DUT 2 Informatique – S3
M3103 – Algorithmique avancée

CM 02 – Complexité des algorithmes de tri

Équipe pédagogique : Camille Kurtz, Jacques Alès-Bianchetti, Simon Fuhlhaber

`prénom.nom@parisdescartes.fr`

18 août 2016

USPC Université Sorbonne Paris Cité

UNIVERSITÉ PARIS DESCARTES INSTITUT UNIVERSITAIRE DE TECHNOLOGIE

Plan

1. Rappels sur les algorithmes de tri
2. Algorithmes lents (complexité quadratique) : Tris par sélection et tris par insertion
3. Algorithmes rapides (complexité quasi-linéaire)
4. Comparaison des algorithmes de tri
5. Pour aller plus loin

Plan

1. Rappels sur les algorithmes de tri
2. Algorithmes lents (complexité quadratique) :
   Tris par sélection et tris par insertion
3. Algorithmes rapides (complexité quasi-linéaire)
4. Comparaison des algorithmes de tri
5. Pour aller plus loin

Introduction

Définition

Qu'est ce qu'un algorithme de tri ?

- Il s'agit de **trier de façon croissante ou décroissante un ensemble d'objets** souvent stockés dans une structure de données
- Pour trier (et comparer !) des objets il est obligatoire . . . d'avoir une relation d'ordre
- Il existe **3 familles principales** d'algorithmes de tris : . . .
  séléction, insertion, rapide

A quoi cela peut servir ?

1. A organiser et ranger des données (maintenir triée une structure)
2. . Rechercher dans une structure (arbre binaire)

Il existe de **nombreux algorithmes de tri**. Il est donc nécessaire de les étudier et de **les comparer afin de comprendre dans quels cas les employer**

Complexité des algorithmes

Remarques

- Tous les algorithmes de tri sont basés sur **2 opérations élémentaires**
  1. **Comparaison** de deux éléments ($a \geq b$)
  2. **Affectation** d'un élément dans un autre ($a \leftarrow b$)
- La **complexité en temps** sera indiquée par le nombre de comparaisons et d'affectations réalisées par l'algorithme
- La **complexité en espace** sera indiquée nombre de cases mémoire supplémentaires requises par le déroulement de l'algorithme
- On s'occupera ici de trier **des tableaux** $T[n]$ **de taille** n

$T[n]$						
Non trié	8	5	9	1	2	5

--→

$T[n]$						
Trié	1	2	5	5	8	9

Familles d'algorithmes

Nous étudierons deux familles algorithmiques de tri :

1. lent (insertion, séléction, bulle) : n^2
2. rapide ' quicksort, tri fusion) : n log n

Plan

1. Rappels sur les algorithmes de tri
2. Algorithmes lents (complexité quadratique) : Tris par sélection et tris par insertion
3. Algorithmes rapides (complexité quasi-linéaire)
4. Comparaison des algorithmes de tri
5. Pour aller plus loin

Tris par sélection

Principe des tris par sélection

- **Trouver le plus petit** (resp. grand) élément du tableau et le **mettre en première** (resp. dernière) position
- **Répéter le même algorithme sur la partie du tableau restante** si celle-ci contient plus d'un élément

3 implémentations sont possibles :

1. Tri à bulle
2. Tri par séléction
3. Tri par insertion

Tri à bulles

Algorithm 1 Tri à bulles

procedure $T \hookleftarrow$ TRI_BULLES($T[n]$: tableau de taille n)
  for $i \leftarrow 2$ to n **do**
    for $j \leftarrow 0$ to $n - i$ **do**
      if $T[j+1] \leq T[j]$ **then**
        $T \leftarrow echanger(T[j], T[j+1])$
      end if
    end for
  end for
  return T
end procedure

-> 2 boucles
-> Dépendant de n
-> Complexité n²

Tri à bulles

...

Tri à bulles

Algorithm 2 Tri à bulles

```
procedure T ↩ TRI_BULLES(T[n] : tableau de taille n)
    for i ← 2 to n do
        for j ← 0 to n − i do
            if T[j + 1] ≤ T[j] then
                T ← echanger(T[j], T[j + 1])
            end if
        end for
    end for
    return T
end procedure
```

Remarque

La boucle interne repousse le plus grand élément à la fin de la liste

Déroulement de l'algorithme sur un tableau de taille $n = 6$

. . .

