Informatique et bases de données
Chapitre 1 : Complexité et tris

Agnès Rico
Bérénice-Alexia Jocteur (berenice.alexia.jocteur@gmail.com)

12 janvier 2025

Lyon 1

1 Complexité

2 Recherche dichotomique

3 Les tris
- Le tri sélection
- Le tri insertion
- Différentes méthodes pour insérer
- Tri rapide

Lyon 1

Introduction

- Motivations : L'accès à des données dans des volumes de plus en plus grand nécessite leur organisation.
- Tris : On peut trier de façons différentes. Chaque méthode est plus ou moins longue. Complexité.
- Dans ce chapitre on trie des valeurs numérique dans l'ordre croissant. On ne s'occupe pas de l'optimisation de la mémoire.

Lyon 1

Définition (Complexité)

Nombre de transitions (opérations élémentaires) entre l'état initial et l'état final. En général on calcule la complexité par rapport à la taille du problème, par exemple :

- *complexité du calcul de la multiplication de 2 valeurs en fonction du nombres de chiffres représentant les valeurs ;*
- *complexité du calcul matricielle en fonction de la dimension de l'espace.*

C'est une notion théorique

- Quelle rapport avec le temps de calcul ?
- Qu'est-ce qu'une transition élémentaire ?
- Qu'est-ce que la taille du problème ?

Lyon 1

Notations

On utilise la notation de Landau $O()$.

Définition (Rappel)

Soient deux fonctions $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$, on dit que

1. $f(n) = O(g(n))$ *ssi* $\exists n_0 \in \mathbb{N}, \exists K \in \mathbb{R}$ *tel que* $\forall n \geq n_0$,
$$f(n) \leq Kg(n).$$
2. $f(n) = o(g(n))$ *ssi* $\forall \varepsilon \in \mathbb{R}^+$, $\exists n_0$ *tel que* $\forall n \geq n_0$,
$$f(n) \leq \varepsilon g(n).$$
3. $f(n) = \Theta(g(n))$ *ssi* $f(n) = O(g(n))$ *et* $g(n) = O((f(n))$ *ssi* $\exists n_0 \in \mathbb{N}, \exists K_1, K_2 \in \mathbb{R}$ *tel que* $\forall n \geq n_0$,
$$K_1 g(n) \leq f(n) \leq K_2 g(n).$$

$O(n)$ signifie que le nombre d'opérations est de l'ordre de n. Par exemple $2n + 48$ est en $O(n)$.

Lyon 1

Notations

On ne tient pas compte des facteurs constants, ce qui revient à dire qu'aller 2 ou 10 fois plus lentement n'est pas un problème.

- Raison théorique : pour aller 2 fois plus vite, il suffit de faire une machine qui effectue 2 opérations élémentaires à la fois.
- Raison réaliste : il faut 30 ans pour que la puissance du plus gros calculateur mondial (1,7GFlops par le CRAY 2 en 1985) devienne celle d'une machine de bureau (100GFlops en 2012) [1]. Donc la puissance de votre machine gommera la différence de complexité.

De manière générale, c'est la progression en fonction de n qui est la notion la plus intéressante.

1. Cf `http://fr.wikipedia.org/wiki/FLOPS`

Lyon 1

Exemple

Exemple (Calcul matriciel)

On considère une matrice de taille n, $M = (m_{ij})_{1 \leq i,j \leq n}$.
Quelle est la complexité du calcul du carré de la matrice : $M^2 = M \times M$?
Soit le résultat $R = (r_{i,j})_{1 \leq i,j \leq n}$.

$$r_{i,j} = \sum_{k=1}^{n} m_{ik} \times m_{kj}$$

- *pour chacun des termes de la matrice R, il faut faire une boucle de taille n.*
- *pour calculer tout les termes r_{ij} de la matrice R, il faut faire 2 boucles de taille n.*

Lyon 1

Exemple

Exemple (Calcul matriciel (suite))

Données : *$M = (m_{ij})_{0 \leq i < n}$ la matrice n sa taille*
Résultat : *$R = (r_{ij})$ la matrice M^2*
début

```
pour i=0 à n-1 faire
    pour j=0 à n-1 faire
        r_ij ← 0
        pour k=0 à n-1 faire
            r_ij = r_ij + m_ik × m_kj
```

Calcul de la complexité. On commence « au centre ».

