Master Sciences, Technologies, Santé
Mention Mathématiques - spécialité Enseignement des mathématiques

Algorithmique et graphes, thèmes du second degré

Algorithmes de tri

Éric SOPENA, eric.sopena@labri.fr

Plan de la présentation

- Introduction
- Tris simples
- Le tri rapide (Quicksort)
- Le tri-fusion

Première partie

Introduction

Complexité(s) d'un algorithme

La complexité d'un algorithme est une mesure permettant d'évaluer la ***performance*** de cet algorithme en fonction de la taille des données du problème à traiter. On distingue :

- la ***complexité en temps***, qui mesure le temps nécessaire à l'exécution d'un algorithme,
- la ***complexité en espace***, qui mesure la taille mémoire nécessaire à l'exécution d'un algorithme.

L'analyse de la complexité consiste à déterminer ces deux grandeurs, dans le but de ***comparer*** des algorithmes résolvant le ***même problème***.

Complexité(s) d'un algorithme

Notons *coût(d)* la complexité (en temps ou en espace) d'un algorithme sur la donnée *d*, et *Dn* l'ensemble des données de taille *n*. On s'intéresse alors aux quantités suivantes :

- la complexité dans le meilleur des cas :
  $Min(n) = min \{ coût(d) / d \in Dn \}$
- la complexité dans le pire des cas :
  $Max(n) = max \{ coût(d) / d \in Dn \}$
- la complexité en moyenne :
  $Moy(n) = \sum_{d \in Dn} coût(d) \times P(d)$,
  $P(d)$: probabilité d'avoir la donnée d en entrée.

Expression de la complexité

On introduit les notations suivantes, qui permettent de comparer les ***ordres de grandeur asymptotiques*** des fonctions :

- $f = O(g)$ (f est dominée par g) si et seulement si il existe une constante C et un rang $n0$, t.q.
  $n > n0 \Rightarrow f(n) \leq C \times g(n)$
- $f = \Theta(g)$ (f et g sont asymptotiquement équivalentes) si et seulement si
  $f = O(g)$ et $g = O(f)$

Ainsi, par exemple :

$3n + 7 = \Theta(n)$ $\quad 5n^2 + 6n - 11 = \Theta(n^2)$

Complexités usuelles

- $\Theta(1)$: complexité constante
  (calcul d'une formule)
- $\Theta(\log n)$: complexité logarithmique
  (recherche dichotomique)
- $\Theta(n)$: complexité linéaire
  (recherche séquentielle)
- $\Theta(n \log n)$: complexité subquadratique
  (tri optimal)
- $\Theta(n^2)$: complexité quadratique
  (tri simple)
- $\Theta(n^k)$: complexité polynomiale
- $\Theta(2^n)$: complexité exponentielle
  (tours de Hanoï)

Complexité et efficience

Temps d'exécution (1 opération = 1 µs) en fonction de la complexité et de la taille des données

	1	logn	n	nlogn	n^2	n^3	2^n
10^2	1 µs	6.6 µs	0.1 ms	0.66 ms	10 ms	1 s	4×10^{16} a
10^3	1 µs	9.9 µs	1 ms	9.9 ms	1 s	16.5 mn	∞
10^4	1 µs	13.3 µs	10 ms	0.13 s	1.5 mn	11.5 j	∞
10^5	1 µs	16.6 µs	0.1 s	1.64 s	2.7 h	31.7 a	∞
10^6	1 µs	19.9 µs	1 s	19.9 s	11.5 j	31.7×10^6 a	∞

$\Theta(n^2)$: 10 fois plus de données, 100 fois plus de temps

Algorithmes de tri

Les algorithmes de tri font partie de la culture de base de tout "algorithmicien".
Ils permettent notamment d'illustrer efficacement la notion de complexité.

N objets à trier (tableau, liste, ...)

↓ **Algorithme de tri**

N objets triés

Complexité

en espace : mémoire nécessaire en plus de la donnée

en temps : évolution du temps d'exécution en fonction de N

Algorithmes de tri

On distingue généralement :

Les algorithmes de tri internes (données en mémoire)
Les algorithmes de tri externes (données dans fichier)

Les algorithmes de tri simples
faciles à mettre en œuvre, mais performances médiocres
Tri par sélection - Tri par insertion - Tri bulle
(utilisables si peu de données à trier)

Les algorithmes de tri sophistiqués
plus délicats à mettre en œuvre, meilleures performances
Tri rapide - Tri par tas - Tri fusion

Algorithmes de tri

Le problème du tri de N entiers est un problème de complexité ***N log N*** : un algorithme de complexité N log N est donc un algorithme ***optimal***.

