IFT2015 hiver 2010 — Notes de cours

Miklós Csűrös

21 février 2010

9 Tri par fusion

9.1 Tri d'un tableau

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

```
   Algo MERGESORT(A[0..n − 1])
M1 si n < 2 alors retourner A        // cas de base : tableau vide ou un seul élément
M2 B1 ← MERGESORT(A[0..⌊n/2⌋ − 1])    // trier la moitié gauche
M3 B2 ← MERGESORT(A[⌊n/2⌋..n − 1])    // trier la moitié droite
M4 B ← FUSION(B1, B2)                // fusion des résultats
M5 retourner B
```

Dans la fusion, on combine les deux tableaux triés en un troisième. Le temps de calcul est donc

$$T(n) = T\big(\lfloor n/2 \rfloor\big) + T\big(\lceil n/2 \rceil\big) + T_{\text{fusion}}\big(\lfloor n/2 \rfloor, \lceil n/2 \rceil\big) + O(1) \tag{9.1}$$

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

9.2 Fusion de deux tableaux triés.

```
   Algo FUSION(A[0..n − 1], B[0..m − 1]) (de type Comparable[], triés)
F1 intialiser C[0..n + m − 1]          // on place le résultat dans C
F2 i ← 0; j ← 0                        // i est l'indice dans A ; j est l'indice dans B
F3 pour k ← 0, 1, . . . , n + m − 1 faire
F4     si j ≥ m ou A[i] ≤ B[j] alors C[k] ← A[i]; i ← i + 1
F5     sinon C[k] ← B[j]; j ← j + 1
```

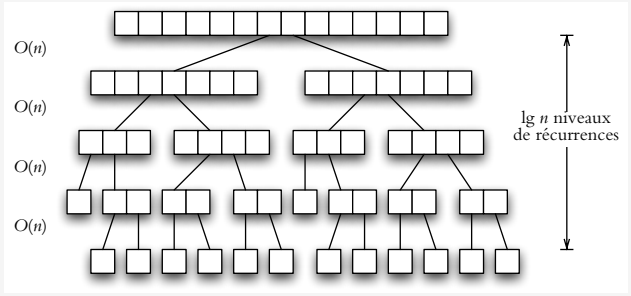
La clé à l'efficacité du tri est qu'on fusionne deux tableaux en un temps linéaire $\Theta(n+m)$. On a $(n+m)$ affectations, et $(n + m - 1)$ comparaisons $A[i]$-$B[j]$ (au pire cas).

9.3 Tri d'une liste chaînée

Il n'est pas difficile d'adapter l'algorithme FUSION aux listes chaînées. Pour adapter l'algorithme MERGESORT, il faut séparer une liste en lignes M2-M3. Une solution simple est d'ajouter des éléments aux listes B_1, B_2 en altérnant entre les deux :

```
tandis que A n'est pas vide
    B1.enqueue(A.dequeue())
    si A n'est pas vide alors B2.enqueue(A.dequeue())
```

9.4 Temps de calcul du tri par fusion



On a donc $T_{\text{fusion}}(k,m) = \Theta(k+m)$ en (9.1), 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 $T(n) = \Theta(n \log n)$ (voir Théorème 5 en §2.9). 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.

9.5 Fusion de $m \geq 3$ listes

En général, on peut fusionner m listes (*multi-way fusion*) de longueur totale n en $O(n \log m)$ temps.

Solution par récurrence. On peut utiliser le principe de **diviser pour régner** en une procédure récursive.

```
   Algo MULTI-FUSION-REC(A_0[], A_1[], ..., A_{m-1}[]) (tableaux triés)
MR1  si m = 1 alors retourner A_0                          // cas de base : un seul tableau
MR2  si m > 1 alors
MR3      B_1 ← MULTI-FUSION-REC(A_0, ..., A_{⌊m/2⌋-1})     // fusion d'une moitié
MR4      B_2 ← MULTI-FUSION-REC(A_{⌊m/2⌋}, ..., A_{m-1})   // fusion de l'autre moitié
MR5      retourner FUSION(B_1, B_2)                        // combiner le résultat des deux moitiés
```

Solution avec file à priorités. On peut modifier l'algorithme FUSION pour traiter plus que deux tableaux. On utilise les indices $i_0, \ldots, i_{m-1}$: en chaque itération, il faut choisir le minimum de $A_0[i_0]$, $A_1[i_1], \ldots, A_{m-1}[i_m - 1]$. Pour cela, on utilise une **file à priorités** F. La liste F contient les paires $(j, A_j[i_j])$, et permet de retirer le minimum par l'opération F.deleteMin(). Un nouvel élément est ajouté par F.add(j,a). La liste F contient m éléments au plus à tout temps.

```
   Algo MULTI-FUSION(A_0[0..n_0 − 1], A_1[0..n_1 − 1], ..., A_{m-1}[0..n_{m-1} − 1]) (tableaux triés)
MF1  initialiser la file à priorités F ← ∅
MF2  pour j ← 0, ..., m − 1 faire i_j ← 0 ; F.add(j, A_j[i_j])
MF3  n ← n_0 + n_1 + ··· + n_{m-1} ; initialiser C[0..n − 1]          // résultat dans C
MF4  pour k ← 0, 1, ..., n − 1 faire
MF5      (j, a) ← F.deleteMin()          // le minimum est dans tableau A_j, avec valeur a = A_j[i_j]
MF6      C[k] ← a
MF7      i_j ← i_j + 1 ; si i_j < n_j alors F.add(j, A[i_j])   // prendre le prochain élément du tableau A_j
MF8  retourner C
```

En une implantation naïve, F est représenté par une liste chaînée : on trouve le minimum en $\Theta(m)$ (parcours complète). Cela donne $\Theta(nm)$ pour le temps de calcul : c'est évidemment beaucoup pire que $O(n \log m)$. On verra bientôt qu'on peut implanter une file à priorités à l'aide d'une structure nommée **tas**, où l'opération deleteMin (ligne MF5) et add (lignes MF2, MF7) s'exécutent en $O(\log m)$ temps $\Rightarrow$ temps total est $O(n \log m)$. En un cas extrême, on a $n = m$, avec des listes de longueur 1. Dans d'autre mots, un tas permet un tri efficace de $O(n \log n)$: c'est l'idée de base du tri par tas.