Algorithmique et Complexité

Algorithmique et Complexité

4. Stratégie I : Diviser pour Régner et Tri
4.2 Algorithmes de tri : Tri rapide, Tri fusion, ...

Nicole Bidoit

Université Paris XI, Orsay

Année Universitaire 2008–2009

Quelques généralités concernant les algorithmes de tri

Le tri (sorting) est omniprésent dans les cours d'informatique !

en programmation, en algorithmique ... **pourquoi ?**

Le tri est LA tâche la plus fréquente effectuée par les ordinateurs

tri = préliminaire à la résolution efficace de beaucoup de problèmes

une fois que les objets sont triés, ces problèmes deviennent "simples"

Exemple phare :

Recherche d'un élément dans un tableau trié :

$\hookrightarrow$ recherche dichotomique complexité = $O(\lg n)$

Les ordinateurs font majoritairement du tri parce qu'ils font majoritairement de la recherche

Le traitement de données à large échelle serait impossible sans le tri!

Autre exemple : Trouver parmi n nombres, les paires les plus proches.

Si les n nombres sont triés, les nombres formant ces paires sont l'un à coté de l'autre.

Autre exemple : Est-ce que n objets sont distincts ?

Trier les n nombres, il suffit ensuite de parcourir la liste triée!

Base de données : suppression des dupliqués après une projection

Autre exemple : Calculer la fréquence d'occurrence d'un élément (de tous les éléments) parmi n.

en exercice.

Généralités concernant les algorithmes de tri

Autre exemple : Le $k$ème plus grand élément ...

il existe un algorithme linéaire ; idée = tri partiel

Autre exemple : Enveloppe convexe

Étant donné n points dans le plan, trouver le plus petit polygone qui contient tous les éléments

géométrie algorithmique : la base d'autres algos sophistiqués.

...

Le tri est un des problèmes le plus / le mieux étudiés en informatique

Toutes les "stratégies", techniques algorithmiques et d'analyse peuvent être illustrées avec le tri :
diviser pour régner, algorithme randonisé, bornes inférieures ...

La librairie du langage de programmation C offre une fonction `qsort` pour trier ...
comme la plupart des langages de programmation.

Ordre des éléments – Fonctions de comparaison
Les librairies fournissent les règles pour les caractères (ponctuation, etc ...), ...

Caractéristiques :

Quelles structures de données ? Coût : qu'est-ce qu'on compte ?

stabilité ?

Les classes de méthodes de tri

Tri par sélection

1. Recherche (sélection) du plus petit élément
2. Placement du plus petit élément en tête

`Tri par tas` (Heapsort)

`Tri à bulles` (Bubble sort)

Tri par insertion

Hypothèse : à l'étape i, la portion $[1..i-1]$ du tableau /liste est triée

L'étape i place le $i$ème élément de sorte que la portion $[1..i]$ du tableau (liste) soit triée

liste versus tableau , insertion dichotomique versus séquentielle

Tri par partition

1. On construit des portions du tableau ayant de bonnes propriétés
2. On combine les portions du tableau en utilisnt ces propriétés.

`Tri rapide` (Quicksort)

`Tri par fusion` (Mergesort)

`Tri par hachage`

Rappel du principe – stratégie ”par partition” et diviser pour régner

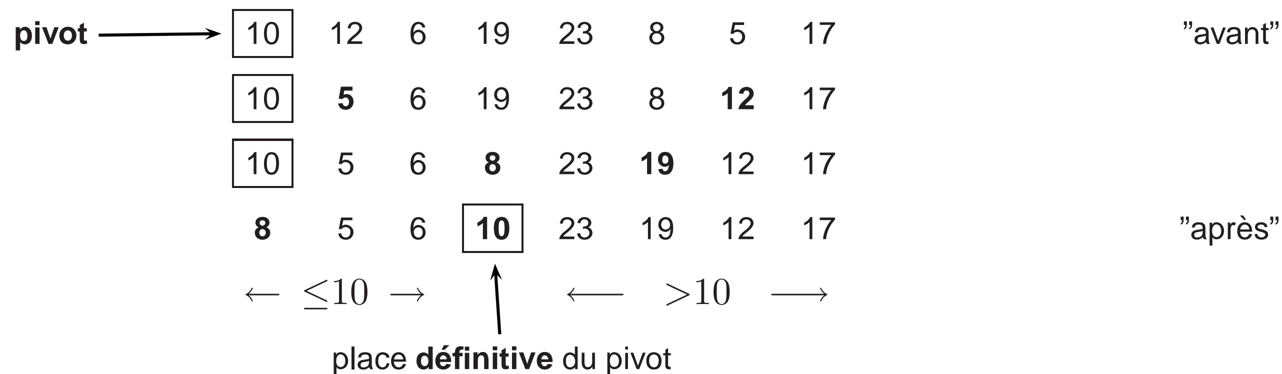
Entrée : tableau T stockant n éléments muni d'un ordre total

