Algorithmes de tri

I.S.I.A.

COURS 4

Franck.Nielsen@sophia.inria.fr

Année 1995-1996

Introduction

→On peut utiliser la recherche dichotomique sur une table d'éléments triés.
→Nombreux algorithmes optimaux pour des modéles de machines variants.
On distingue:

- Le **tri interne**: toutes les clefs à trier sont disponibles en mémoire centrale.
- Le **tri externe**: On utilise de la mémoire secondaire pour stocker les éléments: va et vient des données entre la mémoire centrale et la mémoire secondaire.

→on examine dans la suite quelques **tris internes**.

→Les tris permettent de mettre en relief un certain nombre de paradigmes. On les classe ainsi par les techniques algorithmiques qui mettent en jeu plutôt que leur complexité.

Tris&Paradigmes

Tri par échange. On échange les éléments qui ne sont pas dans le bon ordre. →tri à bulles et tri rapide (ou de Hoare).

Tri par sélection. On sélectionne un élément particulier et on le sépare du reste de la liste. On recommence jusqu'à obtenir la liste vide.

Tri par insertion. On insére un par un les éléments en les plaçant correctement. →tri par insertion dichotomique.

Tri par comptage. chaque élément est comparé aux autres et on compte le nombre d'éléments plus petits que lui. On calcule ainsi son rang.

Tri par fusion. Met en œuvre la méthode "diviser-pour-régner" très en vogue en algorithmique.

Tri par compartiment. Éléments directement ordonnés grâce à un calcul sur leur clef.

→Borne inférieure de $\Omega(n \log n)$.

Position du problème

Notons $\mathcal{E} = \{e_1, ..., e_n\}$ un ensemble de n éléments à trier. À chaque élément e_i est associé une clef unique c_i d'un ensemble totalement ordonné de clefs $(\mathcal{C}, <)$. On prend $\mathcal{C} = [n]$.
On cherche une permutation σ de Σ_n telle que:

$$i \leq j \Leftrightarrow c_{\sigma(i)} \leq c_{\sigma(j)}$$

$\rightarrow$Les éléments (clefs) sont stockés dans un tableau T de taille n: $c_i = T[i]$.
Chaque permutation requiert 3 affectations. Opération de base: comparaison de 2 clefs et permutation.
Si on utilise uniquement le tableau T alors on qualifie le tri de **tri en place**. Si les éléments identiques se retrouvent classés à la fin du tri suivant leur ordre relatif dans T, on parle de **tri stable**.

Tri par échanges: Tri à bulles

→tri en place stable. →L'idée est de faire descendre les éléments les plus petits tandis que les éléments les plus grands remontent (idée de bulle – bubble sort).
→passes successives dans T. Chaque passe consiste en un parcours séquentiel. Durant ce parcours, on permutte les éléments consécutifs non-ordonnés. Le tableau est trié si lors d'une passe on effectue aucune permutation.

```
procedure TriBulle(var T:tableau);
var i:integer;fin:boolean;
begin
fin := false;
while not fin do
                begin
                fin := true;
                for i := 1 to n - 1 do
                if T[i + 1] < T[i] then
                                        begin
                                        Permuter(T, i, i + 1);
                                        fin := false;
                                        end;

                end;
end;
```

Complexité du tri à bulles

Exercice 1 *Illustrer le déroulement de la procédure TriBulle sur le tableau d'entiers suivant:*

$$2, 5, 8, 9, 1, 2$$

Donner un exemple où le tri à bulle demande $n-1$ passes successives. Quel est le prix d'une passe?

$\rightarrow$On peut modifier l'algorihtme pour la k^{e} passe en notant que seuls les éléments de $T[1..n-k+1]$ interviennent lors du k^{e} parcours.
Dans le cas le pire, on effecute $\Sigma_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$ comparaisons. Sur un tableau trié, on a 0 permutation à effectuer mais $n-1$ comparaisons.