- *Pour la boucle k : On a 3 opérations répétées n fois.*
- *Pour la boucle j : on a 1 opération et 1 boucle k que l'on répète n fois.*
  $n \times (1 + boucle_k) = n \times (1 + 3n) = 3n^2 + n.$
- *Pour la boucle i : on répète n fois la boucle j.*
  $n \times (3n^2 + n) = 3n^3 + n^2$ *opérations*

Le calcul de M^2 pour M une matrice de taille n est en $O(n^3)$.

Lyon 1

1 Complexité

2 Recherche dichotomique

3 Les tris
- Le tri sélection
- Le tri insertion
- Différentes méthodes pour insérer
- Tri rapide

Lyon 1

Recherche efficace

L'algorithme le plus simple pour une recherche dans un tableau consiste à passer tous les éléments en revue : la complexité est O(n) si n est la taille du tableau.

- Comment cherche-t-on dans un dictionnaire ?
- Que gagne-t-on ?

Définition (Diviser pour mieux régner)

« Diviser pour mieux régner » (divide and conquer), *est une méthode généralement efficace en informatique qui consiste à diviser le problème en 2 parties que l'on résout séparément puis (si nécessaire) qu'on fusionne.*

Lyon 1

Recherche efficace

Avez-vous joué au jeu « qui est-ce » ?

Définition (Recherche dichotomique)

Quand on recherche une valeur dans un tableau trié, il est possible de :

- *diviser le tableau en 2 parts égales ;*
- *trouver la part qui contient l'élément recherché ;*
- *appliquer la même méthode sur cette part.*

À chaque étape on élimine la moitié des cases du tableau, il ne reste plus qu'une case au bout de $\log_2(n)$ *étapes.*

Lyon 1

Exemple

Exemple (Dichotomie)

Par exemple on recherche la valeur 14 dans un tableau trié.

cases du tableau								*valeur*	*positions*
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*recherchée*	*possibles*
2	*5*	*6*	*9*	*10*	*15*	*16*	*20*	*14*	*0 à 7*
				10	*15*	*16*	*20*	$9 < 14$	*4 à 7*
				10	*15*			$15 \geq 14$	*4 à 5*
					15			$10 < 14$	*5 à 5*
								$15 \neq 14$	*aucune*

Donc la valeur ne se trouve pas dans le tableau. Si on devait l'insérer (voir plus loin) ce serait à la position 5.

Lyon 1

Dichotomie version itérative

Données : Tableau T, Taille n, Valeur recherchée v
Résultat : p la position où se trouve l'élément, i la position où il faudrait l'insérer pour rester trié
début

```
deb ← 0
fin ← n − 1
// On teste les cas d'arrêt immédiats
si T[fin] ≤ v alors
    si T[fin] == v alors return p = n, i = n;
    sinon return p = ∅, i = n;
si v ≤ T[deb] alors
    si T[deb] == v alors return p = 0, i = 0;
    sinon return p = ∅, i = 0;
// On cherche les deux valeurs du tableau qui encadrent v
tant que deb + 1 < fin faire
    milieu ← ⌊(fin+deb)/2⌋
    si v ≤ T[milieu] alors
        // La valeur est dans la première moitié
        fin ← milieu
    sinon
        // La valeur est dans la seconde moitié
        deb ← milieu
// On a trouvé les deux valeurs encadrant v
si v == T[fin] alors
    return p = fin, i = fin
sinon
    return p = ∅, i = fin
```


Preuve

Si l'un des deux premiers tests est vérifié, les valeurs renvoyées sont correctes.
Sinon, $T[0] < v \leq T[n-1]$. La boucle est faite pour garder cette propriété, c'est un invariant de boucle. On le montre par récurrence.

- Cela est vrai pour l'itération 0 (avant le début de la boucle) car on a fait les tests pour cela.
- Si la propriété est vraie à l'itération k :
  - si $v \leq T[millieu_k]$, à l'itération suivante $fin_{k+1} = milieu_k$ et deb ne change pas donc $T[deb_{k+1}] < v \leq T[fin_{k+1}]$;
  - sinon, $T[millieu_k] < v$, à l'itération suivante, fin n'a pas changé et $deb_{k+1} = milieu_k$ donc $T[deb_{k+1}] < v \leq T[fin_{k+1}]$.

À la sortie de la boucle, $deb + 1 = fin$ et $T[deb] < v \leq T[fin]$, donc le test $v == T[fin]$ permet de renvoyer les bonnes valeurs.

