

INSTITUTE FOR RESEARCH
IN IMMUNOLOGY
AND CANCER



Université 
de Montréal

Structures et méthodes pour trier

François Major

Francois.Major@UMontreal.CA

www.major.iric.ca

Structures et méthodes pour trier

Objectifs :

1. Développer un contexte pour étudier les mécanismes et les performances des méthodes de tri
2. Présenter les méthodes de tri les plus classiques et en faire l'analyse mathématique
3. Être en mesure de choisir la bonne méthode de tri pour ses applications

Contenu :

1. Introduction
2. Par insertion
3. Par la médiane
4. Rapide
5. Par sélection
6. Par monceaux
7. Par fusion (sera vu en démonstration)
8. Par comptage
9. Avec seaux
10. Critères de sélection d'une méthode de tri

Introduction

On peut tirer profit d'une liste ordonnée, ici celle des noms de Provinces et territoires du Canada

Manitoba
Alberta
Québec
Île-du-Prince-Édouard
Colombie-Britannique
Nouvelle-Écosse
Terre-Neuve
Nouveau-Brunswick
Saskatchewan
Yukon
Ontario
Territoires du Nord-Ouest
Nunavut



Trier consiste à trouver la permutation des éléments qui respecte des contraintes d'ordre préfixées

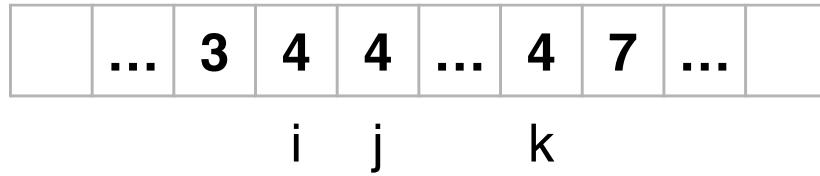
Trier une collection, A, consiste à ordonner les éléments tel que :

Si $A[i] < A[j]$, alors $i < j$.

S'il y a des éléments dupliqués, alors ces éléments doivent être positionnés les uns à la suite des autres, càd

si $A[i] = A[j]$ dans une collection triée, alors il ne peut exister un k tel que $i < k < j$ et $A[i] \neq A[k]$.

ex)



La collection triée A est une permutation des éléments qui formaient originalement A.

Pour trier une collection, on doit pouvoir comparer de manière non ambiguë tous ses éléments

On doit pouvoir admettre un ordre total sur tous les éléments de la collection. Pour n'importe quelle paire d'éléments, p et q, dans la collection, un des trois prédictats suivants doit être vrai :

$p = q$; $p < q$; ou $p > q$

Normalement, les langages de programmation inclut un opérateur d'ordre sur les types primitifs comme les entiers, le réels et les caractères.

Pour des types composites, comme les chaînes de caractères, on suit l'ordre lexical sur chacun des éléments individuels, réduisant ainsi une comparaison complexe en une série de comparaison sur des types primitifs.

Par exemple, “alligator” < “alphabet” < “alternate”, en comparant les lettres de gauche à droite et jusqu'à ce qu'il n'y en ait plus, “ant” < “anthem”.

Comparer se complique quand on considère tous les caractères, toutes les langues et toutes les cultures !

Est-ce que “A” > “a” ? Est-ce que “è” > “ê” ? Est-ce que “œ” > “o” ?

On doit se référer au standard Unicode www.unicode.org/versions/latest qui utilise des encodages comme UTF-16 pour représenter chaque caractère individuel jusqu'à 4 mots. Le consortium Unicode, www.unicode.org, a développé un standard, “the collation algorithm” pour s'occuper des règles d'ordre dans les différentes langues et cultures (Davis & Whistler 2008).



Ken Whistler
Technical Director

Dr. Ken Whistler is working at SAP in database software, implementing Unicode in database-related products. Dr. Whistler was formerly at Metaphor, Inc., where he helped design and implement the Unicode-based internationalization of the Metaphor Data Interpretation Systems. He has a BA in Chinese from Stanford University, 1972 and Ph.D. in Linguistics from the University of California, Berkeley, 1980. He pursued an early career in Sinology, learning both Japanese and Chinese in the course of studying in Japan and in Taiwan. His graduate work focused on the Native American languages of California, including an extended period of field work, archival work, and lexicography. He has developed and marketed text analysis software for linguists.



Mark Davis
President & CLDR-TC Vice-Chair

Dr. Mark Davis co-founded the Unicode project and has been the president of the Unicode Consortium since its incorporation in 1991. He is one of the key technical contributors to the Unicode specifications. Mark founded and was responsible for the overall architecture of ICU (the premier Unicode software internationalization library), and architected the core of the Java internationalization classes. He also founded and is the chair of the Unicode CLDR project, and is a co-author of BCP 47 "Tags for Identifying Languages" (RFC 4646 and RFC 4647), used for identifying languages in all XML and HTML documents. Since the start of 2006, Mark has been working on software internationalization at Google, focusing on effective and secure use of Unicode (especially in the index and search pipeline), the software internationalization libraries (including ICU), and stable international identifiers.

On assume une fonction de comparaison pouvant s'appliquer sur des éléments primitifs ou enregistrements complexes

Pour les algorithmes que nous verrons, on assume l'existence d'une fonction de comparaison, `<(__lt__)`, `>(__gt__)`, ou une fonction « à la Java », telle que *cmp*, qui compare deux éléments, p et q, et qui retourne :

0, si $p = q$,

un nombre négatif, si $p < q$

un nombre positif, si $p > q$

Si les éléments sont des enregistrements complexes, la fonction de comparaison peut comparer une des valeurs, e.g. la clé, des enregistrements ou un champs présélectionné.

ex) Dans un aéroport, on affiche les vols par ordre ascendant des destinations ou des heures de départ, alors que les numéros de vols semblent désordonnés.

Un algorithme de tri stable maintient l'ordre des éléments égaux dans la permutation triée finale

Quand deux éléments, $a[i]$ et $a[j]$, sont considérés égaux par la fonction de comparaison, il arrive de vouloir garder cet ordre dans la collection triée finale.

Si $i < j$, alors la position finale pour $a[i]$ doit être à gauche de la position finale pour $a[j]$.

Un algorithme de tri qui garantie cette propriété est considéré stable.

Soit une collection originale, A, des vols déjà triés par l'heure de départ dans la section du haut du tableau sur la prochaine diapositive (sans regard à la compagnie aérienne ni à la ville de destination). Si on trie ensuite cette collection par la destination, le résultat d'appliquer un tri stable est montré dans la section du bas du tableau.

Un tri stable sur la ville de destination garde l'ordre des heures des départs d'une collection pré-triée sur ces heures de départ

Destination	Airline	Flight	Sched
Buffalo	Air Tran	549	10:42 AM
Atlanta	Delta	1097	11:00 AM
Baltimore	Southwest	836	11:05 AM
Atlanta	Air Tran	872	11:15 AM
Atlanta	Delta	28	12:00 PM
Boston	Delta	1056	12:05 PM
Baltimore	Southwest	216	12:20 PM
Austin	Southwest	1045	1:05 PM
Albany	Southwest	482	1:20 PM
Boston	Air Tran	515	1:21 PM
Baltimore	Southwest	272	1:40 PM
Atlanta	AllItalia	3429	1:50 PM

Destination	Airline	Flight	Sched
Albany	Southwest	482	1:20 PM
Atlanta	Delta	1097	11:00 AM
Atlanta	Air Tran	872	11:15 AM
Atlanta	Delta	28	12:00 PM
Atlanta	AllItalia	3429	1:50 PM
Austin	Southwest	1045	1:05 PM
Baltimore	Southwest	836	11:05 AM
Baltimore	Southwest	216	12:20 PM
Baltimore	Southwest	272	1:40 PM
Boston	Delta	1056	12:05 PM
Boston	Air Tran	515	1:21 PM
Buffalo	Air Tran	549	10:42 AM

Tri stable des informations sur des vols de départ. Trier par heure de départ (haut). Résultat d'appliquer un tri stable sur le tableau du haut sur la destination (bas).

Un des résultats fondamentaux en matière de tri est qu'en comparant deux à deux les n éléments d'une collection, on ne peut pas trier cette collection en moins que $O(n \log n)$

On s'intéresse aux performances des algorithmes de tri en pire cas, en moyenne et en meilleur cas, le dernier étant le plus difficile à montrer dans le cas des algorithmes de tri.

Un résultat fondamental en informatique est qu'aucun algorithme pour trier une collection de n éléments par comparaison d'éléments deux à deux peut faire mieux que $O(n \log n)$ en moyenne ou en pire cas.