Exercice 2 *Que pensez-vous de l'analyse en moyenne de cet algorithme?*

Tri par échange – Tri rapide (Hoare)

→très efficace en pratique (QuickSort). On partage le tableau en deux sous-tableaux par rapport à un élément e choisi au hasard de manière à ce que le premier tableau soit constitué des éléments inférieurs à e tandis que le second contient les éléments supérieurs à e. e est appelé **pivot**.
→appels récursifs sur les deux sous-tableaux.
→fonction partitionnant le tableau autour du pivot. Tri en place.

procedure TriRapide(**var** T:tableau;i, j:**integer**);
var k:**integer**;
begin
if $i < j$ **then**
begin
$k := Partition(T, i, j)$;
TriRapide$(T, i, k-1)$;
TriRapide$(T, k+1, j)$;
end;
end;

Tri rapide en place

La fonction **Partition** est décrite ci-dessous:

```
function Partition(T:tableau;i, j:integer):integer;
var k,l:integer;
begin
l := i + 1;
k := j;
while l ≤ k do
                begin
                while T[k] > T[i] do k := k − 1;
                while l ≤ j and T[l] ≤ T[i] do l := l + 1;
                if l < k then
                        begin
                        Permuter(T, l, k);
                        l := l + 1;
                        k := k − 1;
                        end
                end
Permuter(T, i, k);
Partition := k;
end;
```


Complexité du tri rapide

Exercice 3 *Illustrer le déroulement de la procédure TriRapide sur le tableau d'entiers suivant:*

$$2, 5, 8, 9, 1, 2, 6, 7$$

Donner le schéma des appels récursifs emboîtés.

→Dans le cas le pire, le tri rapide a une complexité en $O(n^2)$. Dans un meilleur cas ou en moyenne, l'algorithme a une complexité en $O(n \log n)$.

Exercice 4 *Prouvez la complexité moyenne de l'algorithme. Améliorer l'algorithme pour que dans les cas meilleurs, il ne requiert que $O(n)$ opérations.*

Tri par sélection

On cherche un élément extrêmal (par ex. le plus petit élément). On le permute avec la première place du tableau puis on itére avec le sous-tableau $T' = T[2..n]$ et ainsi de suite.

```
procedure TriSelection(var T:tableau;debut:integer);
var i,k:integer;
begin
if debut < n then
                begin
                i := debut;
                for k := debut to n do
                                if T[k] < T[i] then i := k;
                Permuter(T, debut, i);
                TriSelection(T, debut + 1);
                end;
end;
```

La complexité en moyenne, dans le meilleur des cas, dans le cas le pire est en $O(n^2)$.

Tri par insertion

→Tri par insertion dichotomique. On suppose que les $i-1$ premiers éléments sont triés et on cherche la place du i^{e} élément dans les $i-1$ premiers. On l'insére à la bonne place et on continue jusqu'au n^{e} élément.
Plusieurs variantes dépendant de la localisation:

- tri par insertion séquentielle.
- tri par insertion dichotomique.

```
procedure TriInsertionDicho(var T:tableau;i:integer);
var j,k,x:integer;
begin
if i > 1 then
        begin
        TriInsertionDicho(T, i − 1);
        if T[i − 1] > T[i] then
                        begin
                        k := PlaceInsertion(T, 1, i − 1, T[i]);
                        x := T[i];
                        for j := i − 1 downto k do
                                                T[j + 1] := T[j];
                        T[k] := x;
                        end;
        end;
end;
```


Complexité du tri par insertion dichotomique

Exercice 5 *Donner l'implantation de la procédure* **PlaceInsertion**. *Faites dérouler à la main l'algorithme sur le tableau:*

$$3, 6, 9, 2, 4, 1, 8, 4$$

Comment améliorer l'algorithme si on utilise de la mémoire supplémentaire (tri non stable)?

Complexité dans le meilleur des cas linéaire. Complexité dans le cas le pire quadratique. Complexité en moyenne en $O(n \log n)$.

Exercice 6 *Prouvez les résultats sur la complexité du tri par insertion dichotomique.*

Tri par comptage

On calcule pour chaque élément son rang: le nombre d'éléments inférieurs à lui. Le tri est non en place. De plus il pose des difficultés lorsque les éléments ne sont plus distincts.
On utilise toujours un espace mémoire linéaire en effectuant toujours $\frac{n(n-1)}{2}$ comparaisons.

```
procedure TriComptage(var T,C:tableau);
var i,j:integer; Ttrie:tableau;
begin
for i := 1 to n do C[i] := 1;
for i := 1 to n - 1 do
    for j := i + 1 to n do
    if T[i] ≥ T[j] then
                    C[i] := C[i] + 1;
                    else C[j] := C[j] + 1;
for i := 1 to n do Ttrie[C[i]] := T[i];
for i := 1 to n do T[i] := Ttrie[i];
end;
```

→peu utilisé en pratique.

