Département d'Informatique
Institut des Sciences et Technologie
Centre Universitaire d'El-Oued

Complexité des algorithmes

AbdelKamel BEN ALI

Centre Universitaire d'El-Oued

Mai 2011

Master 1 d'informatique – SDIA

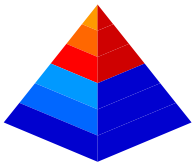
Extraits du cours introductif

Plan du cours complet

- **Première partie I — Complexité théorique et expérimentale**
  - Chap. 1 Eléments de base [30%]
  - Chap. 2 Récurrence et complexité [15%]
  - **Chap. * Diviser pour régner – Les tris**

  [6 Cours/Exercices + TP]
- **Deuxième partie II — Classes de problèmes & réductions**
  - Chap. 3 Problèmes $\mathcal{NP}$-complets
  - Chap. 4 Techniques de démonstration de la $\mathcal{NP}$-complexité
  - Chap. 5 Les autres classes de complexité

  [6-7 Cours/Exercices + TP]



Cela pose de nombreuses questions ... et demande pas mal de savoir-faire

Comment concevoir un algorithme ?

Schémas d'algorithmes, Algorithmic design patterns
"Diviser pour régner"
Programmation dynamique
Algorithmes gloutons

But de ce cours

Illustration des concepts vus précédemment :

- Complexité algorithmique
- Récurrence et complexité
- Stratégie “diviser pour régner”

et c'est en étudiant plusieurs méthodes de tri.

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, classement par date, par taille
- 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 tris

Algorithmes simples
- Tri bulle (Bubble sort)
- Tri par sélection (Selection sort)
- Tri par insertion (Insertion sort)

Algorithmes plus rapides
- Tri fusion (mergesort)
- Tri rapide (quicksort)
- Tri tas (heapsort)

et bien d'autres
- Tri par base (radixsort)
- Tri par paquets (bucket-sort)

Tri bulle

Principe

- Faire remonter la plus grosse valeur en fin de tableau
- Puis la deuxième
- ... etc.

```
void triBulle(float t[]) {
   int i, k;
   for (i = (n - 1); i > 0; i--)
      for (k = 0; k < i; k++)
         if (t[k] > t[k + 1]) echanger(t, k, k + 1);
}
```

Invariant Après chaque itération, les i plus grands éléments sont à leur place définitive

Complexité

- $n(n-1)/4$ échanges en moyenne et le double dans le pire des cas
- $n(n-1)/2$ comparaisons dans tous les cas

Tri par sélection

```
void triSelection(float t[]) {
    int i, j, min;
    for (i = 0; i < n - 1; ++i) {
        min = i;
        for (j = i + 1; j < n; ++j)
            if (t[j] < t[min]) min = j;
        if (min > i) echanger(t, min , i);
    }
}
```

Invariant A la fin d'un pas d'exécution de la boucle, les i plus petits éléments sont bien placés

Complexité

- cas pire : $n(n-1)/2$ affectations $+$ $n-1$ échanges
- dans tous les cas : $n(n-1)/2$ comparaisons

Intéressant pour trier de gros fichiers avec de petites clés.

Tri par insertion

```
void triInsertion(float t[]) {
    int i, k ;
    float x ;
    for (i = 1; i < n; ++i) {
        k = i - 1;
        x = t[i];
        while ((k >= 0) && (t[k] > x))
            t[k + 1] = t[k], k--;
        t[k + 1] = x;
    }
}
```

Invariant A la fin d'un pas d'exécution de la boucle, les i premiers éléments sont triés (mais ils ne sont pas forcément à leur place)

Complexité

- cas pire : $n(n-1)/2$ "déplacements"
- cas moyen : environ la moitié

Ce tri est linéaire si le fichier est presque trié

Version récursive

Il existe une version récursive de ces trois algorithmes,
Par exemple :

```
void triInsertionRec (float t[], int n) {
    if (n > 1) {
        triInsertionRec (t, n - 1);
        /* insérer t[n - 1] à sa place dans la suite
        triée obtenue */
        // à programmer ...
    }
}
```