Complexité

La complexité de la méthode est $\mathcal{O}(\lceil \log_2 n \rceil)$ (n est la taille du tableau) car c'est le nombre d'itérations de la boucle. On le montre aussi par récurrence, soit $C(n)$ le nombre d'itérations de la boucle pour un tableau de taille n. Il est facile de voir que $C(n)$ est croissant.

- Pour $n = 1$ on n'entre pas dans la boucle, donc $C(n) \leq \log_2 n$.
- Si $\forall k < n\ C(k) < \log_2 k$, calculons $C(n)$:
- après une itération, on poursuit avec un tableau de taille $\lfloor \frac{n}{2} \rfloor$ ou $\lceil \frac{n}{2} \rceil$ selon le cas ;
- comme $C(k)$ est croissant, on ne considère que le cas du tableau le plus grand

$$\begin{aligned} C(n) &\leq C\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &\leq 1 + \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil \end{aligned}$$

- or soit $n = 2^k$ et dans ce cas, $\lceil \frac{n}{2} \rceil = 2^{k-1}$ et donc $1 + \lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = k = \lceil \log_2 n \rceil$
- soit $n \neq 2^k$ et dans ce cas, $\lceil \log_2 n \rceil = \lceil \log_2 n + 1 \rceil$ donc

$$\begin{aligned} 1 + \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil \right\rceil &= \left\lceil \log_2 \left\lceil \frac{n}{2} \right\rceil + 1 \right\rceil \\ &= \left\lceil \log_2 2 \times \left\lceil \frac{n}{2} \right\rceil \right\rceil \\ &\leq \lceil \log_2 n + 1 \rceil \\ &\leq \lceil \log_2 n \rceil \end{aligned}$$

- finalement dans tous les cas, $C(n) \leq \lceil \log_2 n \rceil$
- C.Q.F.D.

1 Complexité
2 Recherche dichotomique
3 Les tris
- Le tri sélection
- Le tri insertion
- Différentes méthodes pour insérer
- Tri rapide

Introduction

- Classement de n valeurs :
  - pour faciliter la recherche ;
  - pour créer des index ;
  - nous ne verrons que le tri de valeurs numériques, mais cela est possible dès qu'on a un pré-ordre total.
- Complexité :
  - en fonction de n ;
  - l'algorithme réputé le plus rapide n'est pas celui avec la meilleure complexité théorique, mais celui avec la meilleure complexité moyenne (et la plus petite constante) ;
  - en théorie, la complexité minimale est de $O(n \log_2 n)$.

Lyon 1

Complexité minimal

On compte le nombre de comparaisons :

- Un tri est une permutation d'un ensemble de n éléments.
- Chaque comparaison permet de choisir entre 2 ensembles de permutations possibles, on peut donc voir le tri comme un arbre binaire dont les comparaisons sont les noeuds et chaque feuille représente la permutation finale.
- La complexité est alors la hauteur de l'arbre *i.e.* le nombre maximal de comparaisons afin d'arriver à la bonne permutation.
- Un arbre binaire de hauteur h contient au plus 2^h feuilles différentes.
- Il y a $n!$ permutations différentes.
- Donc h est telle que $2^h > n!$, c'est à dire $h > \log_2(n!)$.
- D'après la formule de Stirling, $n! \sim \sqrt{2\pi n}\,(\frac{n}{e})^n$.
- Donc $\log_2 n! = \Theta(\log_2(n^n)) = \Theta(n \log_2(n))$.

Lyon 1

1 Complexité

2 Recherche dichotomique

3 Les tris
- Le tri sélection
- Le tri insertion
- Différentes méthodes pour insérer
- Tri rapide

Lyon 1

Le principe

À chaque étape, on cherche la plus petite valeur non triée pour la placer dans la bonne case.

- On cherche dans le tableau la plus petite valeur.
- On échange cette valeur avec celle qui se trouve en A[0].
- A partir de l'indice 1, on cherche la plus petite valeur du tableau.
- On échange cette valeur avec celle qui se trouve en A[1].
- ...

Exemple

On considère

8	5	6	9	4

On parcourt tout le tableau : 4 est la plus petite valeur, donc on échange 8 et 4

4	5	6	9	8

5 est la plus petite valeur, donc pas d'échange.
6 est la plus petite valeur, donc pas d'échange.
8 est la plus petite valeur, on échange 8 et 9.

