Tri des tableaux

universite
PARIS-SACLAY

Ressources

Beaucoup d'illustrations en ligne

`https://visualgo.net/en/sorting`

Usage des structures triées

- calculer la médiane, les quantiles d'un ensemble de valeurs
- recherche dichotomique
- éliminer les valeurs en double dans un ensemble
- etc.

Préliminaires

- dans cette partie, on considère des tableaux de taille fixe :
  - la taille $t.n$ du tableau t est notée n dans les algos.
- ordre sur les éléments du tableau donné par l'opérateur $\leq$
- objectif : les valeurs de t doivent être telles que

$$t[i] \leq t[i+1], \forall i \in \{0 \ldots n-2\}$$

Algorithmes de tri

- tris par comparaison
  - tri par insertion
  - tri par sélection
  - tri à bulle
  - tri fusion
  - tri rapide
  - tri par tas
- tris qui reposent sur des hypothèses sur la distribution des valeurs
  - tri par dénombrement
  - tri par base

Algorithmes de tri : autres classifications

- tri en place : ne nécessite pas de mémoire supplémentaire
- tri stable : préserve l'ordre relatif des élements égaux
- complexité dans le pire cas
- complexité en moyenne
- complexité quand le tableau est presque trié

Aucun algorithme de tri n'est optimal pour tous ces critères.

Algorithmes de tri : autres classifications

- tri en place : ne nécessite pas de mémoire supplémentaire
- tri stable : préserve l'ordre relatif des élements égaux
- complexité dans le pire cas
- complexité en moyenne
- complexité quand le tableau est presque trié

Aucun algorithme de tri n'est optimal pour tous ces critères.

Tri par insertion version 1

principe : on parcourt le tableau en insérant les valeurs à leur place dans un deuxième tableau.

Exemple avec le tableau

$$8, 4, 3, 5, 7, 2, 1, 6$$

Tri par insertion

A l'étape i, le tableau est divisé en 2 parties :

- la partie des indices de 0 à i est triée
- la partie des indices $i+1$ à $n-1$ n'est pas triée

tri par insertion : algorithme

Algorithme : tri_insertion(t : tableau de n entiers)

Entrées : t tableau de n entiers

Résultat : tableau t trié

```
pour i de 1 à n − 1
    j ← i − 1;
    tant que j ≥ 0 et t[j] > t[j + 1]
        t[j] ↔ t[j + 1];
        j ← j − 1 ;
```

Cet algorithme est en place.

Complexité : $O(n^2)$.

Tri à bulle : description

principe : une étape consiste à parcourir le tableau en partant de la fin et à permuter les voisins qui ne sont pas dans le bon ordre. On répète cela jusqu'à ce que le tableau soit trié.

A la fin de l'étape i, le tableau est en 2 parties :

- la partie des indices de 0 à $i-1$ est triée
- tous les éléments de la première partie sont inférieurs à ceux de la deuxième partie
- donc le nombre d'étapes maximum est n

Exemple avec le tableau

$$8, 4, 3, 5, 7, 2, 1, 6$$

tri à bulle : algorithme

Algorithme : tr_bulle(t : tableau de n entiers)

Entrées : t tableau de n entiers

Résultat : tableau t trié

Variables locales : i, j : indices du tableau (entiers)

```
pour i de 1 à n − 1
    pour j de n − 1 à i en décroissant
        si t[j] < t[j − 1]
            t[j] ↔ t[j − 1];
```

Complexité : $O(n^2)$.

Complexité minimale pour un tri par comparaison

question : pour un algorithme de tri par comparaisons, combien de comparaisons sont nécessaires au minimum dans le pire cas ?

Les bornes inférieures sont dures à prouver en général, et nécessitent un modèle précis.

théorème

Tout tri par comparaison requiert au moins $\alpha n \log(n)$ comparaisons, où α est une constante.

Arbre de décision d'un tri par comparaison

Le fonctionnement d'un algorithme de tri par comparaison sur toutes les entrées de taille n peut être décrit par un arbre de décision :

- les noeuds internes sont les indices du tableau que l'on compare
- les arêtes gauche correspondent à l'ordre $<$ et l'arête droite à l'ordre $>$ sur les valeurs de ces indices
- les feuilles correspondent à un ordre sur les n éléments
- le nombre de comparaisons pour un ordre est la distance de la feuille correspondante à la racine de l'arbre

Exemple pour le tri par insertion au tableau avec $n = 3$.

Borne inférieure de complexité

théorème

Tout tri par comparaison requiert au moins $\alpha n \log(n)$ comparaisons, où α est une constante.

preuve :

- le nombre d'ordres, donc de feuilles de l'arbre est $n!$
- un arbre binaire de hauteur h a au plus 2^h feuilles, ce qui est atteint par l'arbre complet de hauteur h
- la hauteur d'un arbre binaire complet est le logarithme de sa taille
- comme $n! \leq n^n$, on obtient

$$\log(n!) \in O(n \log(n))$$

Tri fusion

idée : diviser le tableau en 2, trier chaque partie séparément, puis les fusionner.

Tri du tableau suivant :

$$8, 4, 3, 5, 7, 2, 1, 6$$

- tri par insertion : 21 comparaisons
- tri par insertion sur chaque moitié, puis fusion des tableaux triés : 17 comparaisons

Amélioration : utiliser le tri fusion pour trier chaque partie : récursivité.

Tri fusion

idée : diviser le tableau en 2, trier chaque partie séparément, puis les fusionner.

Tri du tableau suivant :

$$8, 4, 3, 5, 7, 2, 1, 6$$

- tri par insertion : 21 comparaisons
- tri par insertion sur chaque moitié, puis fusion des tableaux triés : 17 comparaisons

Amélioration : utiliser le tri fusion pour trier chaque partie : récursivité.

