrow 0, 1, \ldots, n-2$ **do**
S2 minidx $\leftarrow i$ *// (on cherche l'élément minimal en $A[i..n-1]$)*
S3 **for** $j \leftarrow i+1, \ldots, n-1$ **do if** $A[j] < A[\text{minidx}]$ **then** minidx $\leftarrow j$
// maintenant $A[\text{minidx}] = \min\{A[i], A[i+1], \ldots, A[n-1]\}$
S4 **if** $i \neq$ minidx **then** échanger $A[i] \leftrightarrow A[\text{minidx}]$

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

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

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

12.3 Tri par insertion

W(fr)

Idée : séparer les élements 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($A[0\ldots n-1]$) *// première solution*
I1 **for** $i \leftarrow 1,\ldots,n-1$ **do**
I2 $\quad j \leftarrow i-1$; **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.

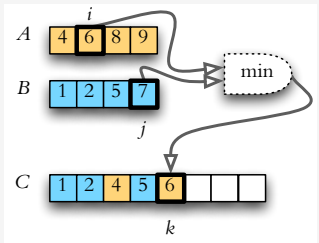
Algo TRI-INSERTION($A[0\ldots n-1]$) *// plus efficace*
IM1 minidx $\leftarrow 0$; **for** $i \leftarrow 1,\ldots,n-1$ **if** $A[i] < A[$minidx$]$ **alors** minidx $\leftarrow i$
IM2 échanger $A[0] \leftrightarrow A[$minidx$]$ *// sentinelle : on ne devra pas vérifier $j \geq 0$ en IM4*
IM3 **for** $i \leftarrow 1,\ldots,n-1$ **do**
IM4 $\quad j \leftarrow i-1$; $a \leftarrow A[i]$; **while** $A[j] > a$ **do** $A[j+1] \leftarrow A[j]$; $j \leftarrow j-1$
IM5 $\quad A[j+1] \leftarrow a$

12.4 Fusion de deux tableaux triés.

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

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

où $\odot$ dénote la concaténation.



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

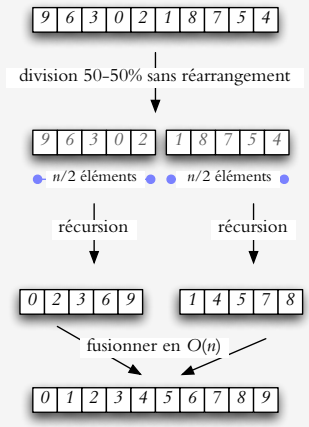
```
     Algo FUSION(A[0..n−1], B[0..m−1]) (de type Comparable[], triés)
F1   initialiser C[0..n+m−1]                    // on place le résultat dans C
F2   i ← 0; j ← 0; k ← 0                        // i est l'indice dans A ; j est l'indice dans B
F3   while i < n et j < m do
F4       if A[i] ≤ B[j] then C[k] ← A[i]; i ← i+1
F5       else C[k] ← B[j]; j ← j+1
F6       k ← k+1
F7   while i < n do C[k] ← A[i]; i ← i+1; k ← k+1
F8   while j < m do C[k] ← B[j]; j ← j+1; k ← k+1
```

Théorème 12.1. *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.*

12.5 Tri par fusion

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

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



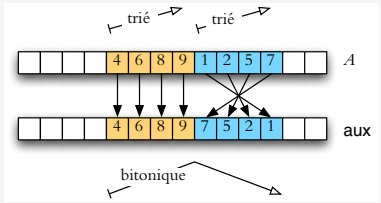

```
     Algo MERGESORT(A[0..n−1], g, d)          // appel initiel avec g = 0, d = n
     // récursion pour trier le sous-tableau A[g..d−1]
M1   if d − g < 2 then return                 // cas de base : tableau vide ou un seul élément
M2   m ← ⌊(d+g)/2⌋                            // m est au milieu
M3   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 couteuse 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.



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

F1 **for** $i \leftarrow g, g+1, \ldots, m-1$ **do** aux$[i] \leftarrow A[i]$

F2 **for** $j \leftarrow m, m+1, \ldots, d-1$ **do** aux$[m+d-1-j] \leftarrow A[j]$

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

F4 **while** $k < d$ **do** *// meilleure condition :* $i < m$

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

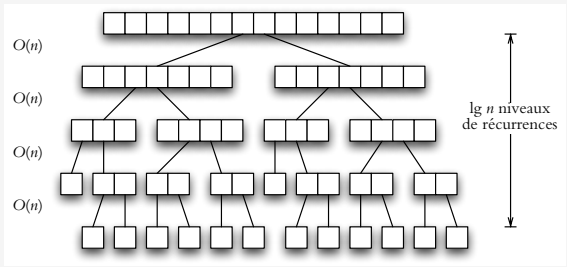
F6 **else** $A[k] \leftarrow$ aux$[j]\,;\, j \leftarrow j-1; k \leftarrow k+1$

12.6 Temps de calcul du tri par fusion

Dans la fusion, on combine les deux tableaux triés en un troisième en un temps linéaire (Théorème 12.1). Le temps de calcul s'écrit donc comme

$$T(n) = T\big(\lfloor n/2 \rfloor\big) + T\big(\lceil n/2 \rceil\big) + T_{\text{fusion}}\big(\lfloor n/2 \rfloor, \lceil n/2 \rceil\big) + O(1) \tag{12.1}$$

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 (12.1), qui mène à la récurrence classique

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

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