5	17		17		17		17		17		55
4	12		12		12		12		55	j . .	17
3	30		30		30		55	j . .	12		12
2	15		15		55	.j. .	30		30		30
1	40		55	.j. .	15		15		15		15
0	55	j . .	40		40		40		40		40

Tri à bulles

Algorithm 3 Tri à bulles

```
procedure T ↩ TRI_BULLES(T[n] : tableau de taille n)
    for i ← 2 to n do
        for j ← 0 to n − i do
            if T[j + 1] ≤ T[j] then
                T ← echanger(T[j], T[j + 1])
            end if
        end for
    end for
    return T
end procedure
```

Déroulement de l'algorithme sur un tableau de taille $n = 6$

. . .

5	55		55		55		55		55
4	17		17		17		17		40
3	12		12		12		40	. j .	17
2	30		30		40	. j .	12		12
1	15		40	.j. .	30		30		30
0	40	j . .	15		15		15		15

Tri à bulles

Algorithm 4 Tri à bulles

```
procedure T ↩ TRI_BULLES(T[n] : tableau de taille n)
    for i ← 2 to n do
        for j ← 0 to n − i do
            if T[j + 1] ≤ T[j] then
                T ← echanger(T[j], T[j + 1])
            end if
        end for
    end for
    return T
end procedure
```

Déroulement de l'algorithme sur un tableau de taille $n = 6$

...

5	55		55		55		55
4	40		40		40		40
3	17		17		17		30
2	12		12		30	. j .	17
1	30		30	j . .	12		12
0	15	j . .	15		15		15

Tri à bulles

Algorithm 5 Tri à bulles

```
procedure T ↩ TRI_BULLES(T[n] : tableau de taille n)
    for i ← 2 to n do
        for j ← 0 to n − i do
            if T[j + 1] ≤ T[j] then
                T ← echanger(T[j], T[j + 1])
            end if
        end for
    end for
    return T
end procedure
```

Déroulement de l'algorithme sur un tableau de taille $n = 6$

...

5	55		55		55
4	40		40		40
3	30		30		30
2	17		17		17
1	12		15	j . .	15
0	15	j . .	12		12

Tri à bulles

Algorithm 6 Tri à bulles

```
procedure T ↩ TRI_BULLES(T[n] : tableau de taille n)
    for i ← 2 to n do
        for j ← 0 to n − i do
            if T[j + 1] ≤ T[j] then
                T ← echanger(T[j], T[j + 1])
            end if
        end for
    end for
    return T
end procedure
```

Déroulement de l'algorithme sur un tableau de taille $n = 6$
. . .

5	55		55		55	T est trié
4	40		40		40	
3	30		30	$\Rightarrow$	30	
2	17		17		17	
1	15		15		15	
0	12	j. . .	12		12	

Tri à bulles

Complexité de l'algorithme de tri à bulles pour différents cas

	nb. de comparaisons	nb. de transferts	mémoire
Meilleur des cas	$n(n-1)/2 \Rightarrow \Theta(n^2)$	$0 \Rightarrow \Theta(0)$	$1 \Rightarrow \Theta(1)$
Cas moyen	$n(n-1)/2 \Rightarrow \Theta(n^2)$	$3n(n-1)/4 \Rightarrow \Theta(n^2)$	$1 \Rightarrow \Theta(1)$
Pire des cas	$n(n-1)/2 \Rightarrow \Theta(n^2)$	$3n(n-1)/2 \Rightarrow \Theta(n^2)$	$1 \Rightarrow \Theta(1)$

Avantages

- Facile à comprendre et à implémenter

Inconvénients

- Un des algorithmes de tri les moins efficaces

Optimisations et variantes possibles

- Changer alternativement l'ordre de balayage – « tri cocktail »
- Détecter s'il reste des éléments à trier en fonction du balayage précédent (Meilleur des cas $\Theta(n)$)

Tri à bulles optimisé

Optimisation étudiée

Détecter s'il reste des éléments à trier **en fonction du balayage précédent**

Algorithm 7 Tri à bulles optimisé

```
procedure T ↫ TRI_BULLES_OPTIM(T[n] : tableau de taille n)
    do
        swapped ← false
        for i ← 1 to n − 1 do
            if T[i − 1] > T[i] then
                T ← echanger(T[i − 1], T[i])
                swapped ← true
            end if
        end for
    while swapped
    return T
end procedure
```

