

Diviser pour régner

Les tris

Michel Van Caneghem

Novembre 2004

De l'importance du tri

C'est l'opération de base sur des données.

- Il est intéressant de trier pour faire plus facilement des recherches : par exemple ordre alphabétique.**
- Souvent grande quantité d'information à trier, donc les performances sont importantes.**
- Dans beaucoup d'algorithmes il faut maintenir de manière dynamique des listes triés, pour trouver rapidement les plus petits et plus grands éléments.**
- Ces méthodes concernent beaucoup d'autres algorithmes.**

Quelques chiffres

Par exemple dans le cas du graphique :

- Nombre de points dans une page A4 :**
- format $A4 = \frac{1}{16}m^2 = 625cm^2$.**
- 300points/pouce = 13950 points/cm².**
- donc 1 page A4 = $625 \times 13950 = 8718750$ points**

- Algorithme en $N^2 \implies 7.6 \times 10^{13}$ opérations**
- Algorithme en $N^{1.5} \implies 2.57 \times 10^{10}$ opérations**
- Algorithme en $N \log N \implies 1.39 \times 10^8$ opérations**
- Algorithme en $N \implies 8.71 \times 10^6$ opérations**

©2002 Google - Nombre de pages Web recensées par Google : 3,083,324,652.

Quelques tris

Bubble sort : $N^2/2$ comparaisons et $N^2/2$ échanges, en moyenne et dans le pire des cas.

Insertion sort : $N^2/4$ comparaisons et $N^2/8$ échanges, en moyenne et le double dans le pire des cas. Linéaire si le fichier est presque trié.

Selection sort : $N^2/2$ comparaisons et N échanges en moyenne. Intéressant pour trier de gros fichiers avec de petites clés.

Shell sort : Complexité inconnue mais inférieure à $N^{3/2}$ comparaisons pour des incréments de 1,4,13,40,121.

et bien d'autres : quicksort, heapsort, mergesort, radix-sort, bucket-sort.

Le théorème principal

Soit la récurrence suivante :

$$T(n) = aT(n/b) + f(n), \quad a, b \geq 1$$

- ① **Si** $f(n) = O(n^{\log_b a - e})$, $e > 0$ **alors** : $T(n) = \Theta(n^{\log_b a})$.
- ② **Si** $f(n) = \Theta(n^{\log_b a})$ **alors** : $T(n) = \Theta(n^{\log_b a} \log n)$.
- ③ **Si** $f(n) = \Omega(n^{\log_b a + e})$, $e > 0$
et si $af(n/b) \leq cf(n)$, $c < 1$ **alors** : $T(n) = \Theta(f(n))$.

On dit que $f(x) = \Theta(g(x))$ **si** :

$$\exists c_1 \neq 0, c_2 \neq 0, x_0, \quad x > x_0 \rightarrow c_1 g(x) < f(x) < c_2 g(x)$$

Le théorème principal (2)

① $T(n) = 4T(n/2) + n$ alors : $f(n) = n^{\log_2 4 - 1}$ et d'après (1) :

$$T(n) = \Theta(n^2)$$

② $T(n) = 2T(n/2) + n$ alors : $f(n) = n^{\log_2 2}$ et d'après (2) :

$$T(n) = \Theta(n \log n)$$

③ $T(n) = 3T(n/4) + n$ alors : $f(n) = n^{\log_4 3 + \log_4(4/3)}$ et
 $3f(n/4) = 3/4n$ et d'après (2) :

$$T(n) = \Theta(n)$$

Quicksort en Java

```
public static void triRapide(int g, int d) {
    int v, i, j, t;
    if (d > g) {
        v = a[d]; i = g - 1; j = d;
        for (;;) {
            while (a[++i] < v) ;
            while (j > 0 && a[--j] > v) ;
            if (i >= j) break;
            t = a[i]; a[i] = a[j]; a[j] = t;
        }
        t = a[i]; a[i] = a[d]; a[d] = t;
        triRapide(g, i-1);
        triRapide(i+1, d);
    }
}
```

Quicksort

Le cas le plus défavorable est celui où le pivot choisi est le plus petit élément de la liste. La partition coupe la liste en un morceau de longueur 1 et un autre de longueur $n - 1$. D'où l'équation de récurrence :

$$L_n = L_{n-1} + n - 1 \quad L_0 = 0$$

$$L_n = n(n - 1)/2$$

La complexité de quicksort est donc $O(n^2)$ dans le pire cas.