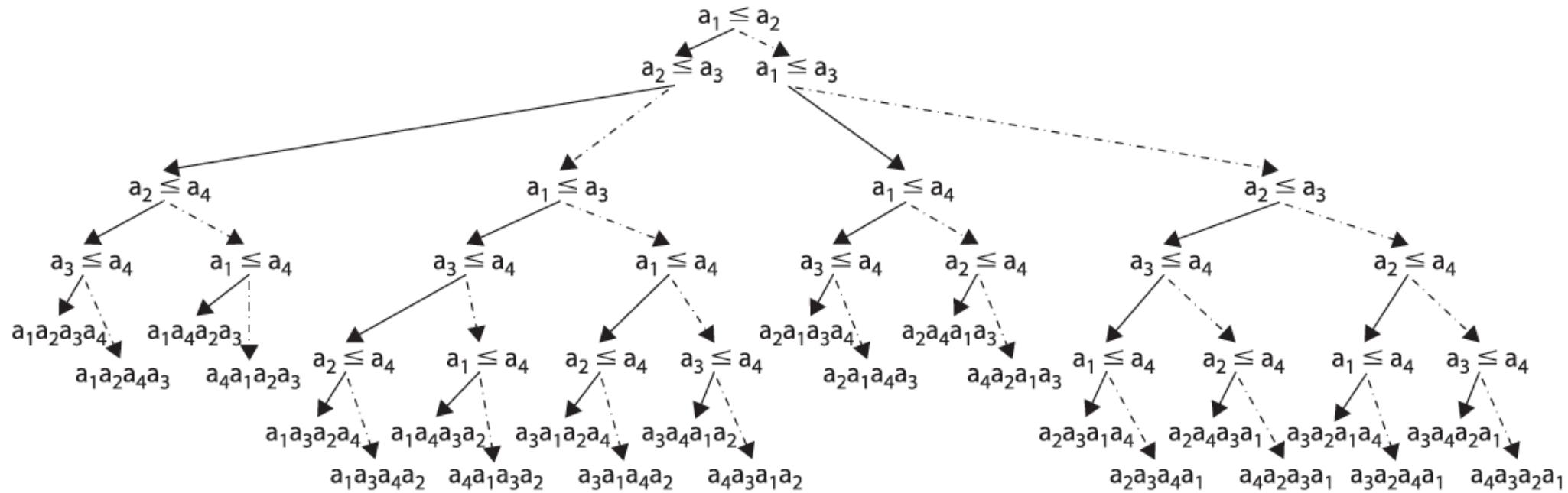
Voici la preuve :

Pour n éléments, il existe $n!$ permutations de ces éléments. Tout algorithme qui tri par comparaison de paires d'éléments correspond à un arbre binaire de décisions. Chaque permutation est représentée par une feuille. Les noeuds et les chemins de la racine jusqu'aux feuilles correspondent à des séquences des comparaisons.

La hauteur de l'arbre est le nombre de comparaisons (ou de noeuds) du plus long chemin de la racine à une feuille.

La figure de la diapositive suivante montre un tel arbre, de hauteur 5 puisque 5 décisions sont nécessaires pour 16 des permutations, ou 4 dans le cas de 8 permutations.

La hauteur d'un arbre binaire de décisions pour un tri de 4 éléments est 5, correspondant au nombre nécessaire de comparaisons à effectuer pour arriver à la bonne permutation



Arbre binaire de décisions pour 4 éléments à trier
 $(4! = 24; 4 \log 4 = 5.5)$

La hauteur d'un arbre binaire de n éléments est comprise entre un minimum de $\lceil \log(n+1) \rceil$ (balancé) et un maximum de $n-1$ (pire cas)

Un arbre binaire de décisions de hauteur h possède au plus 2^h feuilles (e.g. si $h = 5$, on a un arbre d'au plus 32 feuilles).

N'importe quel arbre de décision pour trier n éléments possède une hauteur $\Omega(n \log n)$.

Preuve :

$$\begin{aligned} 2^h &\geq n! \Rightarrow h \geq \log(n!) \text{ (e.g. } 2^5 = 32 \geq 4! = 24 \Rightarrow 5 \geq \log(24) = 3.18 \\ &= \log(n(n-1)(n-2)\dots 2) \\ &\geq \log((n/2)^{n/2}); \text{ on prend un borne inférieure du produit} \\ &= n/2 \log(n/2); \text{ puisque } \log(x^y) = y * \log(x) \\ &= \Omega(n \log n) \end{aligned}$$

Tout est beau dans le meilleur des mondes !

Pour toutes les méthodes de tri que nous verrons, on assume que les données résident en mémoire, soit dans un tableau de valeurs ou de pointeurs vers les éléments, que les fonctions d'accès aux données et de la fonction de comparaison (e.g. *cmp*) sont $\in O(1)$.

Tri par insertion

Une recherche séquentielle n'est pas raisonnable pour mixer beat sur beat

Boîte de disques de vinyl



Algorithm performance
(best, average, and worst)

Name of the algorithm

Concepts

SEQUENTIAL SEARCH			Array
Best	Average	Worst	
$O(1)$	$O(n)$	$O(n)$	
search (A, t)			
1. for $i = 1$ to n do			<i>search (A, 15)</i>
2. if ($A[i] = t$) then			
3. return true			
4. return false			
end			
Pseudocode description			Small example

Fiche technique de la recherche séquentielle

Est-ce raisonnable pour un/une DJ de garder pêle-mêle dans une boîte (ou sur un fichier ou une liste) les morceaux à jouer ?

Est-ce que le/la DJ cherche de manière séquentielle dans sa boîte pour trouver un morceau qu'il pourra mixer avec le morceau courant ?

DJing, règle #1 :

Mixer des titres dont les tempos sont le plus rapprochés possibles, càd à $\pm 2\%$ du tempo courant.

Si le morceau courant a un tempo de 120 bpm, considérer un morceau dont le tempo est entre 118 et 122 est raisonnable, à moins d'utiliser une technique particulière ou si le morceau courant termine abruptement.

Maintenance de la collection des morceaux par insertion

Une façon pour le/la DJ d'organiser sa boîte de vinyls est d'insérer les nouveaux morceaux dans une boîte déjà triée.



Collection triée



Insertion au bon endroit



Nouveau morceau à insérer

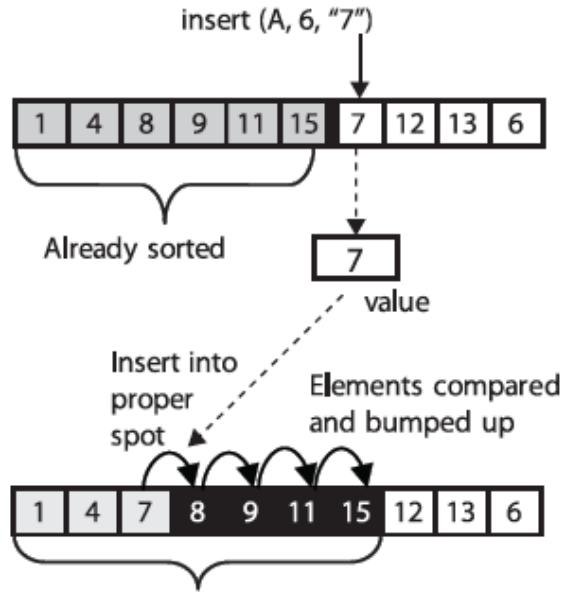


Nouvelle collection triée

Retourne sur la pile à insérer

Tri par insertion des nouveaux morceaux

Le tri par insertion peut se faire directement dans le tableau à trier et invoque $n-1$ fois la fonction “insert”

INSERTION SORT		
Best	Average	Worst
$O(n)$	$O(n^2)$	$O(n^2)$
		■■■■■ Array
sort (A) 1. for $i = 1$ to $n - 1$ do 2. insert ($A, i, A[i]$) end		 insert (A, pos, value) 1. $i = pos - 1$ 2. while ($i \geq 0$ and $A[i] > value$) do 3. $A[i + 1] = A[i]$ 4. $i = i - 1$ 5. $A[i + 1] = value$ end

Fiche technique pour le tri par insertion

“insert” décale les éléments qui sont plus grands que la valeur à insérer vers la droite jusqu’à ce que l’emplacement correct de la valeur soit trouvé

```
insert( A, 1, 9 )
i = 0;
while(i >= 0 && A[i] > 9)
    A[1] = A[0] = 15;
    i = -1;
A[0] = 9
```

```
insert( A, 2, 8 ) →
i = 1;
while(i >= 0 && A[i] > 8)
    A[2] = A[1] = 15;
    i = 0;
A[1] = A[0] = 9;
i = -1;
A[0] = 8
```

15	09	08	01	04	11	07	12	13	06	05	03	16	02	10	14
09	15	08	01	04	11	07	12	13	06	05	03	16	02	10	14
08	09	15	01	04	11	07	12	13	06	05	03	16	02	10	14
01	08	09	15	04	11	07	12	13	06	05	03	16	02	10	14
01	04	08	09	15	11	07	12	13	06	05	03	16	02	10	14
01	04	08	09	11	15	07	12	13	06	05	03	16	02	10	14
01	04	07	08	09	11	15	12	13	06	05	03	16	02	10	14
01	04	07	08	09	11	12	15	13	06	05	03	16	02	10	14
01	04	07	08	09	11	12	13	15	06	05	03	16	02	10	14
01	04	06	07	08	09	11	12	13	15	05	03	16	02	10	14
01	04	05	06	07	08	09	11	12	13	15	03	16	02	10	14
01	03	04	05	06	07	08	09	11	12	13	15	16	02	10	14
01	03	04	05	06	07	08	09	11	12	13	15	16	02	10	14
01	02	03	04	05	06	07	08	09	11	12	13	15	16	10	14
01	02	03	04	05	06	07	08	09	10	11	12	13	15	16	14
01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16

Progression du tri par insertion sur un petit tableau de valeurs entières

Le tri par insertion est le seul tri que nous analyserons qui possède un meilleur cas en $O(n)$

Contexte :

On utilise le tri par insertion quand on a un petit nombre d'éléments à trier ou que les éléments dans la collection sont déjà “presque” triés.

Petit comment ? Ça dépend de l'ordinateur et du langage de programmation, et aussi du type des éléments (faut le tester empiriquement).