4	5	6	8	9

Lyon 1

Algorithme

Données : Le tableau A, sa taille n

Résultat : Le tableau A, trié

début

```
pour i = 0 à n − 2 faire
    // recherche du minimum
    imin ← i pour j = i + 1 à n − 1 faire
        si A[j] < A[imin] alors imin ← j;
    // Échange des valeurs
    Temp ← A[imin]
    A[imin] ← A[i]
    A[i] ← Temp
```

Algorithme 1 : triSelection

Preuve

- Quel est l'invariant pour la boucle en i ? L'invariant est « les i premières cases du tableau sont celles du tableau trié »
- Prouvez l'invariant par récurrence.
  - Pour $i = 0$ un tableau à une case est toujours trié.
  - Hi $\Rightarrow$ Hi+1 : lorsqu'on fait $i + 1$ on rajoute le plus petit des éléments restant à la partie triée. On obtient $i + 1$ cases triées.

Complexité

La boucle j est répétée $n - 1 - i - 1 + 1 = n - 1 - i$; elle nécessite donc $2(n - 1 - i)$ opérations.
La boucle i est effectuée $n - 1$ fois, elle nécessite 4 affectations et une boucle j.
Le nombre d'opérations de la boucle i est donc :

$$\begin{aligned}\sum_{i=0}^{n-2} 2(n-1-i)+4 &= \sum_{i=0}^{n-2} 2n+2-2i \\ &= (2n+2)(n-1) - 2\frac{n(n-1)}{2} \\ &= n^2 + n - 2 \text{ opérations.}\end{aligned}$$

Le tri est donc en $O(n^2)$.
Remarque :
Pour le tri sélection que le tableau soit trié ou non au départ ne change pas le nombre de comparaisons à effectuer.

1 Complexité

2 Recherche dichotomique

3 Les tris
- Le tri sélection
- Le tri insertion
- Différentes méthodes pour insérer
- Tri rapide

Principe

On gère d'un coté les valeurs triées et de l'autre les valeurs à trier. À chaque étape, on insère une nouvelle valeur dans les valeurs triées.

Exemple

partie triée	*partie à trier*
[8]	*5 6 9 4*
on veut insérer 5 dans la partie triée	
[5, 8]	*6 9 4*
on veut insérer 6 dans la partie triée	
[5, 6, 8]	*9 4*
on veut insérer 9 dans la partie triée	
[5, 6, 8, 9]	*4*
on veut insérer 4 dans la partie triée	
[4, 5, 6, 8, 9]	

Lyon 1

Algorithme

Données : Un tableau A, sa taille n
Résultat : A le tableau trié
début
  pour $i = 1$ *à* $n - 1$ **faire**
    Insérer $A[i]$ parmi les i premiers éléments

Algorithme 2 : triInsertion

Lyon 1

Preuve

On considère A un tableau d'entiers de n cases indicées de 0 à $n - 1$.

- HR_0 : On considère que A[0] est trié. On dit alors que le tableau est trié jusqu'au rang 1
- $HR_{i-1} \Rightarrow HR_i$
  On suppose le tableau trié jusqu'au rang i-1.
  L'élément A[i] va être inséré parmi les éléments A[0]... A[i-1] de façon à ce que le tableau soit trié jusqu'au rang i.
- L'opération $HR_{i-1} \Rightarrow HR_i$ se répète jusqu'à ce que tous les éléments soient triés.

Lyon 1

Insertion

Pour insérer $A[i]$ dans le tableau $A[0..i-1]$ il faut

- *trouver où insérer*, c'est à dire trouver k, $0 \leq k \leq i - 1$ tel que $A[k] \leq A[i] < A[k+1]$.
- *faire de la place*, c'est à dire décaler toutes les cases $A[k+1], \ldots, A[i-1]$

Données : A un tableau, i la case à insérer, $A[0] \leq A[1] \leq \cdots \leq A[i-1]$
Résultat : Tableau A trié de 0 à i c'est à dire $A[0] \leq A[1] \leq \cdots \leq A[i]$
début
  $j \leftarrow i - 1$
  $v \leftarrow A[i]$
  // Recherche et décalage des valeurs
  tant que $v < A[j]$ *et* $0 \leq j$ **faire**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  // Placement de A[i]
  $A[j+1] \leftarrow v$