1. Couper le tableau $T[1..n]$ en 2 **en utilisant un élément pivot** $T[k]$

   $\hookrightarrow$ choix du pivot ?

   tous les e dans $T[1..m]$ sont $\leq$ au pivot et tous les e dans $T[m+1..n]$ sont $>$ au pivot
2. Trier chacune des 2 sous instances avec la même méthode
3. Tri(T) est la concaténation des deux sous-instances triées.

Exemple : une étape de partitionnement du tri rapide

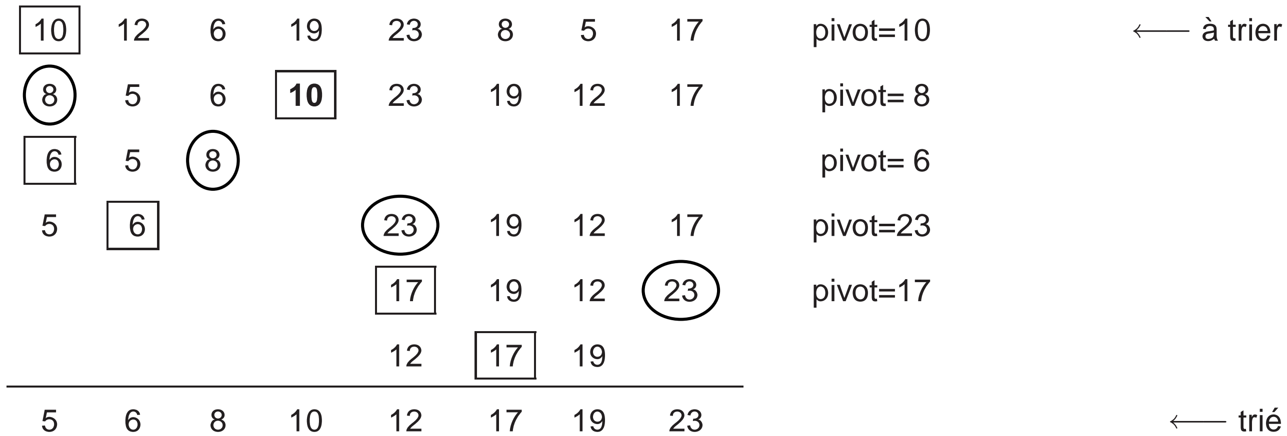


Partitionnement relatif à un pivot

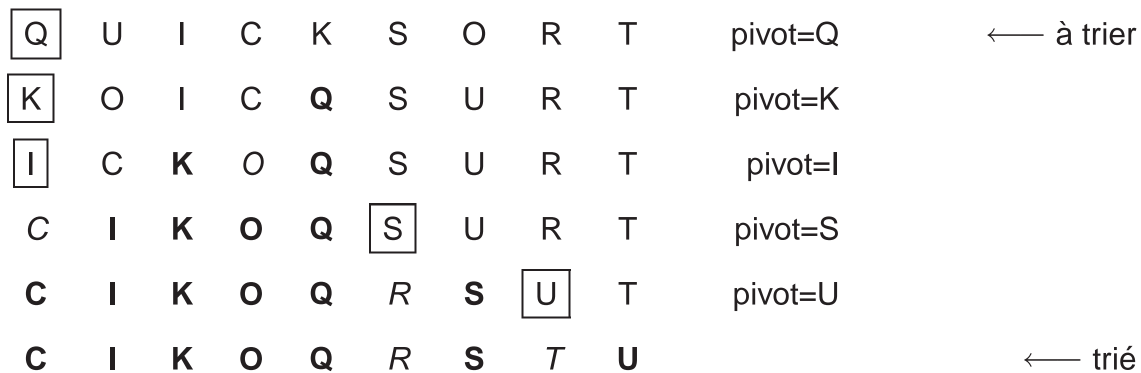
un parcours (de gauche à droite) du tableau

maintien de 3 zones : $\leq pivot$ (début), $>$ pivot (fin), non exploré (entre deux)

