

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 comptage
8. Par "buckets"
9. 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

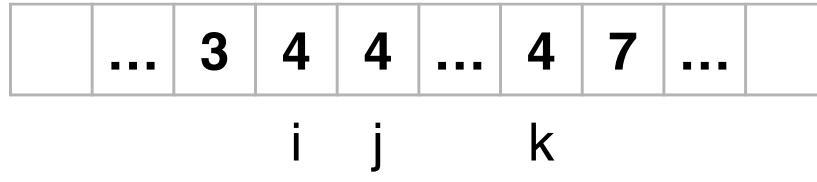
Trier une collection, A, consiste à ordonner les éléments tel que :

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 !

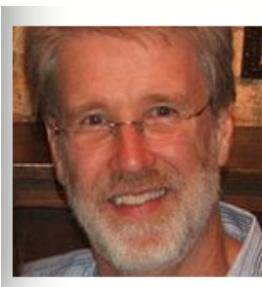
Est-ce que “A” > “a” ? Est-ce que “è” > “ê” ? Est-ce que “œ” > “o” ?

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 `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 `cmp` peut comparer une des valeurs, e.g. la clé, des enregistrements.

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 cmp dans une collection à trier, il arrive de vouloir garder cet ordre dans la collection triée.

Si $i < j$, alors les positions finales 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 la 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 cette collection sur la destination, un résultat possible d'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

<u>Destination</u>	<u>Airline</u>	<u>Flight</u>	<u>Sched</u>
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

<u>Destination</u>	<u>Airline</u>	<u>Flight</u>	<u>Sched</u>
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

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.

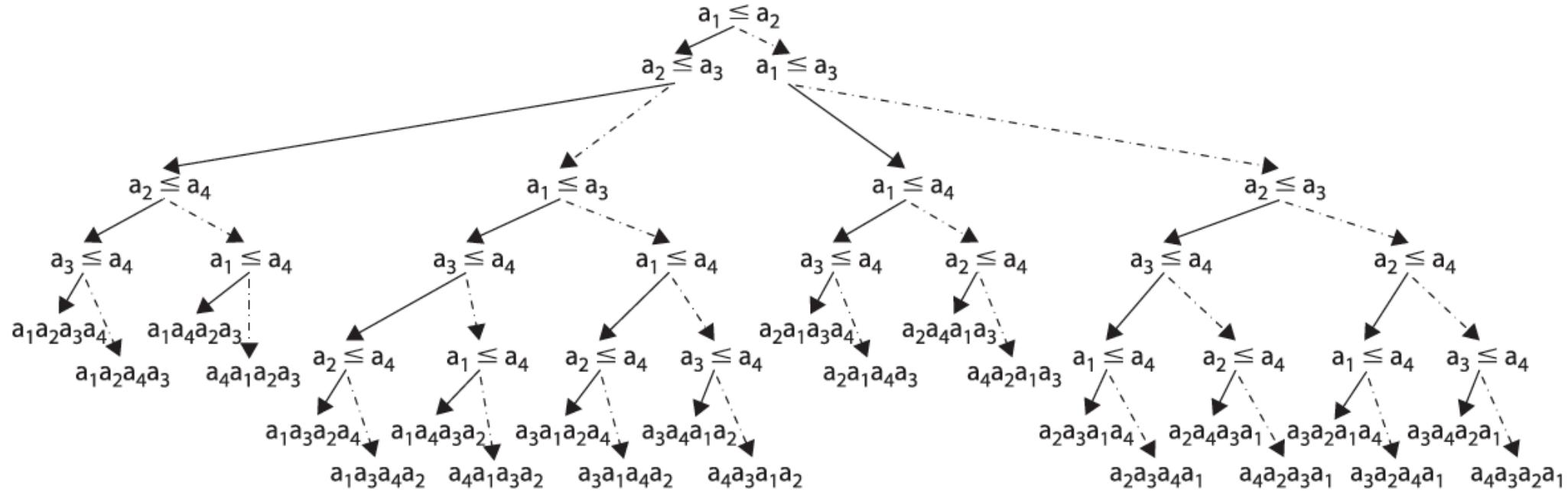
Voici la preuve :

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			
<i>Pseudocode description</i>			