Forces :

On a besoin que de l'espace occupé initialement par la collection, n , + 1 pour le tampon de la valeur courante, donc $n+1$ éléments $\in O(n)$ pour l'espace mémoire.

Conséquences :

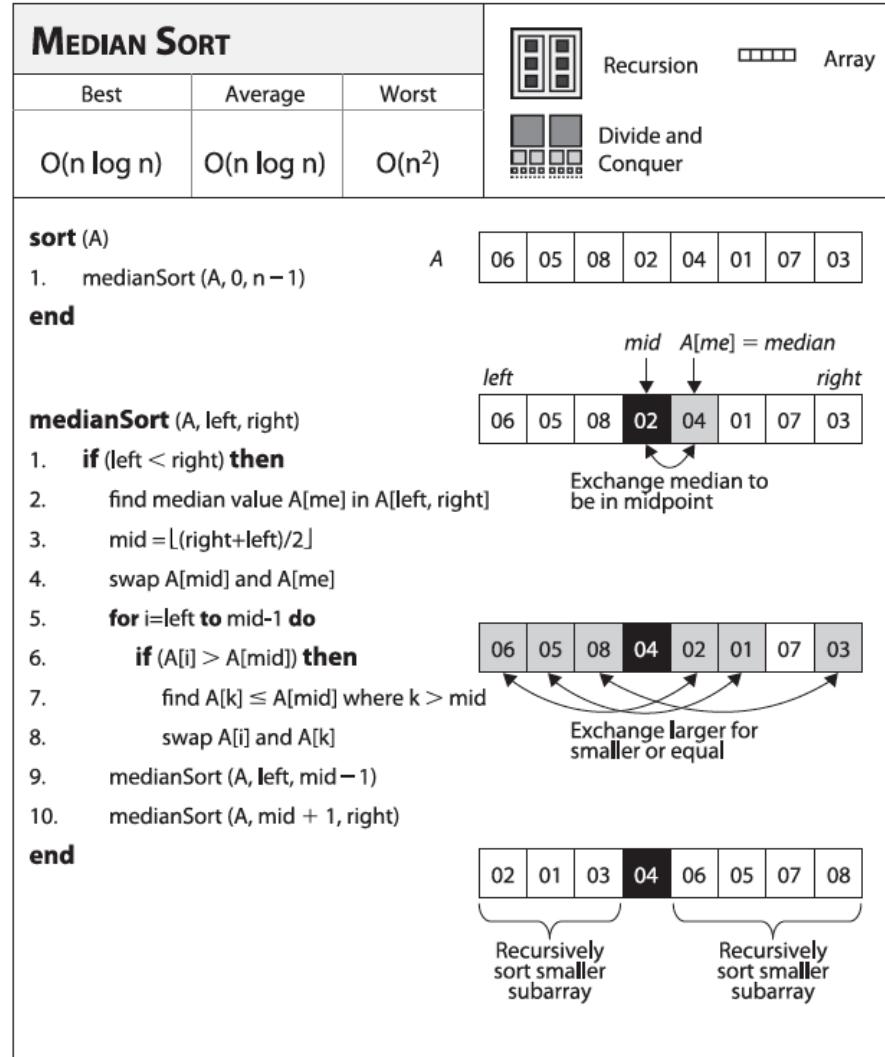
Plus la collection originale est triée et meilleures sont les performances du tri par insertion.

Si le tableau est déjà trié, “insert” n’entre même pas dans la boucle, donc une seul test, $O(1)$, pour les $n-1$ valeurs, soit dans le meilleur cas $n-1$ décalages $\in O(n)$.

Par contre, si le tableau est trié inversement à l'ordre désiré, “insert” décale toutes les valeurs chaque fois, $O(n)$, pour les $n-1$ valeurs, soit au pire cas $n * (n-1)$ décalages $\in O(n^2)$.

Tri par la médiane

Le tri par la médiane applique l'approche diviser-pour-régner



Fiche technique pour le tri par la médiane

On divise jusqu'à des partitions de moins de 4 éléments

(qui se retrouvent nécessairement triées après avoir placé la médiane au milieu et échangé au besoin les parties gauche et droite)

médiane = 8

1a



4 éléments à gauche > 8

1b



4 éléments à droite < 8

1c



parties gauche et droite
prêtes à trier

2a



2 appels récursifs pour
trier parties gauche et
droite du tableau original

2b



2c



appels récursifs pour
trier des parties de plus
en plus petites

3a



Pour une partie de
moins de 4 éléments,
placer la médiane au
milieu en établit l'ordre

3b



3c



4a



4b



4c



■ position milieu
■ médiane/éléments à échanger

Progression du tri par la médiane

La performance du tri par la médiane dépend de l'efficacité de déterminer l'élément médiane d'un tableau non trié

On désire une fonction $p = \text{partition}(\text{gauche}, \text{droite}, \text{indexPivot})$ qui sélectionne l'élément $A[\text{indexPivot}] = \text{pivot}$, qui divise $A[\text{gauche}, \text{droite}]$ en deux partitions égales : une première partition dont les éléments sont $\leq \text{pivot}$ et une deuxième partition où les éléments sont $> \text{pivot}$.

gauche $\leq \text{indexPivot} \leq$ droite et p est la valeur renournée correspondante à l'index dans la partition $A[\text{gauche}, \text{droite}]$ où la valeur du pivot se retrouve.

Un code Python pour partitioner en temps linéaire un tableau autour d'une valeur pivot

```
def partition( tab, gauche, droite, indexPivot ):  
  
    pivot = tab[indexPivot]  
  
    # déplacer le pivot à la fin du tableau  
    swap( tab, indexPivot, droite )  
  
    # toutes les valeurs <= pivot sont déplacées au début du tableau et  
    # le pivot est inséré juste après elles.  
    pivpos = gauche  
    for i in range( gauche, droite ):  
        if tab[i] <= pivot:  
            swap( tab, pivpos, i )  
            pivpos += 1  
  
    swap( tab, pivpos, droite )  
    return pivpos  
  
  
def swap( tab, i, j ):  
    tmp = tab[i]  
    tab[i] = tab[j]  
    tab[j] = tmp
```

Partitioner autour d'une valeur pivot en temps linéaire

	gauche	indexPivot		droite		
i...	15 09 08 01 04 11 07 12 13 06 05 03 16 02 10 14	3 0	4 1	10 2	11 3	13 4
pivpos...						
échange(pivot, droite)	15 09 08 01 04 11 07 12 13 14 05 03 16 02 10 06					
pivpos = gauche	(01) 09 08 15 04 11 07 12 13 14 05 03 16 02 10 06					
pour i = gauche to droite-1	01 (04) 08 15 09 11 07 12 13 14 05 03 16 02 10 06					
si tab[i] <= pivot	01 04 (05) 15 09 11 07 12 13 14 08 03 16 02 10 06					
échange(i, pivpos++)	01 04 05 (03) 09 11 07 12 13 14 08 15 16 02 10 06					
échange(droite, pivpos)	01 04 05 03 (02) 11 07 12 13 14 08 15 16 09 10 06					
	01 04 05 03 02 (06) 07 12 13 14 08 15 16 09 10 11					
		indexPivot	position médiane			

valeurs de i où $\text{tab}[i] \leq \text{pivot}$
valeurs de pivpos lors des échanges

partition(0, 15, 9) retourne 5 tout en mettant à jour tab

À la fin,
 $\text{tab}[\text{gauche}, p]$ contient les éléments $\leq \text{pivot}$
 $\text{tab}[p+1, \text{droite}]$ contient les éléments $> \text{pivot}$

Des appels récursifs de **partition** jusqu'à ce que $p = \text{milieu du tableau}$ détermine la valeur médiane

Partition ne trouve pas la valeur de la médiane !

La valeur rentrée, p , est la position du pivot qui se trouve dans notre exemple à gauche de où on voudrait le retrouver, soit au milieu du tableau.

Aucune des valeurs à gauche de p ne représente la médiane.

Il suffit d'invoquer de manière récursive **partition**, cette fois avec une valeur de **indexPivot** différente dans la partition de droite, $A[p+1, \text{droite}]$ jusqu'à ce qu'elle retourne $p = \text{position de la valeur médiane}$.

Si le pivot est la médiane, partition va séparer le tableau en 2 partitions égales et séparées au milieu.

Donc, on cherche le $k^{\text{ème}}$ élément de la collection, ici $k = 8$

si $k = p+1$, alors la valeur du pivot est le $k^{\text{ème}}$ élément.

si $k < p+1$, alors le $k^{\text{ème}}$ élément de tab est le $k^{\text{ème}}$ élément de tab[gauche, p]

si $k > p+1$, alors le $k^{\text{ème}}$ élément de tab est le $(k-p)^{\text{ème}}$ élément de tab[p+1, droite]

Un code Python pour déterminer la valeur de la médiane en temps moyen linéaire

```
"""En temps moyen linéaire, trouve la position du kième élément de tab, qui est modifié au fur et à mesure de l'exécution.  
Note 1 <= k <= droite-gauche+1. Pire cas quadratique, O(n^2).  
"""  
  
def selecteK( tab, k, gauche, droite ):  
    i = random.randint( gauche, droite )  
    print( "i = ", i )  
    indexPivot = partition( tab, gauche, droite, i )  
    if ( gauche + k - 1 ) == indexPivot:  
        return indexPivot  
  
    # continuer la boucle, réduisant l'intervalle de manière appropriée.  
    # Si on cherche dans la partition de gauche, alors on peut garder k.  
    if ( gauche + k - 1 ) < indexPivot:  
        return selecteK( tab, k, gauche, indexPivot-1 )  
    else:  
        return selecteK( tab, k - ( indexPivot - gauche + 1 ), indexPivot + 1, droite)
```

Pour le choix initial de l'index du pivot, on peut utiliser la première ou la dernière position, gauche ou droite, ou une position aléatoire.