Algorithme 3 : insertionTableauTrie

Lyon 1

Algorithme entier

Données : Un tableau A, sa taille n
Résultat : A le tableau trié
début
  pour $i = 1$ *à* $n - 1$ **faire**
    // Insérer $A[i]$ parmi les i premiers éléments
    $j \leftarrow i - 1$
    $v \leftarrow A[i]$
    // Recherche et décalage des valeurs
    tant que $v < A[j]$ *et* $0 \leq j$ **faire**
      $A[j + 1] \leftarrow A[j]$
      $j \leftarrow j - 1$
    // Placement de A[i]
    $A[j + 1] \leftarrow v$

Algorithme 4 : triInsertion

Lyon 1

Complexité

Quel est la complexité du tri dans le pire cas ? Le pire cas est atteint lorsque le tableau est trié à l'envers.
Complexité pour insérer i : 3 + 4i
Pour le tri ;
Pour i=1, 7
Pour i=2, 11
Pour i=n-1, 4n - 1
On a donc
$\sum_{i=1}^{n-1} 3+4i = \sum_{i=1}^{n-1} 3+\sum_{i=1}^{n-1} 4i = 3(n-1)+4\frac{n(n-1)}{2} = 2n^2+n-3 = O(n^2)$

Lyon 1

Amélioration

- À chaque étape de l'insertion, il est nécessaire de faire 2 tests : $v < A[j]$ et $0 \leq j$. Le second test ne sert qu'à éviter de sortir du tableau lorsqu'on insère un élément trop petit.
  Pour éviter cela, il est possible d'utiliser une sentinelle
  - on ajoute une case au tableau
  - dans la première case ($A[0]$) on place la valeur la plus petite possible : Integer.MIN_VALUE ou DOUBLE.MIN_VALUE
  - Le test $v < A[j]$ est donc forcément vrai et permet d'éviter de sortir du tableau.
- Il est possible de chercher la position d'insertion en utilisant la recherche dichotomique.
  Que gagne-t-on ? Est-ce vraiment intéressant ?

Lyon 1

1 Complexité
2 Recherche dichotomique
3 Les tris
- Le tri sélection
- Le tri insertion
- Différentes méthodes pour insérer
- Tri rapide

Lyon 1

Diviser pour mieux régner

Peut-on appliquer le principe de « *divide and conquer* » au tri ?
Rappel

Définition (Diviser pour mieux régner)

« Diviser pour mieux régner » (divide and conquer*), est une méthode généralement efficace en informatique qui consiste à diviser le problème en 2 parties que l'on résout séparément puis (si nécessaire) qu'on fusionne.*

- Comment diviser ?
- Comment fusionner ?

2 possibilités

Il y a conservation de l... la difficulté

- Si on divise simplement en 2 parts égales
  - la partition est simple (on coupe au milieu) ;
  - les deux parts sont égales ;
  - la fusion est plus difficile (il faut intercaler les éléments des 2 parties).
- Si on divise en 2 paquets ordonnés (les valeurs du premier paquet sont forcément inférieures à celles du second) :
  - la partition est difficile ;
  - il est quasi impossible d'obtenir des paquets de même taille rapidement ;
  - la fusion est simple (on ajoutes les paquets l'un à la suite de l'autre).

Les deux méthodes existent :

- la première est le tri fusion (qu'on ne verra pas), il a une complexité $\mathcal{O}n \log_2 n$
- la seconde est le tri rapide, sa complexité est moins bonne ... mais il est en moyenne plus rapide.

Principe

Le tri rapide est basé sur l'opération de partition
On considère une partie du tableau entre les indices l et r,

A[l]										A[r]

avant la partition les valeurs ne sont pas triées

9	...	5	...	1	14	...	20	...	7	8

après la partition les valeurs à droites sont supérieures à celles à gauche

7	...	5	...	1	8	...	20	...	9	14

pour cela on utilise un pivot

valeurs ≤ 8	8	valeurs ≥ 8

Premier algorithme

On suppose que $partition(A, l, r)$ est une fonction qui effectue la partition de A entre l et r et retourne la position i du pivot.
Le tri rapide est beaucoup plus simple à faire de manière récursive :

Données : Un tableau A, deux entiers $l \leq r$
Résultat : A est trié entre l et r : $A[l] \leq A[l+1] \leq \cdots \leq A[r]$
début
  si $l < r$ **alors**
    $i \leftarrow partition(A, l, r)$
    $triRapide(A, l, i-1)$
    $triRapide(A, i+1, r)$

Algorithme 5 : triRapide(A, l,r)

Le tri complet de A s'effectue en appelant $trirapide(A, 0, n-1)$.