Tri fusion

idée : diviser le tableau en 2, trier chaque partie séparément, puis les fusionner.

Tri du tableau suivant :

$$8, 4, 3, 5, 7, 2, 1, 6$$

- tri par insertion : 21 comparaisons
- tri par insertion sur chaque moitié, puis fusion des tableaux triés : 17 comparaisons

Amélioration : utiliser le tri fusion pour trier chaque partie : récursivité.

tri fusion : algorithme

Algorithme : tri_fusion(t : tableau de n entiers, g, d : indices)

Entrées : t tableau de n entiers, g et d indices

Résultat : tableau t trié entre les indices g et d inclus

si $g < d$

  $m \leftarrow \frac{g+d}{2}$;

  $tri\_fusion(t, g, m)$;

  $tri\_fusion(t, m+1, d)$;

  $fusion(t, g, m, d)$;

L'algorithme de fusion de 2 tableaux de taille n_1 et n_2 se fait en au plus $n_1 + n_2 - 1$ comparaisons (fait en TD).

Cet algorithme nécessite un tableau auxiliaire, il n'est pas en place.

tri fusion : complexité

Soit $c(n)$ le nombre de comparaisons du tri fusion dans le pire cas pour un tableau de taille n.

$$\left\{\begin{array}{lcl} c(n) & = & 2c(n/2)+(n-1) \\ c(1) & = & 0 \end{array}\right.$$

Par le Master théorème on obtient

$$c(n) \in 0(n\log(n))$$

Master theorème (version faible)

Soit une fonction de complexité définie de la manière suivante :

$$c(n) = ac(n/b) + f(n)$$

où f est une fonction entière positive.

- si $f(n) \in O(n^{\log_b a-\epsilon})$ avec $\epsilon > 0$, alors

$$c(n) \in O(n^{\log_b a})$$

- sinon si $f(n) \in O(n^{\log_b a})$, alors

$$c(n) \in O(n^{\log_b a} \log n)$$

- sinon si $af(n/b) \leq kf(n)$ avec $k < 1$ alors

$$c(n) \in O(f(n))$$

méthode “diviser pour régner“

3 étapes :

- **diviser** un problème en plusieurs sous-problèmes
- **résoudre** les sous-problèmes
- **combiner** les solutions des sous-problèmes pour construire la solution du problème entier

Tri rapide : principe

- **diviser** : on divise le tableau $t[g..d]$ en 2 sous-tableaux $t[g..p-1]$ et $t[p+1..d]$ où chaque élément de $t[g..p-1]$ est inférieur à $t[p]$ et chaque élément de $t[p+1..d]$ est supérieur à $t[p]$ (partitionnement).
- **résoudre** : on trie chaque sous-tableau
- **combiner** : rien à faire

Exemple : $4, 5, 8, 1, 3, 7, 2, 6$

tri rapide : algorithme

Algorithme : tri_rapide(t : tableau de n entiers, g, d : indices)

Entrées : t tableau de n entiers

g et d sont les indices entre lesquels on trie le tableau

Résultat : tableau t trié

si $g < d$
    $p \leftarrow partition(t, g, d)$;
    $tri\_rapide(t, g, p-1)$;
    $tri\_rapide(t, p+1, d)$;

tri rapide : algorithme de partition

Algorithme : partition(t : tableau de n entiers, g, d : indices) : indice

Entrées : t : tableau, g et d sont les indices entre lesquels on partitionne le tableau

Sorties : indice du pivot après partitionnement

$j \leftarrow g$
pour i *de* g *à* $d-1$
    si $T[i] < T[d]$
        $T[i] \leftrightarrow T[j]$
        $j \leftarrow j+1$
$T[d] \leftrightarrow T[j]$
retourner j

La valeur du pivot choisie est $T[d]$.

Complexité : $g-d$ comparaisons.

tri rapide : complexité

- l'algorithme de partitionnement de n éléments a la complexité $O(n)$
- la complexité du tri dépend de la place du pivot dans le tableau partitionné :
  - s'il est plutôt au milieu : aussi rapide que le tri fusion
  - s'il est plutôt sur les bords : aussi lent que le tri par insertion

tri rapide : pire cas

- partition en 2 sous-tableaux de taille 0 et $n-1$ à chaque étape
- soit $c(n)$ le nombre de comparaisons dans le pire cas

$$c(n) = n + c(n-1) + c(0)$$

et $c(0) = 0$

$$\begin{aligned} c(n) &= c(n-1) + (n-1) \\ &= c(n-2) + (n-2) + (n-1) \\ &= \textstyle\sum_{i=1}^{n-1} i \\ &= O(n^2) \end{aligned}$$

- pire cas : tableau trié
- alors que dans ce cas le tri par insertion est linéaire

tri rapide : meilleur cas

- partition en 2 sous-tableaux de taille $n/2$ à chaque étape (le pivot choisi est la médiane du tableau)
- soit $c(n)$ le nombre de comparaisons dans le meilleur cas

$$c(n) = 2c(n/2) + (n-1)$$

et $c(1) = 0$

Donc $c(n) = O(n \log n)$ (comme le tri fusion).

tri rapide : analyse en moyenne

- tous les ordres possibles sont équiprobables
- la position du pivot est équiprobable
- équation à résoudre :

$$c(n) = \sum_{i=0}^{n-1} \frac{1}{n} \left(c(i) + c(n-i-1) + (n-1)\right)$$

tris par comparaison en résumé

algorithme	**Pire cas**	**En moyenne**
tri par insertion	$O(n^2)$	$O(n^2)$
tri à bulle	$O(n^2)$	$O(n^2)$
tri fusion	$O(n \log n)$	$O(n \log n)$
tri rapide	$O(n^2)$	$O(n \log n)$