Si le pivot à chaque itération est mal choisi, alors les performances de selecteK se dégradent vers le pire cas, soit $O(n^2)$. En moyenne, cependant, les performances de selecteK sont en $O(n)$.

Une implantation Python complète du tri par la médiane

```
"""Trier le tableau tab[gauche, droite] utilisant la méthode de tri
par la médiane.
"""

def triMediane( tab, gauche, droite):

    # si la tranche du tableau à trier possède 1 (ou moins) éléments,
    # c'est fini !
    if droite <= gauche:
        return

    # obtenir l'index du milieu du tableau
    # et la position de l'élément médiane
    # 1 <= k <= droite-gauche-1).
    milieu = (droite - gauche + 1)//2
    mediane = selecteK( tab, milieu, gauche, droite)

    triMediane( tab, gauche, gauche + milieu - 1 )
    triMediane( tab, gauche + milieu + 1, droite )
```

En voulant parfaitement diviser le problème en sous-problèmes de tailles égales, on crée du travail inutile

En créant successivement des sous-problèmes de la moitié de la taille du problème original, la méthode de tri par la médiane a une performance en moyenne dans $O(n \log n)$.

Bien que la division (pour régner) est optimale, le tri par la médiane effectue plus de travail que nécessaire ! Le problème est avec `selecteK` qui peut dégénérer en $O(n^2)$ lorsque les éléments du tableau sont déjà triés.

Dans le pire cas, partition n-1 fois sans rien changer (cas où on prend indexPivot aléatoire)

	gauche								indexPivot				droite			
i...	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pivpos...	0	1	2	3	4	5	6	7	8	9	9	10	11	12	13	14
échange(pivot, droite)	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
pivpos = gauche	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
pour i = gauche to droite-1	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
si tab[i] <= pivot	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
échange(i, pivpos++)	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
échange(droite, pivpos)	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
	1	2	3	4	5	6	7	8	9	16	11	12	13	14	15	10
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

position médiane indexPivot

partition(0, 15, 9) retourne 9

valeurs de i où tab[i] <= pivot
valeurs de pivpos lors des échanges

Dans le pire cas, partition n-1 fois sans rien changer (cas où on prend indexPivot à gauche)

	indexPivot gauche								droite							
i ...	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pivpos ...	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
échange(pivot, droite)	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
pivpos = gauche	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
pour i = gauche to droite-1	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
si tab[i] <= pivot	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
échange(i, pivpos++)	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
échange(droite, pivpos)	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
	16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	indexPivot								position médiane							
	partition(0, 15, 0) retourne 0															

valeurs de i où tab[i] <= pivot
valeurs de pivpos lors des échanges

Heureusement, il existe un fonction de sélection du K^{ème} élément qui performe en temps linéaire en pire cas

C'est la fonction Blum-Floyd-Pratt-Rivest-Tarjan (BFPRT) (Blum et al. 1973).

n	Randomized pivot selection	Leftmost pivot selection	Blum-Floyd-Pratt-Rivest-Tarjan pivot selection
32,768	0.0187	9.0479	0.0388
65,536	0.0743	47.3768	0.1065
131,072	0.0981	236.629	0.361

Performance du tri par la médiane en pire cas

n	Randomized pivot selection	Leftmost pivot selection	Blum-Floyd-Pratt-Rivest-Tarjan pivot selection
256	0.00009	0.000116	0.000245
512	0.000197	0.000299	0.000557
1,024	0.000445	0.0012	0.0019
2,048	0.0013	0.0035	0.0041
4,096	0.0031	0.0103	0.0128
8,192	0.0082	0.0294	0.0256
16,384	0.018	0.0744	0.0547
32,768	0.0439	0.2213	0.4084
65,536	0.071	0.459	0.5186
131,072	0.149	1.8131	3.9691

Performance du tri par la médiane en moyenne

BFPRT doit “diviser-pour-éviter” le pire cas de partition et ce travail supplémentaire se traduit en une performance en moyenne plus lente

À cause des constantes plus grandes, l’algorithme BFPRT est plus coûteux à exécuter et performe moins bien en moyenne que selecteK avec choix aléatoire.

Il trouve une bonne approximation de la valeur médiane d’une collection désordonnée en formant $n/4$ groupes de 4 éléments (il ignore jusqu’à 3 éléments qui ne formeraient pas un groupe de 4, dans lequel la médiane peut se retrouver). Il cherche la médiane des médianes de chaque groupe, à quel prix ?

On a vu qu’on a besoin d’au moins 5 comparaisons pour trier un groupe de 4, donc cette étape coûte $(n/4) * 5 = 1.25 n \in O(n)$. La valeur médiane de chaque groupe se retrouve au 3^{ème} élément. On les met dans une nouvelle collection et on applique récursivement BFPRT pour trouver l’approximation de la médiane.

Tri rapide (quicksort)

Le tri rapide est plus simple que le tri par la médiane

On a vu qu'en choisissant aléatoirement `indexPivot` permet au tri par la médiane d'avoir une performance en moyenne de `selecteK` en $O(n)$. Cette approche est plus rapide en pratique sur des tailles de collections assez importantes que la stratégie utilisée par BFPRT.

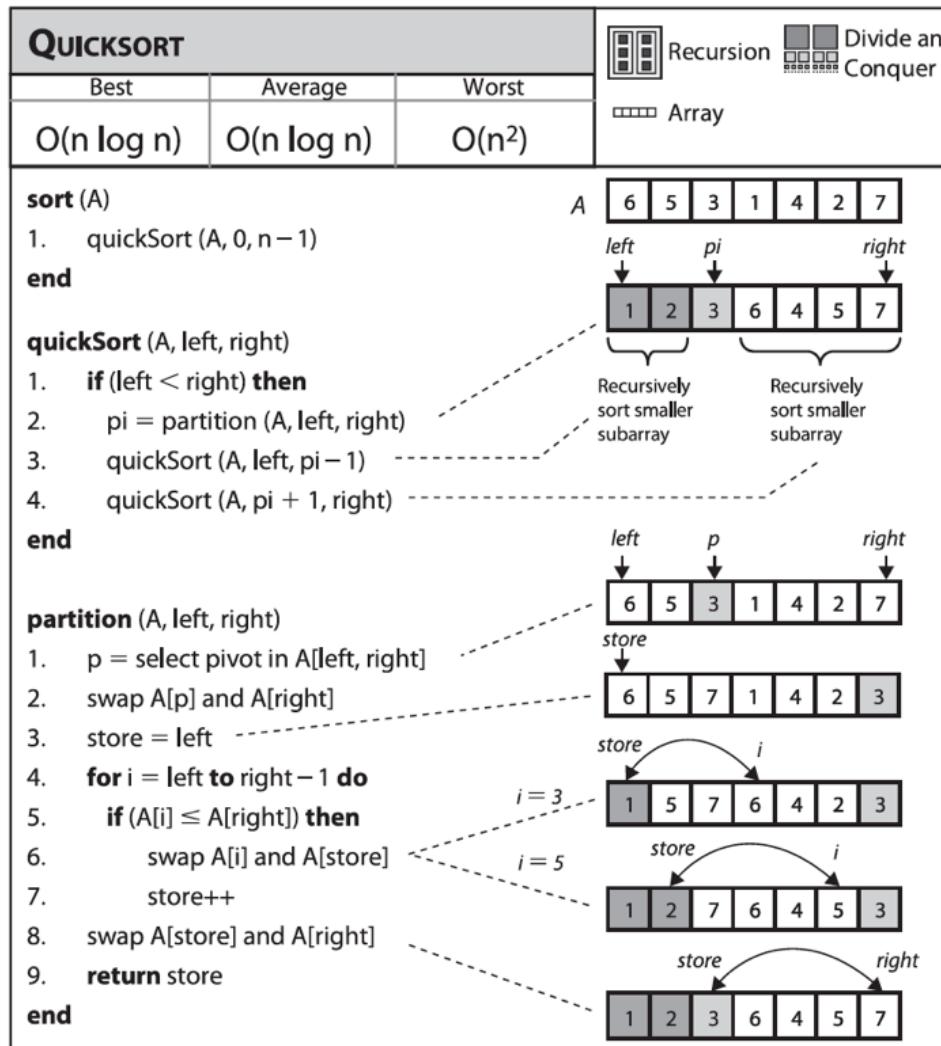
Est-ce qu'on peut simplifier l'algorithme du tri par la médiane sans occasionner de pénalité sur sa performance ? Est-ce que cet algorithme « plus simple » pourrait aussi performer mieux dans certains cas ?

Le tri rapide introduit par Hoare est plus simple que le tri par la médiane et il utilise les mêmes concepts, c'est pourquoi on introduit d'abord le tri par la médiane.

Dans le tri rapide, on ne cherche plus la valeur médiane. On sélectionne un pivot selon une stratégie pré-établie : par exemple aléatoirement, sinon on peut prendre à gauche, à droite, au milieu, ...

Le tri rapide se fait en 2 étapes : 1) autour d'un pivot, on crée 2 partitions, une à gauche (contenant les éléments \leq pivot) et une à droite (contenant les éléments $>$ pivot; et, 2) on trie récursivement chacune de ces partitions.