Remarque : Ce n'est pas le cas pour **heapsort et **mergesort**.**

Quicksort (2)

Le cas le plus favorable est celui où le pivot choisi est l'élément médian de la liste. La partition coupe la liste en deux morceaux de longueur $n/2$. D'où l'équation de récurrence :

$$L(n) = 2L(n/2) + n - 1 \quad L(1) = 1$$

nous venons de voir que la solution est :

$$L(n) = O(n \log n)$$

Complexité en moyenne : La notion de moyenne peut s'envisager en considérant les $n!$ ordres possibles de n nombres et en faisant la moyenne du nombre de comparaisons.

Quicksort (3)

En moyenne, il y a toujours $n - 1$ comparaisons pour partitionner la liste. Intéressons nous à la position i du pivot. Cette position a une probabilité de $1/n$ d'être choisie donc :

mais :

$$F_n = n + 1 + \frac{1}{n} \sum_{i=1}^n (F_{i-1} + F_{n-i})$$

$$\sum_{i=1}^n F_{i-1} = F_0 + F_1 + \cdots + F_{n-1}$$

$$\sum_{i=1}^n F_{n-i} = F_{n-1} + F_{n-2} + \cdots + F_0$$

donc en multipliant par n : $nF_n = n(n + 1) + 2 \sum_{i=1}^n F_{i-1}$

Quicksort (4)

$$nF_n = n(n+1) + 2 \sum_{i=1}^{n-1} F_{i-1} + 2F_{n-1}$$

$$(n-1)F_{n-1} = (n-1)n + 2 \sum_{i=1}^{n-1} F_{i-1}$$

en soustrayant on obtient :

$$nF_n = (n+1)F_{n-1} + 2n \quad F_1 = 2$$

et on obtient :

$$\frac{F_n}{n+1} = \frac{F_{n-1}}{n} + \frac{2}{n+1}$$

Quicksort (5)

$$\frac{F_n}{n+1} = \frac{F_1}{2} + 2 \sum_{i=3}^{n+1} \frac{1}{i}$$

$$F_n = 2(n+1) \left(-1 + \sum_{i=1}^n \frac{1}{i} \right)$$

on sait que :

$$\sum_{i=1}^n \frac{1}{i} = H_n = \ln n + 0.57721$$

$$F_n = 2n \ln n - 0.846n$$

Le nombre moyen de comparaison effectuées par l'algorithme Quicksort trier une suite de n éléments est de l'ordre de $2n \log n$. [demo]

Quicksort (6)

Les performances globales (pour une machine) :

	Nb Opérations	Coût unitaire
partitionnement	n	35
comparaisons	$2n \ln n - 0.846n$	4
échanges	$0.333n \ln n - 1.346n$	11

En tout : $11.66n \ln n + 19.2n$

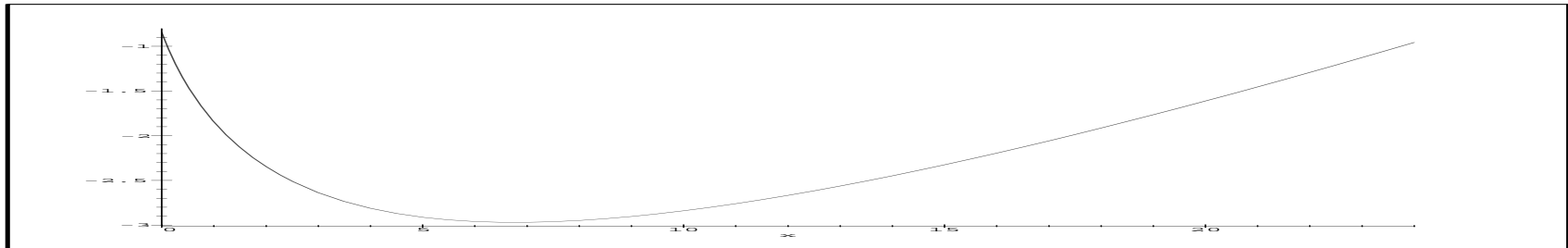
n	10	100	1000	10000	100000	1000000
$11.66n \ln n$	267	5369	80E3	104E4	130E5	156E6
$19.2n$	192	1920	19E3	19E4	19E5	19E6
total	459	7289	99E3	123E4	149E5	175E6