Meilleur des cas Theta(n)

Tri par sélection directe

Algorithm 8 Tri par sélection directe

```
procedure T ↫ TRI_SELECTION(T[n] : tableau de taille n)
    for i ← 0 to n − 2 do
        min ← i                                  ▷ Premier candidat possible
        for j ← i + 1 to n − 1 do
            if T[j] < T[min] then
                min ← j                          ▷ Meilleur candidat
            end if
        end for
        T ← echanger(T[i], T[min])
    end for
    return T
end procedure
```

Tri par sélection directe

Tri par sélection directe

Algorithm 9 Tri par sélection directe

```
procedure T ↫ TRI_SELECTION(T[n] : tableau de taille n)
    for i ← 0 to n − 2 do
        min ← i                                  ▷ ...
        for j ← i + 1 to n − 1 do
            if T[j] < T[min] then
                min ← j                          ▷ ...
            end if
        end for
        T ← echanger(T[i], T[min])
    end for
    return T
end procedure
```

Remarque

.La boucle interne ramène le plus petit élément au début

Déroulement de l'algorithme sur un tableau de taille $n = 6$

...

5	17		17		17		17		17	.j. .	17	
4	12		12		12		12	.j. .	12	. m	55	.j. .
3	30		30		30	j. .	30		30		30	
2	15		15	.j .	15	.m	15	m .	15		15	
1	40	j. . .	40	m.	40		40		40		40	
0	55	.m.	55		55		55		55		12	m

Tri par sélection directe

Algorithm 10 Tri par sélection directe

```
procedure T ↫ TRI_SELECTION(T[n] : tableau de taille n)
    for i ← 0 to n − 2 do
        min ← i                                  ▷ ...
        for j ← i + 1 to n − 1 do
            if T[j] < T[min] then
                min ← j                          ▷ ...
            end if
        end for
        T ← echanger(T[i], T[min])
    end for
    return T
end procedure
```

...

$i = 1$

5	17			17	
4	55			55	
3	30			30	
2	15	.j. .	...	40	m.
1	40	.m.		15	.i. .
0	12			12	

$i = 2$

5	17			40	.m
4	55			55	
3	30	j. .		30	
2	40	m.	...	17	.i. .
1	15			15	
0	12			12	

Tri par sélection directe

Algorithm 11 Tri par sélection directe

```
procedure T ↩ TRI_SELECTION(T[n] : tableau de taille n)
    for i ← 0 to n − 2 do
        min ← i                                   ▷ ...
        for j ← i + 1 to n − 1 do
            if T[j] < T[min] then
                min ← j                           ▷ ...
            end if
        end for
        T ← echanger(T[i], T[min])
    end for
    return T
end procedure
```

...

$i = 3$

5	40			40	
4	55	j...		55	
3	30	m..	...	30	i,m
2	17			17	
1	15			15	
0	12			12	

$i = 4$

5	40	j..		55	m..
4	55	m.		40	i..
3	30		...	30	
2	17			17	
1	15			15	
0	12			12	

Rappels Algorithmes lents Algorithmes rapides Comparaison des algorithmes Pour aller plus loin

Tri par sélection directe

Complexité de l'algorithme de tri par sélection directe

	nb. de comparaisons	nb. de transferts	mémoire
Tous les cas	$n(n-1)/2 \Rightarrow \Theta(n^2)$	$3(n-1) \Rightarrow \Theta(n)$	$1 \Rightarrow \Theta(1)$

Avantages

- Facile à comprendre et à implémenter
- Faible nombre de transfert

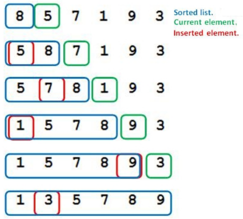
Inconvénients

- Algo lent
- Quadratique au meilleur des cas

Tri par insertion

Principe du tri par insertion

- Considérer la première case **comme un sous-tableau déjà trié** et le reste **comme les éléments devant être insérés** un à un dans ce sous-tableau trié
- L'insertion d'un nouvel élément dans le sous-tableau se fait en **3 étapes** :
  1. Recherche de l'indice où insérer
  2. Décalage de tous les éléments se trouvant après cet indice
  3. Insertion proprement dîte