Le tri rapide applique l'approche diviser-pour-régner sans déterminer la médiane



N.B. La nature aléatoire du choix du pivot rend l'analyse mathématique de la performance en moyenne du tri rapide dans $O(n \log n)$ assez difficile. Si ça vous intéresse, vous pouvez aller voir dans Cormen et al. 2001.

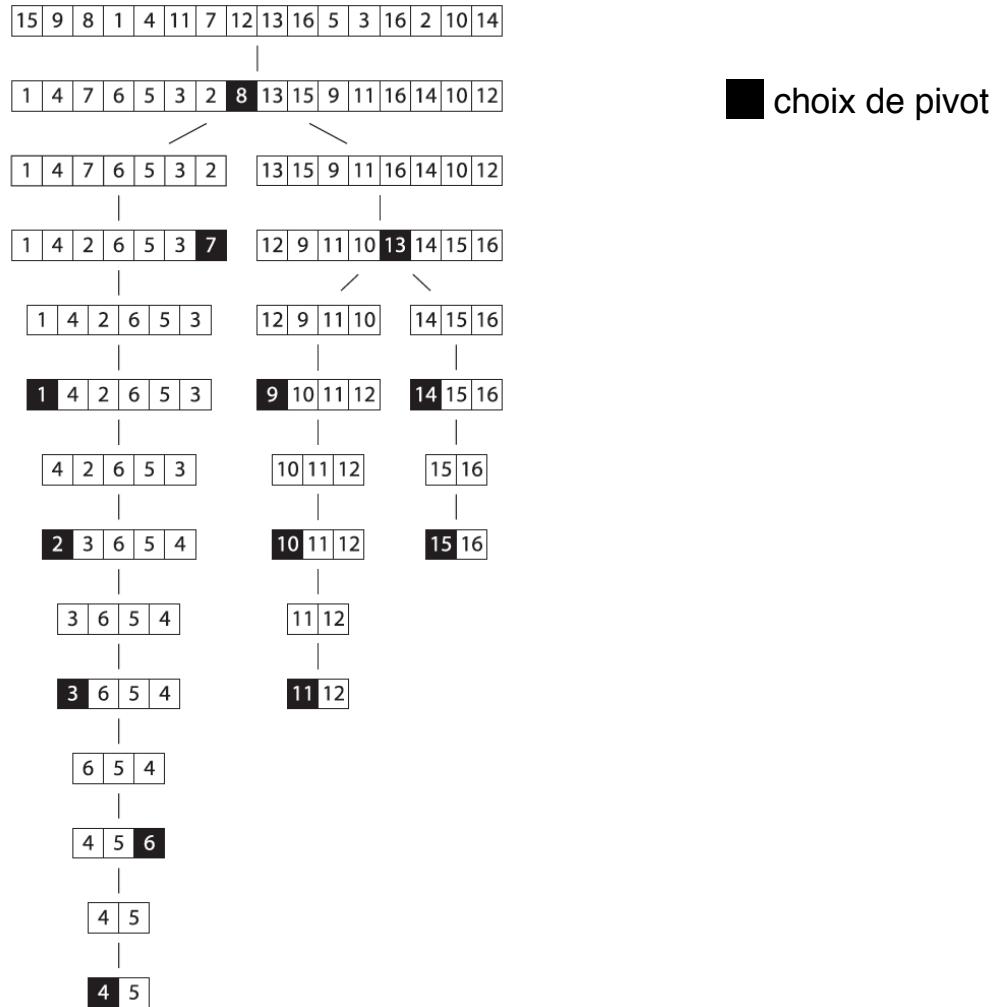
Fiche technique pour le tri rapide

On subdivise les partitions aléatoirement, ce qui génère des partitions qui peuvent différer



Progression du tri rapide

Même collection que la diapositive précédente mais d'autres partitions



Progression différente du tri rapide sur la même collection que la diapositive précédente

Code Python pour le tri rapide

```
"""Trier le tableau tab avec la méthode de tri rapide.
"""

def triRapide( tab, gauche, droite):

    if droite <= gauche:
        return

    # partitions
    indexPivot = random.randint( gauche, droite )
    indexPivot = partition( tab, gauche, droite, indexPivot )

    triRapide( tab, gauche, indexPivot-1 )
    triRapide( tab, indexPivot+1, droite )
```

Analyse du tri rapide

Le tri rapide est quadratique en pire cas, soit lorsqu'on divise la collection de n éléments en une partition vide et l'autre de $n-1$ éléments (n.b. l'élément pivot est retiré à chaque itération et garantie des sous-problèmes de tailles de plus en plus petites).

Le choix du pivot est déterminant et des variations de l'algorithme utilisent des approches différentes pour choisir le pivot.

De manière étonnante, le choix aléatoire du pivot permet au tri rapide de mieux performer en moyenne que la plupart des autres algorithmes. De plus, il existe de nombreuses optimisations et améliorations du tri rapide qui en ont fait le plus performant de tous les algorithmes de tri.

Le tri rapide possède la même complexité que le tri par la médiane. Cependant, son implantation beaucoup plus simple engendre des performances empiriques qui surpassent le tri par la médiane

Dans le cas idéal, le tri rapide divise la collection en deux partitions égales. Si ce comportement s'applique à chaque récursion, alors on observe le même comportement que pour le tri par la médiane mais sans le travail supplémentaire pour trouver la médiane, et donc en meilleur cas on est en $O(n \log n)$. Empiriquement, on observe que le tri rapide est plus performant que le tri par la médiane mais ça reste une question de constantes !

En pire cas, lorsque le plus petit ou plus grand élément est choisi comme pivot, le tri rapide passe au travers de tous les éléments du tableau, $O(n)$, pour ne trier qu'un seul élément à chaque itération. Dans le pire cas, ce processus est répété $n-1$ fois, donc on est en $O(n^2)$.

Tri par sélection

Tri par sélection sélectionne et échange du plus petit au plus grand ou du plus grand au plus petit

Étant donné une pile de vinyls à trier, on sélectionne et retire de la pile le morceau le plus lent et on le met en dernière position.

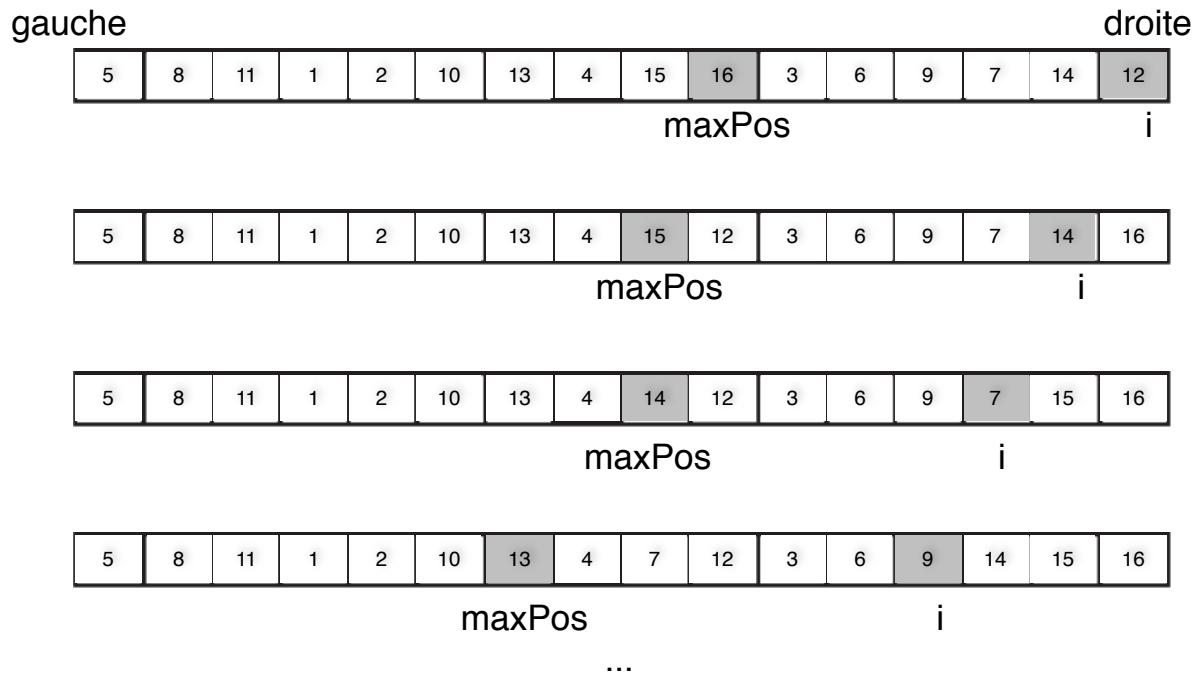
On répète pour tous les morceaux sauf le dernier (qui sera remis dans la pile après la première itération sauf s'il est déjà le plus rapide).

Code Python pour le tri par sélection

```
def selecteMax( tab, gauche, droite ):
    maxPos = gauche
    for i in range( gauche, droite+1 ):
        if( tab[i] > tab[maxPos] ):
            maxPos = i
    return maxPos

def triSelection( tab, n ):
    # répéter selecteMax et échanger avec l'endroit approprié
    for i in range( n-1, 0, -1 ):
        maxPos = selecteMax( tab, 0, i )
        if maxPos != i:
            swap( tab, i, maxPos )
```

La progression “lente” du tri par sélection...