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 :

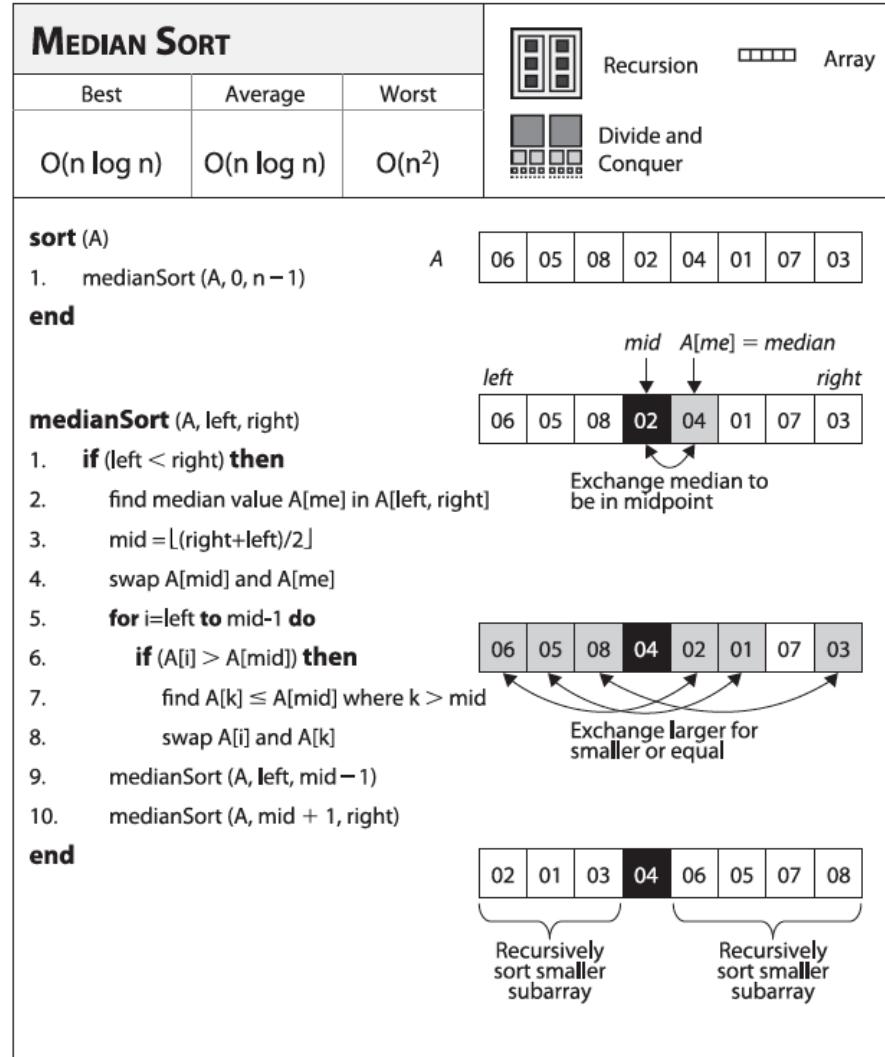
Plus la collection originale est triée et meilleures sont les performances du tri par insertion.

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

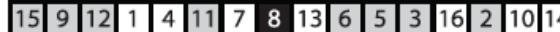
On divise jusqu'à des partitions de moins de 4 éléments

(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

parties gauche et droite
prêtes à trier

1c 

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

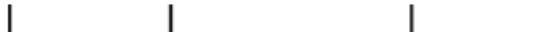
2a 

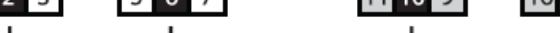
2b 

2c 

appels récursifs pour
trier des parties de plus
en plus petites

3a 

3b 

3c 

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

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 11 2 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	pivot	position médiane	

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

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

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

Des appels récursifs de partition jusqu'à ce que p = milieu du tableau détermine la valeur médiane

Partition ne trouve pas la valeur de la médiane !

La valeur renvoyée, 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, droite] jusqu'à ce qu'elle retourne p = 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^{ème} élément de la collection, ici k = 8

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

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

si k > p+1, alors le k^{ème} élément de tab est le (k-p)^{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 )
```