Quicksort amélioré

Si on fait la coupure en dessous de m éléments en utilisant le tri par insertion, on a :

$$F_n = \begin{cases} n + 1 + \frac{1}{n} \sum_{i=1}^n (F_{i-1} + F_{n-i}) & \text{pour } n > m \\ n(n-1)/4 & \text{pour } n \leq m \end{cases}$$

Ce qui peut s'écrire : $F_n = 2n \ln n + f(m)n$ Il ne reste plus qu'à trouver le minimum de la fonction f .



Quicksort amélioré (2)

La formule de récurrence pour le tri rapide *médiane de trois* :

$$F_n = n + 1 + \sum_{i=1}^n \frac{(n-i)(i-1)}{\binom{n}{3}} (F_{i-1} + F_{n-i})$$

ou $\binom{n}{3} = \mathbb{C}_n^3 = n(n-1)(n-2)/6$ On trouve :

$$F_n = \frac{12}{7}(n+1)\left(H_{n+1} - \frac{23}{14}\right), \quad n \geq 6$$

C'est la version de quicksort la plus utilisée.

Randomize Quicksort

On tire le pivot au hasard !! Analysons le nombre de comparaisons en moyenne :

Soit s_1, s_2, \dots, s_n les éléments de S triés. On définit des variables aléatoires X_{ij} pour $j > i$ de la manière suivante :

$$\begin{aligned} X_{ij} &= 1 \text{ si } s_i \text{ et } s_j \text{ sont comparés pendant l'algorithme} \\ &= 0 \text{ sinon} \end{aligned}$$

On veut calculer :

$$C = \sum_{i=1}^n \sum_{j>i} X_{ij} \qquad E[C] = \sum_{i=1}^n \sum_{j>i} E[X_{ij}]$$

Il faut donc évaluer $E[X_{ij}]$.

Randomize Quicksort (2)

La seule manière de comparer s_i et s_j c'est que soit l'un, soit l'autre soient pivots. Ce qui veut dire que si on considère l'intervalle :

$$[s_i, s_{i+1}, \dots, s_j]$$

aucun autre élément de cet intervalle sauf les bornes ne sont choisis avant s_i ou s_j . Comme il y a $j - i + 1$ élément dans cet intervalle et que les éléments sont choisis au hasard :