Progression du tri par sélection

Tri par monceau (heapsort)

Le tri par monceau minimise le nombre de comparaisons avec le plus grand élément

Le tri par monceau est inspiré du tri par sélection.

Le tri par sélection nécessite $O(n)$ comparaisons pour trouver l'élément maximum. Le tri par monceau minimise ce nombre en utilisant un monceau.

Considérez dans certains sports des tournois pour identifier la meilleure équipe parmi n mais sans avoir à faire jouer cette dernière contre les $n-1$ autres. Par exemple, dans le tournoi de basketball de la NCAA (National Collegiate Athletic Association), on a 64 équipes et chaque équipe joue contre 5 autres équipes avant d'accéder à la finale. L'équipe gagnante doit donc jouer 6 matches = $\log(64)$!

C'est cette idée qui est derrière le tri par monceau.

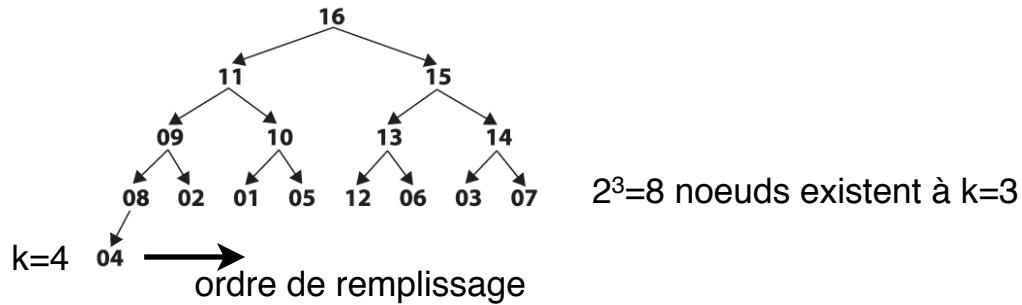
On va créer un monceau dans lequel on va piger la prochaine plus grande valeur, le champion, puis le 2^e rang, 3^e, etc. Obtenir cette valeur et mettre à jour le monceau va nous coûter $O(\log n)$ avec le monceau, comparativement à $O(n)$ avec le tri par sélection. Créer le monceau va coûter $O(n \log n)$.

Regardons ce qu'est un monceau et comment le construire...

Un monceau possède deux propriétés utiles pour le tri

Un monceau est un arbre binaire dont la structure garantie 2 propriétés :

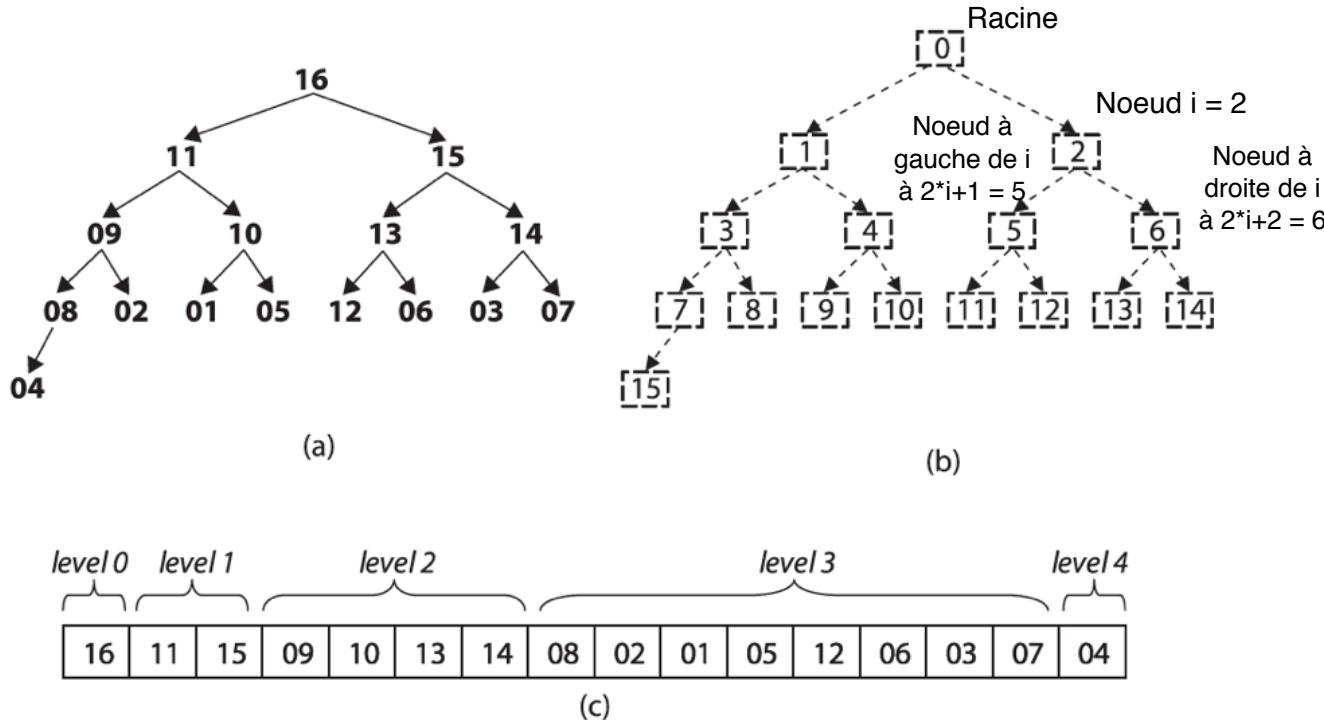
1) Propriété de forme. Une feuille à profondeur $k > 0$ existe seulement si tous les 2^{k-1} noeuds à profondeur $k-1$ existent. De plus, les noeuds sur un niveau partiellement rempli doivent être ajoutés de gauche à droite.



2) Propriété de monceau. Chaque noeud dans l'arbre contient une valeur plus grande ou égale à son ou ses deux enfants, s'il en a.

Un monceau entre dans un tableau sans perte d'information

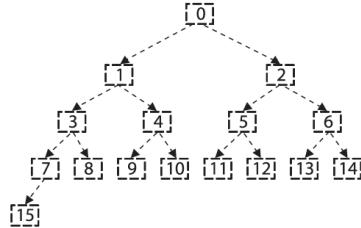
La structure rigide de sa forme permet de stocker un monceau dans un tableau sans perte d'information.



Monceau. (a) Monceau de 16 éléments. (b) Index des éléments du monceau de 16 éléments dans un tableau. (c) Monceau de 16 éléments stocké dans un tableau.

La construction du monceau applique la percolation des valeurs de l'élément milieu, $\lfloor n/2 \rfloor - 1$, jusqu'à la racine

Le tri par monceau commence par construire un monceau, en percolant ses valeurs !

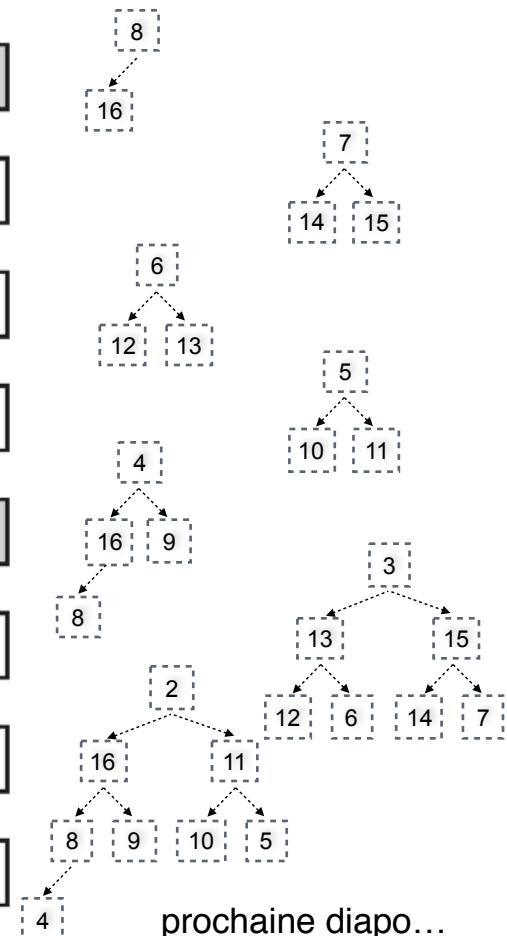


```

buildHeap (A)
1. for i =  $\lfloor n/2 \rfloor - 1$  downto 0 do
2.   heapify (A, i, n)
end

heapify (A, idx, max)
1. left = 2*idx + 1
2. right = 2*idx + 2
3. if (left < max and A[left] > A[idx]) then
4.   largest = left
5. else largest = idx
6. if (right < max and A[right] > A[largest]) then
7.   largest = right
8. if (largest ≠ idx) then
9.   swap A[i] and A[largest]
10.  heapify (A, largest, max)
end
  
```

		$\lfloor n/2 \rfloor - 1$	
7	01 02 03 04 05 06 07 16 09 10 11 12 13 14 15 08		
6	01 02 03 04 05 06 15 16 09 10 11 12 13 14 07 08		
5	01 02 03 04 05 13 15 16 09 10 11 12 06 14 07 08		
4	01 02 03 04 11 13 15 16 09 10 05 12 06 14 07 08		
3	01 02 03 16 11 13 15 08 09 10 05 12 06 14 07 04		
2	01 02 15 16 11 13 14 08 09 10 05 12 06 03 07 04		
1	01 16 15 09 11 13 14 08 02 10 05 12 06 03 07 04		
0	16 11 15 09 10 13 14 08 02 01 05 12 06 03 07 04		



prochaine diapo...