On s'intéressera à deux opérations fondamentales : comparaison et déplacement d'éléments.

Nous illustrerons les algorithmes de tri sur un tableau d'entiers défini ainsi :

```
constante  CMAX = …
type       TTableau = tableau de CMAX entiers

variables  t : TTableau
           n : entier naturel  # nombre d'entiers à trier
```

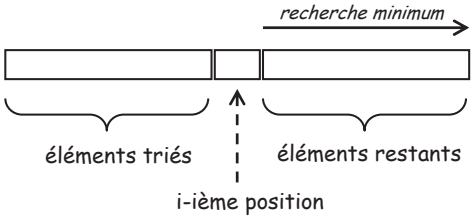

Deuxième partie

Tris simples

Tri par sélection

Idée : on sélectionne (recherche) l'élément devant figurer dans une position donnée...

- 1ère position <---- plus petit
- 2ème position <---- plus petit des restants
- etc.



Tri par sélection

```
Action Tri_Sélection ( ES t : TTableau ; E n : entier )
# tri par sélection
variables  i, pos, posPlusPetit : entiers
début
    Pour pos de 0 à n - 2
    faire
          # plus petit élément entre pos et n-1 ?
        posPlusPetit ← pos
        Pour i de pos + 1 à n - 1
        faire
            Si ( t[i] < t[posPlusPetit] )
            alors posPlusPetit ← i
        fin_pour
            # on range l'élément à sa place
        Echanger (t, pos, posPlusPetit)
    fin_pour
fin
```


Tri par sélection

Complexité

- **déplacements** : chaque élément placé l'est définitivement, ce qui, par échange, correspond à 3(n-1) déplacements, d'où une complexité en $\Theta(n)$.
- **comparaisons** : le nombre de comparaisons est

  $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$,

  d'où une complexité en $\Theta(n^2)$.

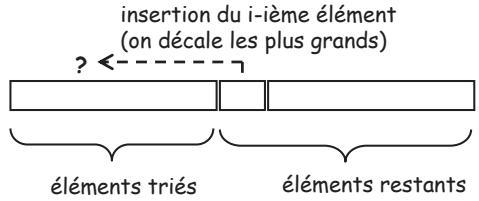
Complexité globale en moyenne : $\Theta(n^2)$

Remarque. Le nombre d'opérations est identique quel que soit l'ordre initial des éléments à trier.

Tri par insertion (joueur de cartes)

Idée : on insère les éléments un à un dans la "portion" déjà triée...

- 1er élément ----> déjà en place
- 2ème élément ----> inséré par rapport au 1er
- etc.



Tri par insertion (joueur de cartes)

```
Action Tri_Insertion ( ES t : TTableau ; E n : entier )
# tri par insertion
variables  pos, i, elem : entiers, trouvé : booléen
début
    pour i de 1 à n - 1
    faire
        elem ← t[i]
        pos ← i - 1, trouvé ← Faux
        tantque ( (non trouvé) et (pos >= 0) )
        faire  si ( t[pos] > elem )
               alors  t[pos+1] ← t[pos]
                      pos ← pos - 1
               sinon  trouvé ← Vrai
               fin_si
        fin_tantque
        t[pos+1] ← elem
    fin_pour
fin
```


Tri par insertion (joueur de cartes)

Complexité

- **déplacements** : $\Theta(n)$ dans le meilleur des cas (déjà trié), $\Theta(n^2)$ dans le pire des cas (trié inverse), $\Theta(n^2)$ en moyenne.
- **comparaisons** : idem.

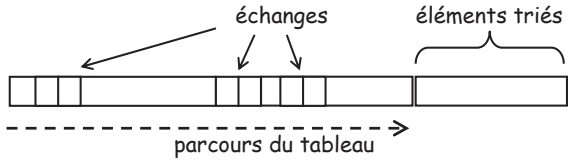
Complexité globale en moyenne : $\Theta(n^2)$

Remarque. On peut accélérer la phase "recherche de la position d'insertion" en procédant par dichotomie. On diminue le nombre de comparaisons, mais le nombre de déplacements (décalages) reste identique.

Tri-bulle (Bubble Sort)

Idée : on fait "remonter" les petits, et "descendre" les grands...

- parcours du tableau en échangeant les éléments consécutifs "mal rangés" (le plus grand est placé)
- on répète ce parcours, jusqu'à effectuer un parcours sans échange...



