Complexité

Tris

Quelques éléments

Rappels

- Algorithme (en général)

  Description de la **résolution** d'un **problème** sous forme d'une **suite finie d'opérations élémentaires**.

- Algorithme (en informatique)

  Description du **traitement** d'une **donnée**

  pour obtenir un **résultat**.

  Un **traitement** définit une **relation** entre une **donnée `d`** et un **résultat `r`**.

- Programme

  Réalisation d'un algorithme dans un **langage** qu'un **ordinateur** est capable **d'exécuter**.

Analyse de la complexité

L'exécution d'un programme nécessite l'utilisation des ressources de l'ordinateur :

- Temps de calcul pour exécuter les opérations
- Occupation de la mémoire pour contenir et manipuler le programme et ses données.

L'objet de l'analyse de la complexité est de quantifier les deux grandeurs physiques **temps d'exécution** et **place mémoire**, dans le but de comparer entre eux diffèrents algorithmes qui résolvent le même problème.

Analyse de la complexité

Il s'agit d'abord de déterminer quelle mesure utiliser pour calculer ces deux quantités : pour un programme donné sur une machine donnée, on peut par exemple exprimer la complexité en temps (resp. en place) par le nombre de cycles machine (resp. mots mémoire) utilisés lors de l'exécution du programme, en comptant :

- **pour le temps** : le nombre d'opérations effectuées par le programme et le nombre de cycles nécessités par chaque opération.
- **pour la place** : le nombre d'instructions et le nombre de données du programme, avec le nombre de mots mémoire nécessaires pour stocker chacune d'entre elles, ainsi que le nombre de mots mémoire supplémentaires pour la manipulation des données

Type d’énoncé que l’on souhaite produire :

« *Sur toute machine, et quel que soit le langage de programmation, l’algorithme A1 sera* ***meilleur*** *que l’algorithme A2 pour les données de* ***grande taille*** »

Ou encore

« *L’algorithme A est* ***optimal*** *en nombre de comparaisons pour résoudre le problème Q* »

Complexité algorithmique
Mesure de la complexité en temps

Le temps d'exécution d'un programme dépend des facteurs suivants:

1. Les données entrant dans le programme.
2. La qualité du code généré par le compilateur pour la création du programme objet.
3. La nature et la vitesse d'exécution des instructions du processeur utilisé pour l'exécution du programme.
4. La complexité "algorithmique" du programme.

Mettre en évidence une ou plusieurs opérations fondamentales : le temps d'exécution est toujours proportionnel au nombre de ces opérations.

Exemples:

- **recherche** d'un élément dans une liste en mémoire centrale : nombre de comparaisons entre cet élément et les entrées (éléments) dans la liste
- **trier** une liste d'éléments : nombre de comparaisons et nombre de déplacements
- **multiplication** de matrices : nombre de multiplications et nombre d'additions

Deux points importants:

1. Degré de précision de l’analyse lié au nombre d’opérations fondamentales.
   Analyse très précise → **toutes** les opération du programme sont fondamentales.

2. Temps d’exécution proportionnel à la mesure choisie.
   ➔ Classes d'algorithmes.

Calcul de la Complexité

Il s'agit de compter le nombre d'opérations de chaque type

a. **Lorsque les** opérations sont dans une séquence d'instructions, leurs nombres s'ajoutent

b. **Pour les branchements** conditionnels, il est en général difficile de déterminer quelle branche de la condition sera exécutée, et donc quelles sont les opérations à compter. Il faut majorer:

```
P(if C then I1 else I2)<= P(C)+Max(P(I1),P(I2))
```

c. **pour les boucles**, le nombre d'opérations dans la boucle est $\sum_i P(i)$, où i est l'indice de la boucle, et P(i) le nombre d'opérations fondamentales lors de l'exécution de la $i^{éme}$ boucle.

Pb : Évaluation du nombre d'itérations.

d. **pour les appels de procédures** :

- Pas de procédure récursive : ordonner les procédures
- Procédures récursives:

  Compter le nombre d'opérations fondamentales

  ➔ Relations de récurrence à résoudre.

Le nombre d'opérations T(n) dans l'appel de la procédure avec un argument de taille n s'écrit, selon la récursivité, en fonction de divers T(k), pour k<n.