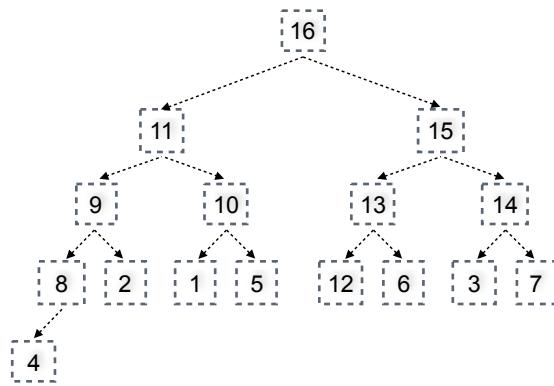
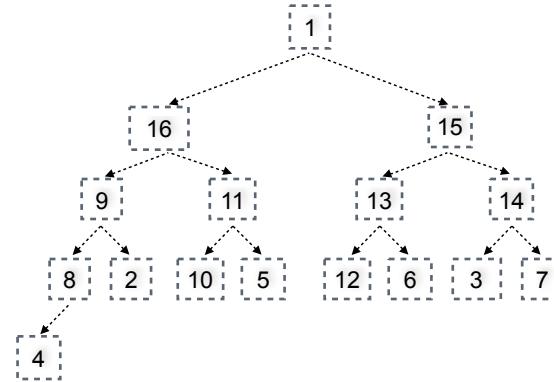
Progression de la percolation des éléments d'un tableau

Les 2 dernières étapes de la construction du monceau

... suite de la diapo précédente

0

16	11	15	09	10	13	14	08	02	01	05	12	06	03	07	04
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----



Percolation (suite) et résultat de buildHeap

Le tri par monceau divise en 2 parties le tableau à trier

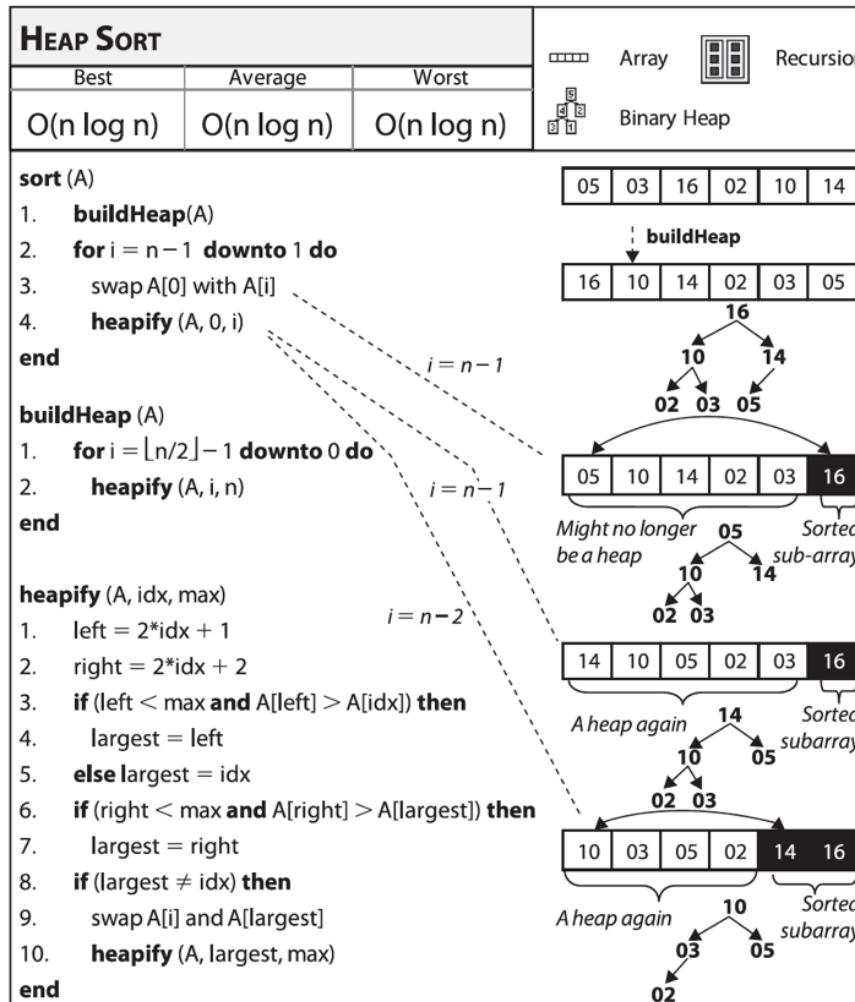
Le tri par monceau traite la collection en 2 parties distinctes, soit respectivement un monceau de taille m et un sous-tableau trié de taille $n-m$.

Au fur et à mesure qu'on itère sur un index i allant de $n-1$ à 1 , le sous-tableau grandit d'un élément $A[i,n]$ de droite à gauche en échangeant les éléments $A[0]$ avec $A[i]$; soit en retirant du monceau la plus grande valeur à chaque itération (e.g. tri par sélection).

Après avoir retiré la plus grande valeur et l'avoir placée dans la région triée du tableau, la nouvelle racine n'est pas nécessairement la plus grande valeur du nouveau monceau, il faut donc la percoler pour reconstruire un monceau dans la partie $A[0,i]$.

La région triée du tableau l'est nécessairement car à chaque itération on a la garantie d'y ajouter la plus grande valeur des valeurs restantes, soit la racine du monceau.

Le tri par monceau est un tri par sélection qui utilise un monceau pour trouver efficacement la prochaine plus grande valeur



Fiche technique du tri par monceau

Analyse du tri par monceau

Le tri par monceau évite les cas problématiques du tri rapide, restant en $O(n \log n)$ en pire cas comparativement à $O(n^2)$ pour le tri rapide. Cependant, en moyenne le tri rapide performe mieux malgré la même complexité.

La percolation est l'opération dominante du tri par monceau. Dans la construction du monceau, elle est appelée $\lfloor n/2 \rfloor - 1$ fois et durant le tri elle est appelée $n-1$ fois, pour un total de $\lfloor 3*n/2 \rfloor - 2$ fois.

La percolation est une opération récursive qui s'exécute un nombre fixe de fois jusqu'à la fin du monceau. À cause des propriétés structurales du monceau, la profondeur de l'arbre est toujours $\lfloor \log n \rfloor$ pour n éléments.

La performance résultante est donc bornée par $(\lfloor 3*n/2 \rfloor - 2)^* \lfloor \log n \rfloor$, qui est dans $O(n \log n)$.

Une version non-réursive du tri par monceau est légèrement plus rapide

n	Non-recursive Heap Sort	Recursive Heap Sort
16,384	0.0118	0.0112
32,768	0.0328	0.0323
65,536	0.0922	0.0945
131,072	0.2419	0.2508
262,144	0.5652	0.6117
524,288	1.0611	1.1413
1,048,576	2.0867	2.2669
2,097,152	4.9065	5.3249

Comparaison de versions réursive et non du tri par monceau sur 1 000 jeux de données de tailles variées

Code Python pour le tri par monceau

```
def triMonceau( tab ):
    n = len( tab )
    construitMonceau( tab, n )
    for i in range( n-1, 0, -1 ):
        swap( tab, 0, i )
        monceaurise( tab, 0, i )

def construitMonceau( tab, n ):
    for i in range( n//2 - 1, -1, -1 ):
        print( "in range i = ", i )
        monceaurise( tab, i, n )

def monceaurise( tab, i, max ):
    gauche = 2*i+1
    droite = gauche + 1
    if gauche < max and tab[gauche] > tab[i]:
        plusgrand = gauche
    else:
        plusgrand = i
    if droite < max and tab[droite] > tab[plusgrand]:
        plusgrand = droite
    if plusgrand != i:
        swap( tab, i, plusgrand )
    monceaurise( tab, plusgrand, max )
```

Tri par comptage

Trier en $O(n)$, est-ce possible ?

Un comptable est responsable des livres d'un petit commerce où chaque soir le commerçant enregistre les ventes de la journée sur un reçu avec le montant total et la date. Le commerçant sauve ses reçus dans une boîte qu'il donne au comptable à la fin de l'année. Le comptable doit d'abord s'assurer qu'il ne manque aucun reçu.

Comme vous pouvez l'imaginer, les reçus sont tous mêlés dans la boîte. Le comptable a 2 solutions :

1) trier les reçus par date et ensuite réviser les reçus triés pour identifier les dates manquantes, le cas échéant.

2) prendre un calendrier et marquer ce dernier aux dates correspondantes aux reçus qu'il sort aléatoirement de la boîte. À la fin, il n'a qu'à réviser le calendrier pour obtenir les dates manquantes.

Notez qu'à aucun moment la date de 2 reçus sont comparées.

S'il y a peu de reçus dans la boîte, la 2ème approche n'est peut-être pas nécessaire mais imaginez si on veut faire les comptes du commerce après plusieurs années; trier en comparant les dates devient alors beaucoup plus long que de marquer des calendriers !

Si vous devez trier n éléments mais qu'on vous dit qu'ils sont dans l'intervalle $[0, k-1]$, où k est beaucoup plus petit que n , alors vous pouvez tirer avantage de la situation et produire un tri linéaire, $O(n)$, qu'on appelle aussi le tri par comptage.