Tri-bulle (Bubble Sort)

```
Action Tri_Bulle ( ES t : TTableau ; E n : entier )
# tri bulle
variables  i, j : entiers,  fini : booléen
début
   i ← n
   répéter
      fini ← Vrai
         # recherche des consécutifs mal rangés
      pour j de 1 à i
      faire  Si ( t[j] < t[j-1] )
             alors  Echanger (t, j, j-1 )
                    fini ← Faux
             fin_si
      fin_pour
         # le i-ième élément est en place
      i ← i - 1
   Jusqu'à ( fini ou (i = 0) )
fin
```


Tri-bulle (Bubble Sort)

Complexité

- **déplacements :** dans le pire des cas (tableau trié inverse), le nombre d'échanges est de (n-1) + (n-2) + ... + 1 = n(n-1)/2, soit une complexité au pire de $\Theta(n^2)$.
  Dans le meilleur des cas (déjà trié), on n'effectue aucun déplacement (complexité en $\Theta(1)$).
- **comparaisons :** idem pour le pire des cas. Dans le meilleur des cas, on effectue n-1 comparaisons, soit une complexité de $\Theta(n)$ au mieux.

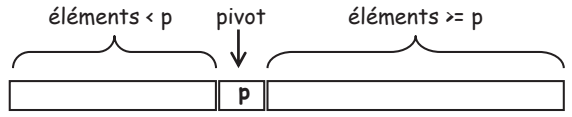
Complexité globale en moyenne : $\Theta(n^2)$

Troisième partie

Le tri rapide (Quicksort)

Tri rapide (Quicksort)

Idée : on va réorganiser le tableau de façon à obtenir une configuration de la forme suivante :



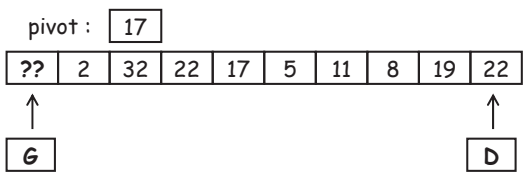
- Le pivot p est à sa place...
- On réorganise ensuite les deux portions de tableau, en utilisant la même méthode (récursivité)

Réorganisation - principe de l'algorithme

Tableau à trier :

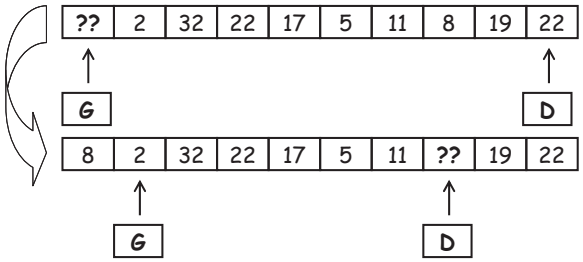
17	2	32	22	17	5	11	8	19	22

Étape 1. On met de côté le premier élément, qui sera notre pivot, ce qui "libère" la case de gauche.



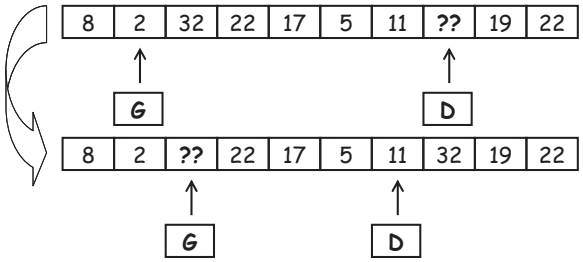
Réorganisation - principe de l'algorithme

Étape 2. En utilisant l'indice D, on cherche à droite un élément (inférieur à pivot) pouvant être rangé à gauche (en position G). On l'y range (G avance), ce qui "libère" une case à droite...



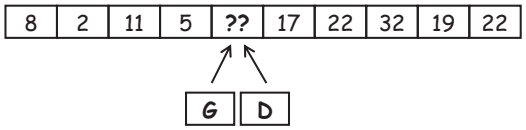
Réorganisation - principe de l'algorithme

Étape 3. En utilisant l'indice G, on cherche à gauche un élément (supérieur ou égal à pivot) pouvant être rangé à droite (en position D). On l'y range (D recule), ce qui "libère" une case à gauche...

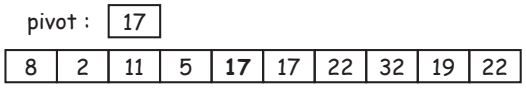


Réorganisation - principe de l'algorithme

Et... on continue jusqu'à ce que les deux indices G et D se rejoignent.