Remarque

Du fait que le sous-tableau est trié, la recherche de la position où insérer peut se faire par dichotomie

Tri par insertion

Algorithm 12 Tri par insertion

```
procedure T ↩ TRI_INSERTION(T[n] : tableau de taille n)
    for i ← 1 to n − 1 do
        if T[i] < T[i − 1] then
            tmp ← T[i]                              ▷ Il n'est pas à sa place
            j ← recherche(T, 0, i − 1, tmp)
            T ← supprimer(T, i)
            T ← insérer(T, tmp, j)
        end if
    end for
    return T
end procedure
```

Note :
La fonction *recherche*() renvoie la position que devrait avoir *tmp* dans le début du tableau T (compris entre les positions 0 et $i-1$).

Tri par insertion

Tri par insertion

Algorithm 13 Tri par insertion

```
procedure T ↩ TRI_INSERTION(T[n] : tableau de taille n)
    for i ← 1 to n − 1 do
        if T[i] < T[i − 1] then
            tmp ← T[i]                              ▷ Il n'est pas à sa place
            j ← recherche(T, 0, i − 1, tmp).
            T ← supprimer(T, i)
            T ← insérer(T, tmp, j)
        end if
    end for
    return T
end procedure
```

Déroulement de l'algorithme sur un tableau de taille $n = 6$ **(test pour** $i = 3$**)**

17
12
30
...
55
40
15

... ...

Tri par insertion

Complexité de l'algorithme de tri par insertion pour différents cas

	nb. de comparaisons	nb. de transferts	mémoire
Meilleur des cas	$n-1 \Rightarrow \Theta(n)$	$1 \Rightarrow \Theta(1)$	$1 \Rightarrow \Theta(1)$
Cas moyen	$n^2/4 \Rightarrow \Theta(n^2)$	$n^2/4 \Rightarrow \Theta(n^2)$	$1 \Rightarrow \Theta(1)$
Pire des cas	$n^2/2 \Rightarrow \Theta(n^2)$	$n^2/2 \Rightarrow \Theta(n^2)$	$1 \Rightarrow \Theta(1)$

Avantages

- La complexité reste linéaire ... si le tableau est presque trié

Inconvénients

- Le pire cas est atteint lorsque le tableau ... est trié à l'envers

Optimisations et variantes possibles

- En utilisant une recherche par **dichotomie** pour trouver l'emplacement où insérer l'élément, on peut ne faire que $\Theta(n \log n)$ comparaisons. Le nombre d'affectations reste en $\Theta(n^2)$

Plan

1. Rappels sur les algorithmes de tri
2. Algorithmes lents (complexité quadratique) : Tris par sélection et tris par insertion
3. Algorithmes rapides (complexité quasi-linéaire)
4. Comparaison des algorithmes de tri
5. Pour aller plus loin

Notions préliminaires

Rappel

Une **construction est récursive** si elle se définit à partir d'elle-même

Exemple : Dessin de la vache qui rit

Autre exemple : le triangle de Sierpinski

Notions préliminaires

Algorithmes récursifs

- En algorithmique, une fonction qui s'invoque elle-même est dite **récursive**
- Dans les algorithmes récursifs, **2 points sont importants** :
  - la condition d'arrêt : . . . contoler la terminaison de l'algo
  - l'appel récursif : . . . appelle de la fonction à elle même

Remarques

- Les algorithmes récursifs sont souvent employés dans le cadre de problèmes « Diviser pour régner » ou pour le traitement et le parcours des structures de données arborescentes
- On oppose généralement **les algorithmes récursifs aux algorithmes dits impératifs ou itératifs** qui s'exécutent sans invoquer ou appeler explicitement l'algorithme lui-même

Exemple avec le calcul de $n!$

Exemple avec le calcul de $n!$

La **factorielle** se définit intuitivement pour des entiers positifs de la façon suivante :

$n! = \ldots$ Produit de i pour i allant de 1 à n

L'idée de la récursivité est d'utiliser une définition équivalente, à savoir une suite récurrente :

$n! = \ldots$

1 (si n = 0 ou 1)

n(n-1)! (sinon)