Preuve

L'hypothèse de récurrence est « *triRapide* trie tout les tableau de taille $k \leq n$ »

- vrai pour $n = 1$ ou $n = 2$ car partition fait son travail
- si c'est le cas, et si $r - l = n + 1$, $triRapide(A, l, i - 1)$ et $triRapide(A, i + 1, r)$ sont appelées sur des tableaux de taille $\leq n$ donc elles trient. La fusion est déjà faite.
- C.Q.F.D.

Fonction de partition

Principe

1. On cherche à partir de l'indice l le plus petit indice i tel que $A[i] \geq p$.
2. On cherche à partir de l'indice r le plus grand indice j tel que $A[j] \leq p$.
3. On échange $A[i]$ et $A[j]$.
4. On recommence 1, 2 et 3 jusqu'à ce que $i \geq j$.
5. On échange $A[r]$ et $A[i]$.

Exemple

7	2	5	3	9	6
$i\uparrow$			$j\uparrow$	$j\uparrow$	
$7 \geq 6 arret$			$3 < 6 arret$	$9 \geq 6$	

On permute 7 et 3 car $i < j$.

	3	2	5	7	9	6
	$i\uparrow$	$i\uparrow$	$i,j\uparrow$	$i,j\uparrow$		
pour i	$3 < 6$	$2 < 6$	$5 < 6$	$7 \geq 6 arret$		
pour j			$5 < 6 arret$	$7 \geq 6$		

Comme $i \geq j$ tout le tableau a été vu et la partition est terminée
On échange le pivot et $A[i]$ qui vaut 7.

3	2	5	6	9	7

Remarques

- La partition ne trie pas à la fin :
  - les parties à gauche et à droite du pivot ne sont pas triées ;
  - les éléments de la partie gauche sont inférieurs au pivot, les éléments de la partie droite sont supérieurs.
- Il faut vérifier que les boucles sur i et j ne sortent pas de la partie de tableau considérée ($i \leq r$ et $j \geq l$).
  - la boucle sur i ne peut pas dépasser le pivot car si $i = r$, $A[i] \geq A[r]$;
  - si $l > 0$ (si le tableau considéré n'est pas à gauche), il y a forcement une valeur inférieure à gauche ;
  - seul le cas ou $l = 0$ peut causer un problème, pour éviter cela, on place une sentinelle en $A[0]$ (la plus petite valeur possible Integer.MIN_VALUE).

Algorithme de la partition

Données : A un tableau, deux entiers $l \leq r$

Résultat : A est divisé entre l et r en fonction du pivot $A[r]$ de plus la fonction retourne i la position du pivot

début
- $p \leftarrow A[r]$
- $i \leftarrow l$
- $j \leftarrow r - 1$
- **tant que** $i < j$ **faire**
  - **tant que** $A[i] < p$ **faire**
    - $i \leftarrow i + 1$
  - **tant que** $A[j] \geq p$ **faire**
    - $j \leftarrow j - 1$
  - **si** $i < j$ **alors**
    - échanger $A[i]$ et $A[j]$
- **si** $A[i] > A[r]$ **alors**
  - échanger $A[r]$ et $A[i]$
- **return** i

Algorithme 6 : partition(A, l, r)

Retour sur l'algorithme

Données : Un tableau A, sa taille n

Résultat : A est trié

début
- Ajouter la sentinelle à A
- triRapide(A, 1, n)
- Retirer la sentinelle de A

Algorithme 7 : triRapide(A)

Données : Un tableau A, deux entiers $l \leq r$

Résultat : A est trié entre l et r : $A[l] \leq A[l+1] \leq \cdots \leq A[r]$

début
- **si** $l < r$ **alors**
  - $i \leftarrow partition(A, l, r)$
  - $triRapide(A, l, i-1)$
  - $triRapide(A, i+1, r)$

Algorithme 8 : triRapide(A, l,r)

Complexité

- Si le tableau est trié dans l'ordre ou l'ordre inverse :
  - après chaque partition, l'une des parties est de taille maximum $(l - r - 1)$, l'autre est vide ;
  - chaque calcul de partition demande $l - r$ opération ;
  - l'ensemble demande $n + n - 1 + \cdots + 1 = O(n^2)$ opérations.
- Mais le nombre de tableau triés est faible, en moyenne les parties sont de tailles équivalentes et la complexité est de $O(n \log n)$.

Cependant, il arrive souvent qu'on tente de trier un tableau entièrement ou partiellement trié. Donc dans la pratique on tire au sort des cases que l'on échange afin de corriger ces cas.