Exemple : les partitionnements successifs (tri complet)



Exemple : un autre



```
Algo Partition
```

entrée : $T[1..n]$ of elt; inf,sup integer; % tableau à partitionner entre inf et sup

sortie : $T[1..n]$ of elt; place_pivot : integer; % tableau partitionné et placement du pivot

auxiliaire : gauche, droit : integer; pivot: elt

$pivot \leftarrow T[inf]$;

$gauche \leftarrow inf{+}1$;

$droit \leftarrow sup$;

while $gauche \leq droit$ **do**

  while $T[gauche] \leq pivot$ **do** $gauche \leftarrow gauche{+}1$;

  while $T[droit] >$ pivot **do** $droit \leftarrow droit{-}1$;

  if $gauche < droit$ **then**

    $T[droit] \leftrightarrow T[gauche]$;

    $gauche \leftarrow gauche{+}1$; $droit \leftarrow droit{-}1$

  endif

endwhile

$T[inf] \leftrightarrow T[droit]$;

$place_pivot \leftarrow droit$;

return(place_pivot);

```
Algo Tri-rapide
```

entrée : $T[1..n]$ of elt; inf,sup integer % tableau à trier entre inf et sup

sortie : $T[1..n]$ of elt ; % T trié entre inf et sup

auxiliaire : k integer ;

if $inf < sup$ **then**

place_pivot ← Partition(T,inf,sup);

Tri-rapide(T,inf,place_pivot-1);

Tri-rapide(T,place_pivot$+1$,sup);

endif

Exercice : Ecrire une version du tri rapide pour laquelle le pivot est le dernier élément de la portion du tableau à partitionner. Reprendre les exemples précédents pour concrétiser ce qui se passe dans la phase de partitionnement.

Exercice : Montrer que l'algorithme de tri rapide est correct

Exercice : Le tri rapide est-il stable ? Sinon expliquez comment modifier l'algorithme pour le rendre stable.

Analyse du partitionnement

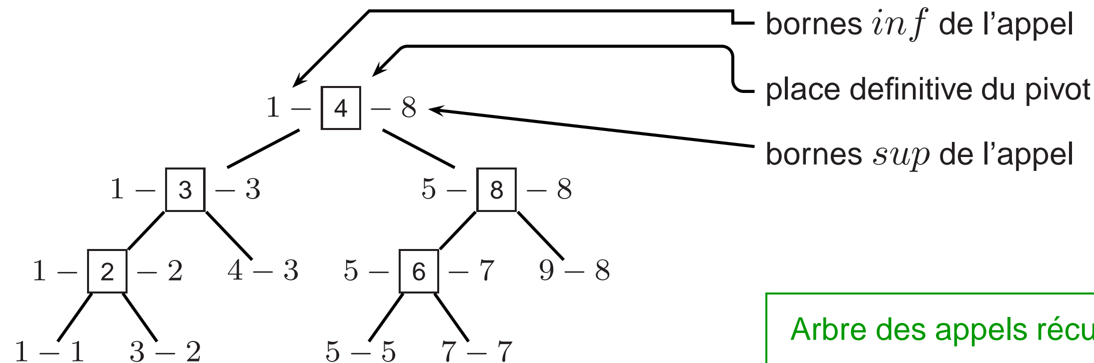
Que compte-t-on ? comparaisons et échanges

Analyse des **cas limite** et des **cas d'arrêt** du partitionnement

Nombre de comparaisons dans $\Theta(n)$ entre $n-1$ et $n+1$

Nombre d'échanges dans $\Theta(n)$ au pire $\lceil \frac{n}{2} \rceil$

Analyse du tri rapide



Arbre des appels récursifs pour Tri-Rapide(T,1,8)

avec le tableau T de l'exemple numérique

Analyse du tri rapide

déterminer le nombre de niveaux de l'arbre des appels

au niveau k, l'algo "traite" p sous-tableaux dont la taille "cumulée" vaut $n-k$ (k pivots placés)

$\hookrightarrow n-k+1$ comparaisons au pire $\quad \hookrightarrow \lceil \frac{n-k}{2} \rceil$ échanges au pire.