Rappel : Le théorème principal

Récurrences de partitions

Récurrences utiles — **Le théorème principal**

Soit la récurrence suivante :

$$\begin{cases} t(n_0) = d \\ t(n) = at(n/b) + f(n), \quad n > n_0 \end{cases}$$

En général : $f(n) = cn^k$, et n/n_0 puissance de b

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

Tri fusion (Merge sort): principe

- Ce tri utilise le principe "diviser pour régner"

Méthode

1. On divise le tableau en deux
2. On trie les deux sous-tableaux
3. On produit un tableau trié contenant tous les éléments des deux sous-tableaux triés

Tri fusion : Algorithme

```
void triFusion(int T[], int i, int j) {
    int m;
    if (i < j) {
        m = (i + j - 1) / 2;
        triFusion(T, i, m);
        triFusion(T, m + 1, j);
        interClass(T, i, m, j);
    }
}
```

Reste à programmer l'interclassement ...

Tri fusion : algorithme (2)

Voici une méthode optimale pour interclasser les deux sous tableaux triés :

```
void interClass(int T[], int l, int m, int r) {
    int R[n];
    int i, j, k;
    for (i = g; i <= m; i++)
        R[i] = T[i];
    for (j = m + 1; j <= d; j++)
        R[j] = T[m + d + 1 - j];
    i = g; j = d; k = g;
    while ((i <= m) && (k <= d)) {
        if (R[i] < R[j]) T[k] = R[i], i++;
        else T[k] = R[j], j--;
        k++;
    }
}
```

Complexité

- cas pire : $(n-1)$ comparaisons
- meilleur cas: $n/2$ comparaisons

Tri fusion — Complexité : pire cas

Pour simplifier, on suppose que le tableau est de taille $n = 2^k$

$$c(n) = 2 \times c(n/2) + tfusion(n)$$

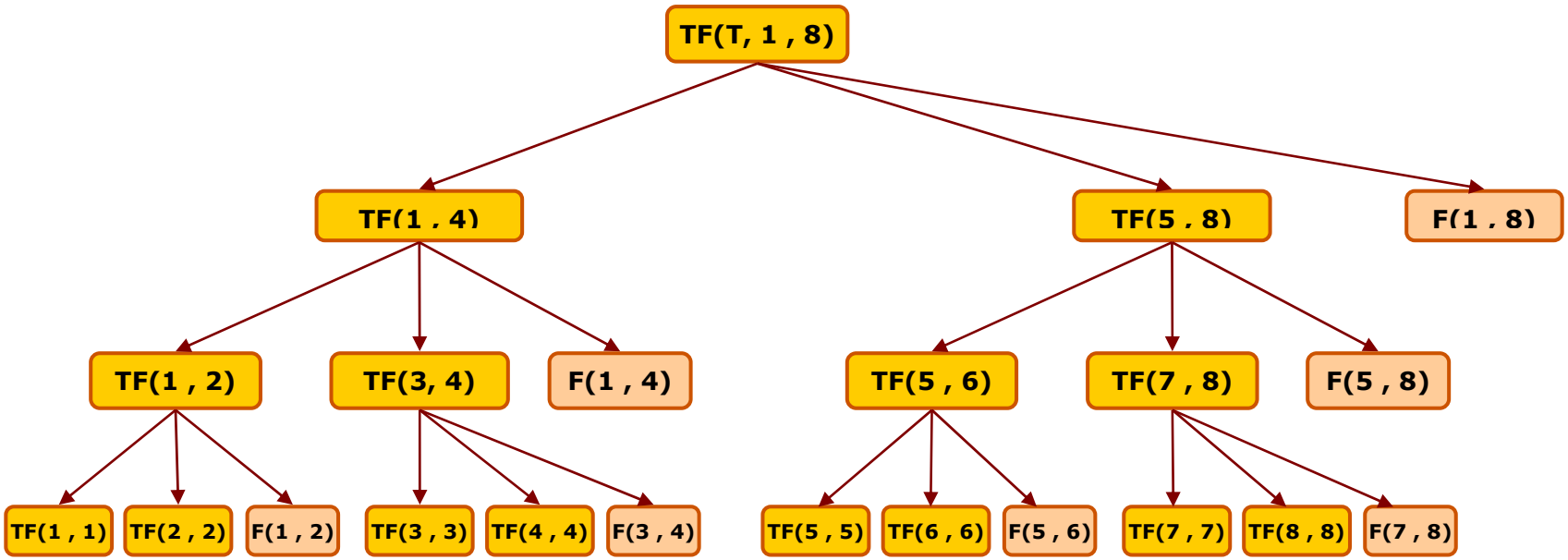
où $tfusion$: le coût de l'algorithme d'interclassement