Traduction en Java :

```java
int factorielle(int n) {
    if (n <= 1)  return 1; //Condition d'arret
    else  return  n * factorielle(n - 1); //Appel recursif
}

public static void main(String [] args) {
    //Appel initial de la fonction recursive
    int f = factorielle(5);
}
```

Exemple avec le calcul de la suite de Fibonacci

Exemple avec le calcul de la suite de Fibonacci

La **suite de Fibonacci** est une suite d'entiers dans laquelle **chaque terme est la somme des deux termes qui le précèdent**. Ses premiers termes sont :

$$\mathcal{F}^1 = 1, \mathcal{F}^2 = 1, \mathcal{F}^3 = 2, \mathcal{F}^4 = 3, \mathcal{F}^5 = 5, \mathcal{F}^6 = 8, \mathcal{F}^7 = 13, \mathcal{F}^8 = 21, \textit{etc}$$

Voici sa formule récurrente :

$\mathcal{F}^n = \dots$ 1 Si n=1 ou n=2

F_{n+2} F_{n+1} + F_n

Traduction en Java :

```
int fiboRec(int n) {
    if (n<=2)  return 1; //Condition d'arret
    else  return fiboRec(n-1) + fiboRec(n-2); //Appel recursif
}

public static void main(String [] args) {
    //Appel initial de la fonction recursive
    int fib = fiboRec(5);
}
...
```


Notions préliminaires

Attention

- Exécuter trop d'appels de fonction fera **déborder la pile d'exécution** !
- Il est souvent possible d'écrire un même algorithme en itératif et en récursif
- L'exécution d'une version récursive d'un algorithme est généralement **un peu moins rapide** que celle de la version itérative, même si le nombre d'instructions est le même (à cause de **la gestion des appels de fonction**)
- Sur des structures de données **naturellement récursives**, il est bien plus facile d'écrire des algorithmes récursifs qu'itératifs
- Certains algorithmes sont extrêmement difficiles à écrire en itératif

```
public static void testPile(int nbAppels){
    System.out.println("appel numero " + nbAppels);
    testPile(nbAppels + 1);
}
testPile(1);

---------CONSOLE---------
appel numero 1
appel numero 2
...
appel numero 5613
Exception in thread "main" java.lang.StackOverflowError
at sun.nio.cs.SingleByte$Encoder.encodeArrayLoop(SingleByte.java:187)
```

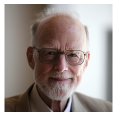
Tri rapide (Quicksort)

Histoire

- Le **tri rapide** (anglais : Quicksort) est créé en **1960 par Tony Hoare**, alors étudiant en visite à l'Université d'État de Moscou, lors d'une étude sur la traduction automatique pour le National Physical Laboratory
- Il a alors besoin de l'algorithme pour **trier les mots devant être traduits**, afin de les faire correspondre à un dictionnaire Russe-Anglais déjà existant, stocké sur une bande magnétique

Propriétés

- Comme son nom l'indique le tri rapide est . . . rapide avec une complexité temporelle en Theta(n)
- Son calcul repose sur l'utilisation d'un algo récursif

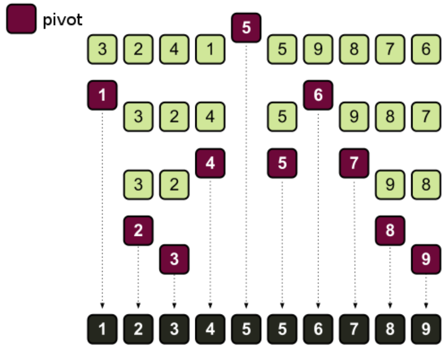


Sir Antony Richard Hoare
(naissance : 1934)

Tri rapide (Quicksort)

Principe du tri rapide

1. **Choisir une valeur de la liste en tant que pivot**
2. **Réorganiser les valeurs de la liste autour du pivot** de façon à ce que toutes les valeurs strictement plus petites soient à gauche et les strictement plus grandes à droite
3. **Recommencer le même algorithme sur la sous-liste** regroupant les valeurs plus petites (s'il contient plus d'un élément) et sur la sous-liste regroupant les valeurs plus grandes (même condition)



Tri rapide (Quicksort)