Cas le meilleur

le nombre de niveaux de l'arbre des appels est minimum

le pivot est médiant et il est placé au milieu du sous-tableau à chaque fois

chaque sous-instance est de taille $\frac{n}{2^k}$

... et l'arbre a $\lg n$ niveaux $\boxed{O(n \lg n)}$

Cas le pire

le nombre de niveaux de l'arbre des appels est maximum

le pivot est un plus petit (plus grand) élément, il est placé au début (fin)

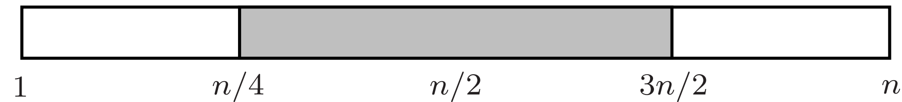
et coupe l'instance de taille t en une instance de taille 1 et une instance de taille $t-1$

... et l'arbre a $n-1$ niveaux $\boxed{O(n^2)}$

Analyse du tri rapide

(Intuition) pour le cas en moyenne

Hypothèse : pas de répétition d'éléments, la place définitive du pivot est équiprobable



La moitié du temps, le choix du pivot place celui-ci dans le "centre" du tableau.

Dans ces cas là, la taille maximale des deux sous-instances est $\frac{3n}{4}$!

En acceptant que le pivot soit toujours placé finalement dans cette zone centrale, de combien de niveaux d'appels

récursifs (partitions) avons-nous besoin pour descendre jusqu'à des partitions de taille 1 ?

$(\frac{3}{4})^f.n = 1 \Rightarrow n = (\frac{4}{3})^f \Rightarrow \lg n = f. \lg \frac{4}{3}$

donc $f = \frac{\lg n}{\lg \frac{4}{3}} \leq 2 \lg n$

et les mauvaises partitions ? c'est l'autre moitié des cas

Plus rigoureusement, on montre que la complexité en moyenne est dans $\boxed{\mathbf{O(n \lg n)}}$

(peut-être fait en cours ou en TD, mais cette analyse est complexe)

Choisir un bon pivot

Le cas le pire (le tableau est déjà trié ou presque) est vraiment pénalisant

Cas de certaines applications

$\hookrightarrow$ Choisir pour pivot l'élément du milieu ou

$\hookrightarrow$ Choisir pour pivot le médian des éléments premier, milieu et dernier, ou

$\hookrightarrow$ Choisir un pivot aléatoirement $\longleftarrow$ bonne solution

Le cas le pire est toujours possible mais ”peu” probable (en pratique)

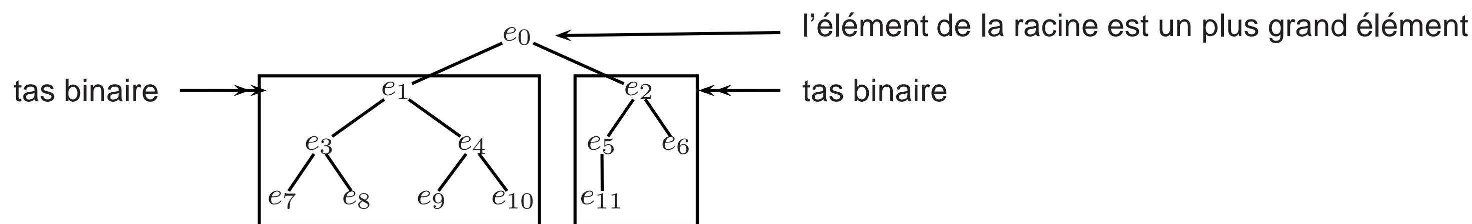
Faire mieux que le tri rapide ... en théorie