$$tfusion(n) = n - 1 \quad \text{comparaisons}$$

- Solution asymptotique d'après le théorème (2) : $c(n) = \Theta(n \log(n))$
- Solution exacte : $c(n) = n \log(n) + 1 - n$ [Calculer]

Tri fusion — Complexité : pourquoi $O(n \log(n))$

On dessine l'arbre des appels pour $n = 8$.



Tri fusion — Complexité : meilleur cas

Pour simplifier, on suppose que le tableau est de taille $n = 2^k$

$$c(n) = 2 \times c(n/2) + tfusion(n)$$

$$tfusion(n) = n/2 \quad \text{comparaisons}$$

- Solution asymptotique d'après le théorème (2) : $c(n) = \Theta(n \log(n))$
- Solution exacte : $c(n) = \frac{1}{2} n \log(n)$ [Calculer]

Tri rapide (Quick sort)

C'est un tri :

- sans tableau auxiliaire
- et de complexité moyenne $O(nlog(n))$

Utilise une approche "diviser pour régner" :

- Diviser le tableau en deux sous-tableaux
- Trier chacun des sous tableaux
- Fusionner le tout

Tri rapide : principe

1. On prend le premier élément du tableau, $T[1]$, appelé *pivot*
2. On met tous les éléments du tableau $> T[1]$ à droite, et tous élements $< T[1]$ à gauche
3. On met le pivot au milieu
4. On applique TriRapide aux deux sous-tableaux obtenus

Remarque : pas besoin de "fusionner" après les deux sous-tableaux triès.

Tri rapide : une réalisation en C

```
void triRapide(int a[], 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(a, g, i - 1);
        triRapide(a, i + 1, d);
    }
}
```

Tri rapide : Complexité

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 morceau de longueur $n-1$.
D'où l'équation de récurrence :

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

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

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

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

Tri rapide : Complexité (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 :

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

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.

Tri rapide : Complexité (3)

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

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

mais :

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

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

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

Tri rapide : Complexité (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 :

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

et on obtient :

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

Tri rapide : Complexité (5)

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

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

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 pour trier une suite de n éléments est de l'ordre de $2n \log n$. **[demo]**

Tri rapide : Complexité (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

Tri rapide — Variantes : Tri rapide amélioré

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

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

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

Tri rapide — Variantes : Tri rapide amélioré (2)

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

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

où $\binom{n}{3} = \mathbf{C}_n^3 = n(n-1)(n-2)/6$ on trouve :

$$F_n = \frac{12}{7}(n+1)(H_{n+1} - \frac{23}{44}), \qquad n \geq 6$$

C'est la version de quicksort[1] la plus utilisée.

[1] Cette version améliorée de quicksort est implémentée par J. Bentley et D. Mcilroy en 1993 pour la librairie standard C "`stdlib.h`".

Tri rapide — Variantes : Randomize Quicksort

On tire le pivot au hasard !! Analysons le nombre de comparaisons en moyenne :
Soit $s_1, s_2, \ldots, s_n$ 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 $E[X_{ij}]$.