Principe du tri rapide

1. **Choisir une valeur de la liste en tant que pivot**
2. **Réorganiser les valeurs de la liste autour du pivot** de façon à ce que toutes les valeurs strictement plus petites soient à gauche et les strictement plus grandes à droite
3. **Recommencer le même algorithme sur la sous-liste** regroupant les valeurs plus petites (s'il contient plus d'un élément) et sur la sous-liste regroupant les valeurs plus grandes (même condition)

Remarques

1. Le choix de la valeur du pivot peut être quelconque mais le mieux est de choisir une valeur médiane parmi celles présentes dans la liste
2. **Plusieurs algorithmes permettant de réorganiser la liste existent**. Les plus efficaces la partagent en 3 zones :
   - valeur < pivot
   - valeur = pivot
   - valeur > pivot

Tri rapide

Algorithm 14 Répartition

procedure $(T,\ p_{new}) \hookleftarrow$ REPARTITION($T[n]$, entiers premier, dernier, p)
    $echanger(T[p], T[dernier])$ $\triangleright$ le pivot prend la place du dernier
    $j \leftarrow premier$
    for $i \leftarrow premier$ to $dernier - 1$ **do**
        if $T[i] \leq T[dernier]$ **then** $\triangleright$ il n'est pas à sa place
            $echanger(T[i], T[j])$
            $j \leftarrow j + 1$
        end if
    end for
    $echanger(T[dernier], T[j])$
    return (T, j)
end procedure

Algorithm 15 Tri rapide récursif

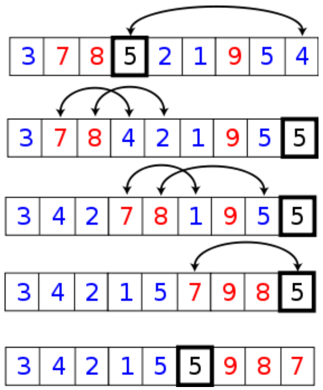
procedure $T \hookleftarrow$ TRI_RAPIDE_REC($T[n]$, entier premier, entier dernier)
    if $premier < dernier$ **then**
        $p \leftarrow CHOIX\_PIVOT(T, premier, dernier)$ $\triangleright$ $p =$ indice pivot
        $(T, p_{new}) \leftarrow REPARTITION(T, premier, dernier, p)$
        $T \leftarrow TRI\_RAPIDE\_REC(T, premier, p_{new} - 1)$
        $T \leftarrow TRI\_RAPIDE\_REC(T, p_{new} + 1, dernier)$
    end if
    return T
end procedure

Tri rapide

Algorithm 16 Appel principal du tri rapide

```
procedure T ↩ TRI_RAPIDE(T[n])
    return TRI_RAPIDE_REC(T, 0, n − 1)
end procedure
```

Exemple de partitionnement sur un tableau de taille $n = 9$



Tri rapide

Complexité de l'algorithme de tri rapide pour différents cas

	nb. de comparaisons	nb. de transferts	mémoire
Meilleur des cas	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(\log_2 n)$
Cas moyen	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(\log_2 n)$
Pire des cas	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Avantages

- ... Très bonne complexité

Inconvénients

- L'algorithme repose sur un double appel récursif qui peut ne pas être efficace pour des tableaux de grande taille (pile des appels)
- ... Mauvaise complexité au pire des cas

Optimisations et variantes possibles

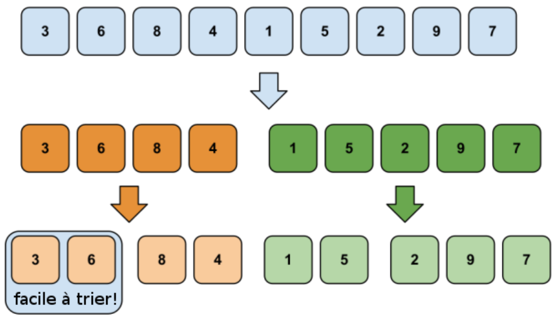
- **Différentes stratégies sont possibles pour le choix du pivot** : pivot aléatoire, pivot optimal (valeur médiane du sous-tableau). Elles peuvent permettre de diminuer la complexité dans certains cas.