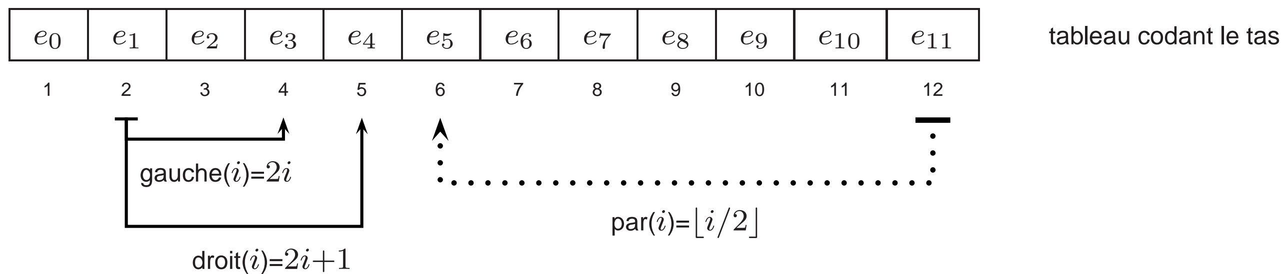
Tri par sélection : recherche plus petit élément placé au début

exploiter la recherche du + petit elt pour structurer le tableau

Tas binaire : c'est un arbre binaire très "dense" et "régulier" codé dans un tableau

$\hookrightarrow$ maintenir un **ordre partiel** sur les éléments du tableau

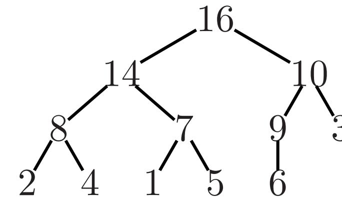




Propriété d'un tas : $T[par(i)] \geq T[i]$ pour $i > 1$ $\quad$ $T[1]$ est un plus grand elt.

Exemple de tas

1	2	3	4	5	6	7	8	9	10	11	12
16	14	10	8	7	9	3	2	4	1	5	6



Exemple : tri à partir d'un tas

échange (1 ↔ 12)

1	2	3	4	5	6	7	8	9	10	11	12	
6	14	10	8	7	9	3	2	4	1	5	16	refaire un tas avec $T[1...n-1]$ sachant que $gauche(1)$ et $droit(1)$ sont des tas
14	6	10	8	7	9	3	2	4	1	5	16	échange de $T[1]$ et $T[gauche(1)]$
14	8	10	6	7	9	3	2	4	1	5	16	échange de $T[2]$ et $T[gauche(2)]$

5	8	10	6	7	9	3	2	4	1	14	16	échange de $T[1]$ et $T[n-1]$
10	8	5	6	7	9	3	2	4	1	14	16	échange de 5 et 10
10	8	5	6	7	9	3	2	4	1	14	16	échange de 5 et 9

...

Tri par Tas :

Construire un tas à partir du tableau d'éléments initial

Effectuer le tri du tas en itérant la suite d'étapes :

Placer le plus grand élément à la fin de la portion de tableau

Refaire un tas avec la portion du tableau diminuée du dernier élément

Construire un tas : méthode simple

Hypothèse : la portion $T[1..i]$ est un tas

Insérer $T[i{+}1]$ dans le tas $T[1..i]$ pour obtenir un tas $T[1..i{+}1]$

Echanger le père $T[par(i{+}1)]$ et $T[i{+}1]$ si $T[par(i{+}1)] < T[i{+}1]$

Complexité : on compte les comparaisons et les échanges

il y a $n{-}1$ insertions pour construire le tas

chaque insertion consiste à remonter dans le tas d'une feuille jusqu'à la racine (au pire)

en effectuant à chaque pas de remontée une comparaison et un échange.

nbre comparaisons et nbre échanges sont donc liés à la hauteur h du tas.