Tri par fusion

Méthode basée sur le principe *"diviser pour régner"*: on décompose un problème en sous-problèmes puis on résoud récursivement ces problèmes. Enfin, on déduit la solution du problème originel à l'aide des solutions aux sous-problèmes.
$\rightarrow$On divise le tableau de taille n en deux sous-tableaux T_1 et T_2 de taille $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$. On trie récursivement T_1 et T_2 et on **fusionne** les deux sous-tableaux triés.

```
procedure TriFusion(var T:tableau;i, j:integer);
var k:integer;
begin
if i < j then
        begin
        k := (i + j) div 2;
        TriFusion(T, i, k);
        TriFusion(T, k + 1, j);
        Fusion(T, i, k, j);
        end;
end;
```

Algorithme de Fusion

```
procedure Fusion(var T:tableau;i,k,j:integer);
var m,l,z,r:integer;Q:tableau;
begin
m := i;
l := k + 1;
z := i;
while m ≤ k and l ≤ j do
      begin
      if T[m] ≤ T[l] then
                  begin
                  Q[z] := T[m];
                  z := z + 1;
                  m := m + 1;
                  end
                  else begin
                  Q[z] := T[l];
                  z := z + 1;
                  l := l + 1;
                  end;
      end;
if m > k  then
          for r := l to j do
                    kbegin
                    Q[z] := T[r];
                    z := z + 1;
                    end
          else for r := m to k do
                        begin
                        Q[z] := T[r];
                        z := z + 1;
                        end;
for z := 1 to n do T[z] := Q[z];
end;
```


Complexité du tri fusion

$\rightarrow$Trier deux listes triées requiert un temps linéaire.
La complexité en moyenne, dans le meilleur des cas, dans le cas le pire est la même. Soit $c(n)$ la complexité pour trier n éléments. On a:

$$c(n) = \begin{cases} 0 & \text{is } n = 1 \\ c(\lfloor \frac{n}{2} \rfloor) + c(\lceil \frac{n}{2} \rceil) + A \times n & \text{sinon.} \end{cases}$$

où A est le coefficient de la procédure de fusion.
$\rightarrow$On montre que l'équation récursive de la complexité est en $O(n \log_2 n)$ ($c(n) \leq An \log_2 n$).

Exercice 7 *Améliorer cet algorithme de façon à ce que le tri d'une liste triée se fasse en temps linéaire. Dérouler l'algorithme sur l'échantillon:*

$$1, 7, 3, 6, 3, 9, 5, 2$$

Donner l'arbre des appels récursifs. Prouver la complexité $c(n)$ à l'aide de l'arbre des appels récursifs.

Tri par sauts

Du terme anglais **bucket sort** ou tri par compartiment.
$\rightarrow$on considère des entiers de $[n] = \{1, ..., n\}$.
On range chaque élément en fonction de sa clef. Si les éléments varient dans $[m]$ alors on crée un tableau S initialisé à 0 et on range l'élément $e \in [n]$ dans $T[e]$. On parcout alors le tableau S de 1 à m et on en déduit la liste triée de $\mathcal{E}$.

$\rightarrow$Tri a une complexité en $O(n+m)$. On choisit $m = O(n)$ de façon à obtenir un tri linéaire. Notez qu'il y a seulement m valeurs possibles pour $e \in \mathcal{E}$.

$\rightarrow$on chaine les éléments en collision.

$\rightarrow$Tri valide que pour des types énumérés ordonnables.

Bucket sort(suite)

```
procedure BucketSort(var T:tableau);
var i, j:integer;U:tableau liste;p:liste;
begin
for i := 1 to m do U[i] := nil;
for i := 1 to n do Ajouter(T[i], U[T[i]]);
j := 1;
for i : +1 to m do
    begin
    if U[i] ≠ nil  then begin
                    p := U[i];
                    while p ≠ nil  do begin
                                    T[j] := p.val;
                                    j := j + 1;
                                    p := p ↑ suivant;
                    end;
                    end;
    end;
end;
```

→Complexité linéaire.

→Algorithme adaptable pour le tri lexicographique d dimensionel en $O(dn)$

Borne inférieure pour le tri

→Modéle RAM.
→nécessite $\Omega(n \log n)$ comparaisons.
→modéle de fonctionnement d'un algorithme de tri: **l'arbre de décision**.
On fait tourner génériquement un algorithme et on branche suivant le résultat d'une comparaison.
Une feuille est une permutation trouvée grâce au chemin de la racine jusqu'à cette feuille. Il y a $n!$ permutations possibles.
Puisque l'arbre est strictement binaire, on sait que la hauteur minimale d'un tel arbre est $\log_2(n!) = O(n \log n)$ en utilisant la formule de Stirling ($n! \simeq n^{n+\frac{1}{2}}\sqrt{2\pi} \exp -n$).
La borne inférieure sur le modéle de comparaison est donc:

$$\boxed{\Omega(n \log_2 n)}$$