Tri par fusion

Principe du tri par fusion

Le tri par fusion est basé sur la technique ... divide and conquer :

- **On divise** le tableau à trier en 2 parts égales
- **On trie chacun de ces deux tableaux**
- Finalement, **on fusionne** ces deux tableaux triés



Tri par fusion

Principe du tri par fusion

Le tri par fusion est basé sur la technique ... divide and conquer :

- **On divise** le tableau à trier en 2 parts égales
- **On trie chacun de ces deux tableaux**
- Finalement, **on fusionne** ces deux tableaux triés

Implémentation

- On utilise 2 fonctions *fusion*() et *tri_fusion*() :
  1. La fonction *fusion*() réalise **la fusion entre deux tableaux** $T_1[n_1]$ et $T_2[n_2]$ et place le résultat dans un tableau $T[n]$ $(n_1 + n_2 = n)$
  2. La fonction *tri_fusion*() est **l'algorithme de tri** qui peut se définir récursivement via un double appel récursif

Tri par fusion

Algorithm 17 Fusion

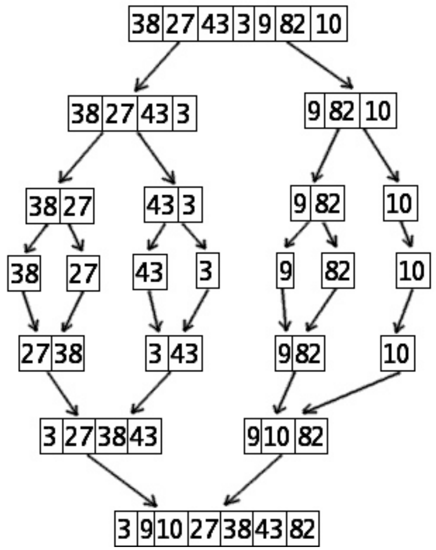
procedure $T \hookleftarrow$ FUSION($T_1[n_1]$, $T_2[n_2]$)
    $j, k \leftarrow 1$
    for $i \leftarrow 1$ to n **do**
        if $(j \leq n_1) and ((k > n_2) or (T_1[j] \leq T_2[k]))$ **then**
            $T[i] \leftarrow T_1[j], \quad j \leftarrow j + 1$
        else
            $T[i] \leftarrow T_2[k], \quad k \leftarrow k + 1$
        end if
    end for
    return T
end procedure

Algorithm 18 Tri par fusion

procedure $T \hookleftarrow$ TRI_FUSION_REC($T[n]$)
    if $n == 1$ **then return** T **end if**
    $n_1 \leftarrow n/2, \quad n_2 \leftarrow n - n_1$
    for $i \leftarrow 1$ to n_1 **do** $T_1[i] \leftarrow T[i]$ **end for**
    for $i \leftarrow 1$ to n_2 **do** $T_2[i] \leftarrow T[n_1 + i]$ **end for**
    $T_1 \leftarrow TRI\_FUSION\_REC(T_1[n_1])$
    $T_2 \leftarrow TRI\_FUSION\_REC(T_2[n_2])$
    $T \leftarrow FUSION(T_1[n_1], T_2[n_2])$
end procedure

Tri par fusion

Déroulement de l'algorithme sur un tableau de 7 éléments :



Tri par fusion

Complexité de l'algorithme de tri par fusion pour différents cas

	nb. de comparaisons	nb. de transferts	mémoire
Tous les cas	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n)$

Avantages

- Efficacité temporelle comparable au tri rapide
- La fusion requiert un temps linéaire. Il en est de même pour la copie dans les tableaux temporaires

Inconvénients

- L'algorithme repose sur un double appel récursif qui peut ne pas être efficace pour des tableaux de grande taille (pile des appels)
- Couteux en espace

Optimisations et variantes possibles

On peut limiter la mémoire utilisée à $n/2$ éléments en recopiant seulement la première des deux listes à fusionner en mémoire temporaire

Plan

1. Rappels sur les algorithmes de tri
2. Algorithmes lents (complexité quadratique) : Tris par sélection et tris par insertion
3. Algorithmes rapides (complexité quasi-linéaire)
4. Comparaison des algorithmes de tri
5. Pour aller plus loin