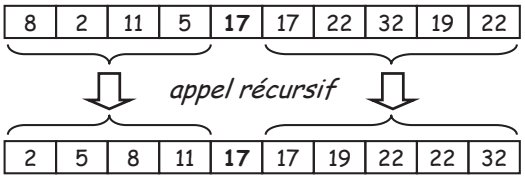


Il ne reste plus qu'à ranger le pivot à sa place...



Tri rapide - récursivité...

En rappelant cette action récursivement sur les deux portions de tableau, celui-ci finira par être trié...



Complexité. La complexité est en moyenne en $\Theta(n \log n)$, donc optimale, mais avec une complexité dans le pire des cas (trié inverse) en $\Theta(n^2)$.

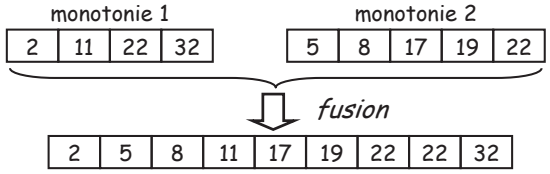
Quatrième partie

Le tri-fusion

Tri-fusion

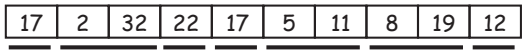
Le tri fusion est la méthode utilisée pour effectuer des tris externes (tri de fichiers). Elle peut facilement s'adapter au cas des tris internes.

Idée : On va détecter les monotonies existantes (suites d'entiers déjà triés), puis les fusionner deux à deux, jusqu'à obtenir une unique monotonie.

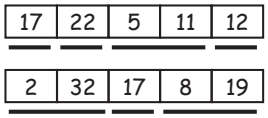


Tri-fusion - principe de l'algorithme

Tableau à trier :

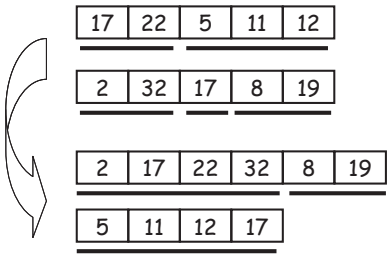


Étape 1. On détecte les monotonies naturelles, on les range alternativement dans deux tableaux



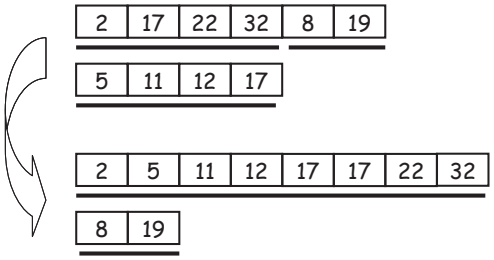
Tri-fusion - principe de l'algorithme

Étape 2. On fusionne (alternativement) dans deux nouveaux tableaux les monotonies "en vis à vis" (avec un peu de chance, certaines monotonies se concatènent...)



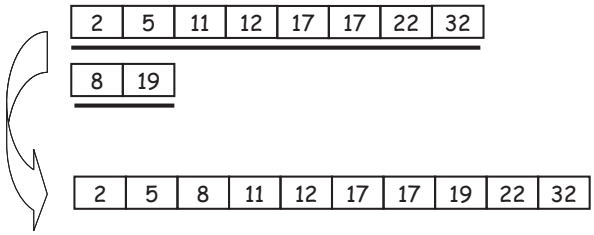
Tri-fusion - principe de l'algorithme

Et... On continue jusqu'à obtenir une unique monotonie...



Tri-fusion - principe de l'algorithme

Et... On continue jusqu'à obtenir une unique monotonie...



Tri-fusion - Complexité

Dans le pire des cas (tableau trié inverse), la 1ère étape construit des monotonies de longueur 1, la 2ème étape des monotonies de longueur 2, la i-ième étape des monotonies de longueur 2^{i-1}.

Le nombre d'étapes est alors log n.

Chaque étape (fusions), est en $\Theta(n)$, d'où une complexité au pire en $\Theta(n \log n)$.

Dans le meilleur des cas (tableau déjà trié), une seule étape suffit, d'où une complexité en $\Theta(n)$.

Tri-fusion - Complexité

La complexité en moyenne est en $\Theta(n \log n)$, cet algorithme est optimal.

Complexité en espace.

Pour trier N éléments, cet algorithme nécessite un espace de 2N cases (chaque élément co-existe dans deux tableaux).

Tous les algorithmes précédents n'utilisaient qu'une case en plus des N éléments à trier (pour réaliser les échanges, ou sauvegarder le pivot).