$$p_{ij} = E[X_{ij}] = \frac{2}{j - i + 1}$$

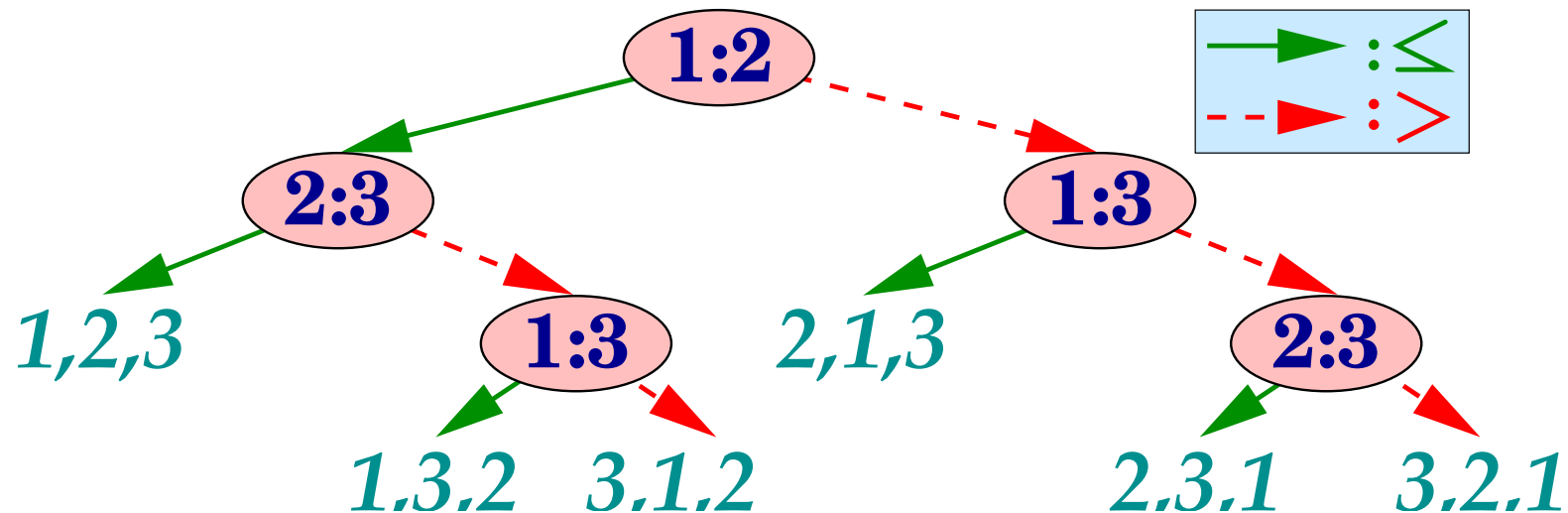
Randomize Quicksort (3)

$$\begin{aligned} E[C] &= \sum_{i=1}^n \sum_{j>i} E[X_{ij}] = \sum_{i=1}^n \sum_{j>i} \frac{2}{j-i+1} \\ &= \sum_{i=1}^n (2H_{n-i+1} - 1) = 2 \sum_{i=1}^n H_i - n < 2nH_n - n \end{aligned}$$

Ce n'est pas meilleur que Quicksort normal, mais plus régulier. Pour tomber dans le pire des cas, il faut beaucoup de malchance !!

Tri par comparaison

Une comparaison a deux possibilités : \leq ou $>$. Un tri peut donc être vu comme un arbre de décision binaire.



Comme il y a $n!$ feuilles, il faut au moins la hauteur de l'arbre comme nombre de comparaisons : $\log(n!) = O(n \log n)$.

Un tri en temps linéaire

Sans comparaison !

Problème : Trier un fichier de N enregistrements dont les clefs sont des entiers entre 0 et $M - 1$:

```
for j:=0 to M-1 do nombre[j]:=0;
for i:=1 to N do nombre[C[i]]:=nombre[C[i]]+1;
for j:=1 to M-1 do nombre[j]:=nombre[j-1]+nombre[j];
for i:=N downto 1 do begin
  R[nombre[C[i]]:=C[i];
  nombre[C[i]]:=nombre[C[i]]-1;
end;
```

Visiblement ce tri est linéaire et stable (préserve l'ordre des enregistrements de clés égales), il demande $2N$ passages sur le fichier.

Radix Sorting

On profite de la représentation binaire des clefs dans un mot machine pour éviter de faire des comparaisons.

- **Radix exchange sort** : Similaire à Quicksort (même complexité). On partitionne le fichier en utilisant récursivement les bits de la clé de la gauche vers la droite.
- **Straight Radix sort** : On examine maintenant les b bits de la droite vers la gauche en rangeant chaque fois au début les clés ayant le bit correspondant à zéro. Au bout de b passages le fichier est trié. Similaire à la méthode utilisée dans le temps avec des cartes perforées.

Complexité : Pour trier N enregistrements avec des clés de b bits il faut b/m passages si on utilise 2^m compteurs.