$\sum_{i=0..h} 2^i = 2^{h+1} - 1 \geq n$ et donc $h = \lfloor \lg n \rfloor$ $\boxed{\Theta(n \lg n)}$

$\hookrightarrow$ **mieux** : fusion de deux tas et d'un nouvel élément (exemple)

```
Algo Heapify                                              % (Robert Floyd)
```

entrée : $T[1..n]$ of elt; i, sup integer; % (voir commentaire ci-dessous)

sortie : $T[1..n]$ of elt; % l'arbre en i est un tas (voir commentaire)

auxiliaire : $gauche$, $droit$, $topi$: integer

$gauche \leftarrow gauche(i)$, $droit \leftarrow droit(i)$;

if $gauche \leq sup$ **then if** $T[gauche] > T[i]$ **then**

$topi \leftarrow gauche$ **else** $topi \leftarrow i$;

if $droit \leq sup$ **then if** $T[droit] > T[topi]$ **then** $topi \leftarrow droit$;

if $topi \neq i$ **then** { $T[i] \leftrightarrow T[topi]$; Heapify(T, $topi$, sup) }

Entrée :

Les ”arbres” en $gauche(i)$ et $droit(i)$ sont des tas.

La zone du tableau examiné est comprise entre les indices i et sup.

Sortie :

L'arbre binaire de racine i dont le codage dans le tableau est compris (mais n'occupe pas forcément tout) entre les indices i et inf est un tas.

Complexité : on compte les comparaisons et les échanges

même raisonnement que précédemment $\boxed{O(\lg n)}$

```
Algo Cons-Tas
```

entrée :$T[1..n]$ of elt;

sortie : $T[1..n]$ of elt; % $T[1..n]$ est un tas

auxiliaire : i integer

for $i = \lfloor n/2 \rfloor$ **downto** 1 **do** Heapify(T, i, n);

Complexité de la construction du tas de départ:

chaque appel Heapify(T, i, n) coûte $O(h)$ comparaisons et échanges si le noeud en i est à hauteur h

dans un tas de taille n, à hauteur h, il y a au plus $\lceil n/2^{h+1} \rceil$ noeuds.

$$\sum_{h=0..\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0..\lfloor \lg n \rfloor} \frac{h}{2^h}) \qquad \sum_{h=0..\lfloor \lg n \rfloor} \frac{h}{2^h} \leq \sum_{h=0..\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$$

donc le nombre de comparaisons et d'échanges est majoré par $2n$ soit en $\boxed{O(n)}$

Tri par tas – Heapsort

`Algo Tri-Tas`

entrée : $T[1..n]$ of elt;

sortie : $T[1..n]$ of elt; % tableau trié

auxiliaire : i, fin_tas: integer

Cons-Tas(T); $fin_tas \leftarrow n$;

for $i = n$ **downto** 1 **do** $\{$ $T[1] \leftrightarrow T[i]$; $fin_tas \leftarrow fin_tas - 1$; Heapify($T$, 1, fin_tas); $\}$

Complexité :

la construction est en $O(n)$

le tri à partir du tas est en $O(n \lg n)$

$\boxed{O(n \lg n)}$

Résultat important (preuve éventuellement en cours ou en TDs)

1. Dans le cas le pire, il faut $\lceil \lg(n!) \rceil \in O(n \lg n)$ comparaisons pour trier n éléments.
2. En moyenne, il faut $\lceil \lg(n!) \rceil \in O(n \lg n)$ comparaisons pour trier n éléments.

   $\hookrightarrow$ le tri rapide est optimal, en moyenne mais pas dans le cas le pire

   $\hookrightarrow$ le tri par tas est optimal en moyenne et dans le cas le pire.

$\hookrightarrow$ **Ne pas espérer trouver un algorithme de tri en $O(n)$! comparaisons**

Il existe des algorithmes qui utilisent d'autres opérations que de la comparaison

Tri par tas – Heapsort

Est-ce que le tri par tas est réellement meilleur que le tri rapide ?

Rappel : le tri rapide est en $O(n^2)$ dans le pire cas

le tri par tas est en $O(n \lg n)$ dans le pire cas

Le tri rapide (implémentation rigoureuse) est 2 à 3 fois plus performant

opérations plus simples dans la boucle interne

Quelques exercices

Exercice : La représentation d'un arbre "tas binaire" par un tableau peut-elle se généraliser à tout type d'arbre binaire, quelconque ?

Exercice : Le tri par tas est-il stable ?

Exercice : Décrire un algorithme en $O(k)$ (comparaisons) dans le pire des cas qui permet de déterminer si le $k$ième plus grand élément du tas $T[1..n]$ est strictement plus petit que r. Aide : il n'est pas nécessaire de trouver le $k$ième plus grand élément.