Tri rapide — Variantes : 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 :

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

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 :

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

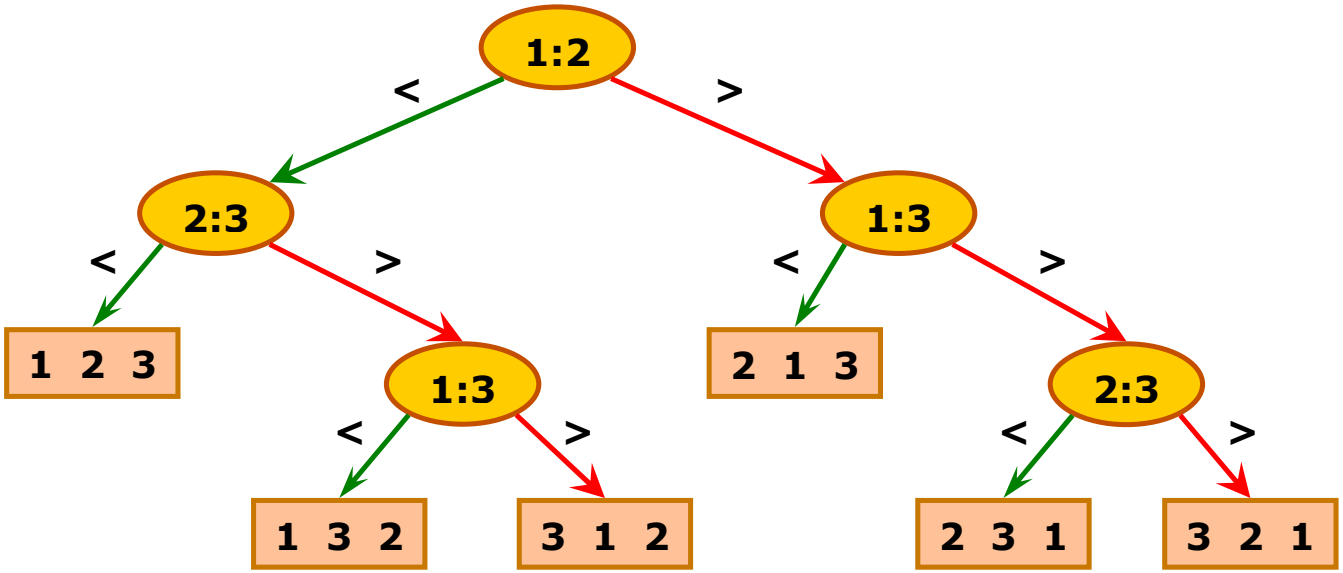
Tri rapide — Variantes : Randomize Quicksort (3)

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

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 borne inférieure

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, Tout algorithme de tri par comparaisons nécessite, dans le cas le plus défavorable, au moins $\lceil n \log n \rceil - n$ (la hauteur de l'arbre) comparaisons pour trier des suites de longueur 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; j < M; j++)
   nombre[j] = 0;
for(i = 0; i < N; i++)
   nombre[C[i]] = nombre[C[i]] + 1;
for(j = 1; j < M; j++)
   nombre[j] = nombre[j-1] + nombre[j];
for(j = N - 1; j > 0; j--) {
   R[nombre[C[i]] = C[i];
   nombre[C[i]] = nombre[C[i]] - 1;
}
```

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

Tri par base (Radix Sort)

Ce tri se base sur la représentation binaire (interne) des clés 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.

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

Tri par base (Radix Sort)

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

C'est une méthode pour trier un jeu de carte.

Tri par base (Radix Sort)

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

C'est une méthode pour trier un jeu de carte.

Tri par base (Radix Sort)

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

C'est une méthode pour trier un jeu de carte.

Tri par base (Radix Sort)

(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.

Tri par paquets (Bucket sort)

Pour 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 (ou paquets) de même 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 à l'aide d'un tri auxilaire (Quicksort ou insertion sort).
- On rassemble les données en parcourant les classes dans l'ordre.

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

Tri par paquets (Bucket sort)

Exemple

- N est compris entre 1 000 000 et 5 000 000 d'éléments ;
- $M = 231 = 2147483648$;
- $K = 32768$ (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 Quicksort !

Sélection

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 premier (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.

Sélection (2)

```
void SELECT(int T[], int i, int j, int k) {
    int p;
    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 $O(n)$.

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

Sélection (3)

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 $m_1, m_2, \ldots, m_{n/5}$.

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 !!!

Complexité des algorithmes — Exercices — TD 3

Un autre algorithme de tri, tri par tas

AbdelKamel BEN ALI

Centre Universitaire d'El-Oued

Mai 2011

Master 1 d'informatique – SDIA