Radix Sorting

(5) 101	(2) 010	(4) 100	(1) 001
(2) 010	(4) 100	(5) 101	(2) 010
(4) 100	(2) 010	(1) 001	(2) 010
(3) 011	(5) 101	(5) 101	(3) 011
(7) 111	(3) 011	(2) 010	(4) 100
(1) 001	(7) 111	(2) 010	(5) 101
(5) 101	(1) 001	(3) 011	(5) 101
(2) 010	(5) 101	(7) 111	(7) 111

C'est une méthode pour trier un jeu de carte

Bucket sort

On veut trier une liste de tailles N dont les éléments sont répartis aléatoirement dans l'intervalle $[0..M - 1]$. On découpe cet intervalle en K classes de taille M/K .

- ➔ On calcule combien il y a d'élément dans chaque classe : en moyenne N/K . (division ou décalage)
- ➔ On range les éléments dans la bonne classe.
- ➔ On trie les éléments de chaque classe (Quicksort ou insertion sort).

$$\begin{aligned}C(N) &= AN + K(2N/K \log(N/K) - 0.846N/K) \\ &= AN + 2N \log N - 0.846N - 2N \log K\end{aligned}$$

Bucket sort (2)

Un exemple :

- ⇒ N est compris entre 1 000 000 et 5 000 000 d'éléments ;
- ⇒ $M = 2^{31} = 2147483648$;
- ⇒ $K = 32768$ (décalage de 16) (2 tableaux de cette taille) ;
- ⇒ N/K entre 30 et 152.

$$T_{\mu s} = 0.26N \log N - 1.17N - 7 \times 10^{-6}$$

Soit 2 fois plus rapide que qsort !

Selection

Trouver le k ème élément dans l'ordre d'une liste.

- ❁ **On trie la liste et on prend le k ème élément (Complexité $O(n \log n)$)**
- ❁ **Mais, cela peut se faire en un temps linéaire.**
- ❁ **par exemple on peut trouver le 1er (minimum) ou le dernier (maximum) élément d'une liste en un temps linéaire.**

L'élément **médian d'une liste de taille n impair est un élément tel qu'il est plus grand que $(n - 1)/2$ éléments de la liste et plus petit que $(n - 1)/2$ éléments de la liste.**

Selection (1)

```
fonction SELECT(T, i, j, k) {  
    p = PIVOTER(T, i, j);  
    if (k <= p) SELECT(T, i, p, k);  
    else SELECT(T, p + 1, j, k);  
}
```

Si la taille du sous tableau de l'appel récursif ne dépasse pas αn avec $\alpha < 1$, et si la complexité de PIVOTER est en $O(n)$ alors la complexité de la fonction SELECT est en $O(n)$.

$$T(n) = an + T(\alpha n)$$

Selection (2)

La fonction PIVOTER est définie de la manière suivante :

- ① **On découpe le tableau T en blocs B_i de 5 éléments.**
- ② **Pour chacun de ces blocs B_i on cherche le médian m_i**
- ③ **On applique la fonction SELECT pour chercher le médian de la liste $m_1, m_2, \dots, m_{n/5}$.**

- ① **Chaque médian m_i est plus grand que 2 éléments et plus petit que 2 éléments**
- ② **Le médian des médians (p) est plus grands que $n/10$ médians et plus petits que $n/10$ médians**
- ③ **Donc p est plus grands que $2n/10$ ou plus petit que $2n/10$ éléments**
- ④ **Donc la liste est découpé en deux parties, dont la plus grande est au plus de longueur $8n/10$. Donc $\alpha < 0.8$**

Conclusion sur le tri

Les résultats précédents concernent des performances en moyenne : attention à ne pas se trouver dans le mauvais cas d'une méthode.

La méthode de tri à utiliser dépend fortement de la nature, de la répartition et du nombre de données à trier. Cela dépend également du support du fichier (ex : bandes magnétiques, disques optiques, ...).

Il y a des mauvaises méthodes, mais il n'y a pas la bonne méthode universelle.

Heapsort, Shellsort et Quicksort sont des choix raisonnables. Ne pas oublier bucket sort!!!