Les indices des seaux sont les valeurs des éléments à trier

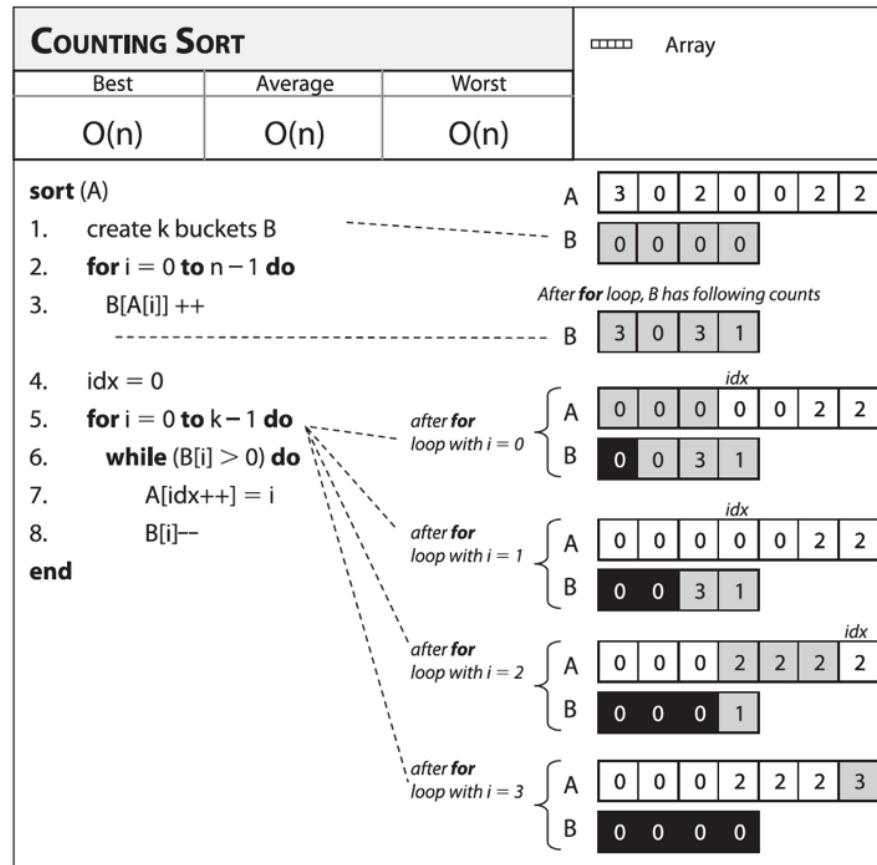
Ce tri ne nécessite pas de fonction de comparaison.

Dans l'exemple de la fiche, on a $n=7$ éléments à trier qui sont dans l'intervalle $[0..3]$.

On crée un « seau » sur l'intervalle $[0..3]$, donc un tableau B de 4 éléments, soit un par valeur.

On fait d'abord le dénombrement de chaque valeur des éléments à trier (partie du haut du pseudo-code).

Ensuite, pour chaque valeur (seau), on reconstruit le tableau à trier en insérant de gauche à droite le nombre de chaque valeur; p.e. on insère 3 éléments de valeur 0, aucun élément de la valeur 1, 3 éléments de la valeur 2 et 1 élément de la valeur 3, résultant dans le tableau A trié au bas de la fiche.



Fiche technique du tri par comptage

Code Python pour le tri par comptage

```
"""Trier n éléments dans l'intervalle 0 à k-1.  
"""  
  
def triComptage( tab, n, k):  
    seau = []  
    for i in range( k ):  
        seau.append( 0 )  
  
    for i in range( n ):  
        seau[tab[i]] += 1  
  
    idx = 0  
    for i in range( k ):  
        while seau[i] > 0:  
            seau[i] -= 1  
            tab[idx] = i  
            idx += 1
```

Le tri par comptage est bel et bien un tri en O(n) !

L'algorithme fait 2 passes sur tout le tableau. La première passe au travers de tous les éléments. Dans la deuxième passe, la boucle “while” est exécutée $B[i]$ fois pour chaque élément du « seau », $0 \leq i < k$. Donc, l’incrémentation d’une entrée est exécutée exactement n fois. Au total, on a un total de $2*n$, càd une performance en $O(n)$.

Tri avec seaux (bucket sort)

Le tri avec seaux fonctionne si 2 conditions sont respectées

I) Distribution uniforme

Les données doivent être distribuées uniformément.

Selon cette distribution, n seaux sont créés pour former des partitions égales.

2) Fonction de hashing ordonnée

Les seaux sont ordonnés tel que si $i < j$, alors les éléments insérés dans le seau i sont plus petit que les éléments dans le seau j .

Le tri avec seaux est une généralisation du tri par comptage

BUCKET SORT																											
Best	Average	Worst																									
O(n)	O(n)	O(n)																									
sort (A) <ol style="list-style-type: none"> 1. create n buckets B 2. for $i = 0$ to $n - 1$ do 3. $k = \text{hash}(A[i])$ 4. add $A[i]$ to the k^{th} bucket $B[k]$ 5. extract (B, A) end																											
			<p style="text-align: center;">use $\text{hash}(x) = \lfloor x / 3 \rfloor$</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">7</td> <td style="border: 1px solid black; padding: 2px;">5</td> <td style="border: 1px solid black; padding: 2px;">13</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">14</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{2,1}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{7,6}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	7	5	13	2	14	1	6	B	{2,1}	{5}	{7,6}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	7	5	13	2	14	1	6																				
B	{2,1}	{5}	{7,6}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				
			<p style="text-align: center;"><i>After for loop executes</i></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">13</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">14</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{2,1}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{7,6}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	1	2	13	2	14	1	6	B	{2,1}	{5}	{7,6}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	1	2	13	2	14	1	6																				
B	{2,1}	{5}	{7,6}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				
			<p style="text-align: center;"><i>After for loop with i = 0</i></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">5</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">14</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{1,2}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{7,6}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	1	2	5	2	14	1	6	B	{1,2}	{5}	{7,6}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	1	2	5	2	14	1	6																				
B	{1,2}	{5}	{7,6}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				
			<p style="text-align: center;"><i>After for loop with i = 1</i></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">5</td> <td style="border: 1px solid black; padding: 2px;">6</td> <td style="border: 1px solid black; padding: 2px;">7</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{1,2}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{6,7}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	1	2	5	6	7	1	6	B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	1	2	5	6	7	1	6																				
B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				
			<p style="text-align: center;"><i>After for loop with i = 2</i></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">5</td> <td style="border: 1px solid black; padding: 2px;">6</td> <td style="border: 1px solid black; padding: 2px;">7</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{1,2}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{6,7}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	1	2	5	6	7	1	6	B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	1	2	5	6	7	1	6																				
B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				
			<p style="text-align: center;"><i>After for loop with i = 3</i></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">5</td> <td style="border: 1px solid black; padding: 2px;">6</td> <td style="border: 1px solid black; padding: 2px;">7</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{1,2}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{6,7}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	1	2	5	6	7	1	6	B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	1	2	5	6	7	1	6																				
B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				
			<p style="text-align: center;"><i>After for loop with i = 4</i></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">5</td> <td style="border: 1px solid black; padding: 2px;">6</td> <td style="border: 1px solid black; padding: 2px;">7</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{1,2}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{6,7}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	1	2	5	6	7	1	6	B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	1	2	5	6	7	1	6																				
B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				
			<p style="text-align: center;"><i>After for loop with i = 5</i></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">A</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">2</td> <td style="border: 1px solid black; padding: 2px;">5</td> <td style="border: 1px solid black; padding: 2px;">6</td> <td style="border: 1px solid black; padding: 2px;">7</td> <td style="border: 1px solid black; padding: 2px;">1</td> <td style="border: 1px solid black; padding: 2px;">6</td> </tr> <tr> <td style="text-align: right;">B</td> <td style="border: 1px solid black; padding: 2px;">{1,2}</td> <td style="border: 1px solid black; padding: 2px;">{5}</td> <td style="border: 1px solid black; padding: 2px;">{6,7}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{13,14}</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> <td style="border: 1px solid black; padding: 2px;">{ }</td> </tr> <tr> <td></td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> <td style="text-align: center;">5</td> <td style="text-align: center;">6</td> </tr> </table>	A	1	2	5	6	7	1	6	B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }		0	1	2	3	4	5	6
A	1	2	5	6	7	1	6																				
B	{1,2}	{5}	{6,7}	{ }	{13,14}	{ }	{ }																				
	0	1	2	3	4	5	6																				

Fiche technique du tri avec seaux

Code Python pour le tri avec seaux

```
def triSeaux( tab, n ):
    #création des seaux
    seau = []
    for i in range( n ):
        seau.append( [] )
    #création des partitions
    for i in range( n ):
        k = hash( tab[i] )
        seau[k].append( tab[i] )
    #extraction des données
    extraire( seau, tab, n )

def extraire( seau, tab, n ):
    idx = 0
    for i in range( n ):
        seau[i].sort()
        for m in range( 0, len(seau[i]) ):
            tab[idx] = seau[i][m]
            idx += 1
```

Quelle méthode de tri ?

(sommaire)

Quelques éléments seulement = Tri par insertion

Presque déjà triés = Tri par insertion

Inquiétude de tomber dans un pire cas = Tri par monceau

Bonne performance en moyenne = Tri rapide

Éléments très diversifiés = Tri par « seaux »

Plus petit code possible = Tri par insertion

sort de Python plus rapide que n'importe quelle
implantation car invocation d'un code C