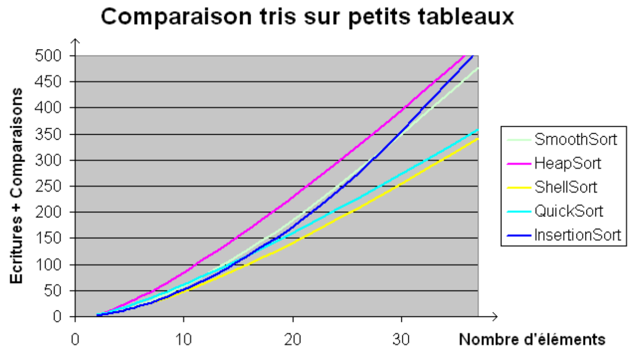
Comparaison des algorithmes

Remarques

- De **nombreux algorithmes de tri existent**
- Les **plus efficaces** ont une .complexité de l'ordre n log(n)
- Une complexité trop grande peut rendre inapplicable un algorithme

  si $n = 1000$ alors ...
- **L'algorithme de tri le plus efficace** connu à ce jour est .introspection (combinaison du tri rapide et du tri par tas)
- Les bibliothèques standard de Java et du C++ proposent chacune une fonction de tri rapide
  - std::sort pour le C++
  - Java.util pour Java

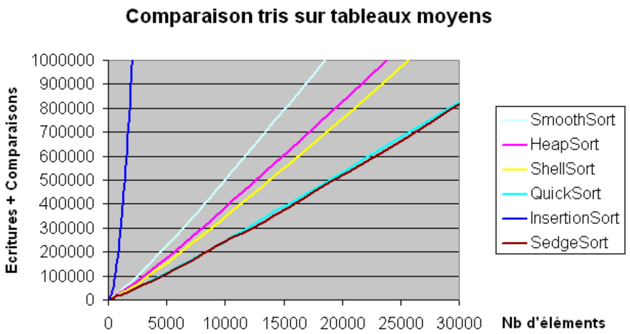
Comparaison des algorithmes sur petits tableaux



Comparaisons

- **Les algorithmes de tri les plus rapides sur des tableaux de moins de 40 éléments** sont .tri de shell et tri rapide
- Si le tri par insertion est parmi les premiers pour moins de 10 éléments, sa complexité augmente rapidement au-delà
- Le tri par tas est clairement le plus lent
- Le smoothsort obtient une position intermédiaire

Comparaison des algorithmes sur tableaux moyens



Comparaisons

- **Lorsque l'on prend un nombre d'éléments moyen (entre 50 et 30 000),** le plus rapide est le tri rapide
- La variante Sedgesort est légèrement plus rapide si l'on choisit bien la taille des petites listes, triées à la fin (ici au plus 8)
- Le smoothsort et le tri par tas changent de place
- Enfin, **la complexité du tri par insertion s'envole**

Plan

1 Rappels sur les algorithmes de tri

2 Algorithmes lents (complexité quadratique) : Tris par sélection et tris par insertion

3 Algorithmes rapides (complexité quasi-linéaire)

4 Comparaison des algorithmes de tri

5 Pour aller plus loin

Tri d'un tableau d'objets

Question ?

En Java (langage objet), comment faire pour trier (ou maintenir trié) un tableau d'objets ?

Exemple : comment comparer deux objets issus de la classe *Personne* ?

Réponse

Il faut pouvoir ordonner les objets de la classe *Personne* :

- En pratique, l'objet doit étendre l'interface *Comparable*. Cette interface implique l'écriture de la méthode de comparaison entre l'instance de l'objet et une autre instance de ce même objet.
- Cette méthode *compareTo* retourne un nombre entier qui indique :
  1. $si < 0$: l'instance est avant la deuxième instance
  2. $si = 0$: les deux instances sont à la même position
  3. $si > 0$: l'instance est après la deuxième instance
- Pour l'exemple des personnes à trier, la classe Personne implémente Comparable et la méthode compareTo : on peut par exemple trier sur la date de naissance des personnes

En pratique

Question ?

En Java, quel est l'algorithme qui se cache derrière la fonction *Arrays.sort*() ?

Réponse de la doc Java 7

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers $O(n \log(n))$ performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Références

Bibliographie

Des éléments de ce cours sont empruntés de [Caraty(2013), Cormen(2011)]

M. J. Caraty.
CM 7 – Algorithmes de tri.
Module M1103 (Structures de données et algorithmes fondamentaux), IUT Informatique, Université Paris Descartes, 2013.

T. Cormen.
Introduction à l'algorithmique.
Dunod, 2011.