Exemple : calcul de la factorielle d'un entier positif:

```
fonction fact(n: ENTIER) = r : ENTIER
si  n = 0 alors r ← 1
           sinon r ← n * fact(n-1)
```

Si l'on choisit comme opération fondamentale la multiplication de deux entiers, on obtient clairement que le nombre T(n) d'opérations fondamentales vérifie:

```
{T(n) = T(n-1) + 1, pour n>1 et T(1) =1}
```

d'où par une résolution directe de cette récurrence très simple:

```
T(n) = n.
```

Exemple: algorithme de recherche séquentielle d'un élément dans un tableau `t` de taille `n`:

```
chercher(x: Élément) = r : entier
(1)  i ← 1
(2)  tantque (i<n) ∧ (x ≠ t[i])
(3)    faire  i ← i + 1
(4)  fintantque
(5)  si (x = t[i])alors r ← i
      sinon r ← 0 finsi
```

Les éléments significatifs pour analyser la complexité sont les suivants:

- le nombre d'itérations,
- le nombre de comparaisons par itérations

Remarques:

- L'instruction `i ← i + 1` de la ligne `(3)` est dépendante de la programmation : elle disparaît si on programme l'algorithme différemment, avec une boucle **`for.`** Il en est de même de la comparaison `(i<n)` de la ligne `(2)`. On voit bien qu'il ne faut pas prendre en compte ces opérations pour l'évaluation de l'algorithme.

- De plus ces instructions sont dépendantes de la structure de données choisie pour représenter la liste d'éléments. Avec une liste chaînée on aurait d'autres instructions: manipulations de pointeurs, test si un pointeur est égal à **`nil`**.

- Les opérations significatives dans cet algorithme sont donc les comparaisons de **`x`** avec les éléments de la liste. Il y en a une par itération.

Le nombre d'itérations est égal à **`n`** si **`x`** n'est pas dans la liste, et à **`j`**, index de la première occurrence de **`x`**, si **`x`** est dans la liste.

On démontre avec **invariant** et **condition d'arrêt**.

Cet algorithme effectue:

- **`j`** comparaisons si **`j`** est l'indice de la première occurrence de **`x`**
- **`n`** comparaisons si **`x`** n'est pas dans la liste.

Trois points:

- choix des opérations a priori,
- complexité dépend de la taille des données (ici **`n`**),
- pour une taille fixée la complexité dépend de la configuration des données (ici de **`1`** à **`n`**).

Complexité en moyenne et au pire

Mesure de taille sur les données

Configuration de la donnée

Soit $\mathtt{D}_{\mathtt{n}}$ l'ensemble des données de taille `n`.

Notons $\mathtt{coût}_{\mathtt{A}}\mathtt{(d)}$ la complexité en temps de l'algorithme `A` sur la donnée `d`.

On s'intéresse à plusieurs mesures :

a. la complexité dans le meilleur des cas

$$\mathrm{Min}_A(n) = \min\{\mathrm{coût}_A(d);\ d \in D_n\}$$

b. la complexité dans le pire des cas

$$\mathrm{Max}_A(n) = \max\{\mathrm{coût}_A(d);\ d \in D_n\}$$

c. la complexité en moyenne

$$\mathrm{Moy}_A(n) = \sum_{d \in D_n} p(d).\mathrm{coût}_A(d)$$

où p(d) est la probabilité que l'on ait la donnée d en entrée de l'algorithme.

Si toutes les données sont équiprobables alors la complexité en moyenne s'exprime simplement en fonction du nombre $\|D_n\|$.

$$Moy_A(n) = 1/ \|D_n\| \,.\, \sum_{d \in D_n} coût_A(d)$$

En général, les données n'ont pas toutes la même probabilité et la définition de la complexité moyenne nécessite l'introduction d'un modèle probabiliste lié au problème.

$$Min_A(n) \leq Moy_A(n) \leq Max_A(n)$$

Exemple A : Multiplication de matrices carrées

```
type matrice = array [1..n, 1..n] of integer;
procedure MULMAT (a,b : matrice ; var c :matrice);
   var i, j, k : integer;
   begin
        for i := 1 to n do
            for j := 1 to n do
                begin c[i,j] := 0;
                    for k := 1 to n do
                        c[i,j] := c[i,j] + a[i,k]*b[k,j]
                end;
    end MULMAT;
```

La complexité de l'algorithme MULMAT, comptée en nombre de multiplications de réels ne dépend que de la taille des matrices:

$$\mathrm{Min}(n) = \mathrm{Moy}(n) = \mathrm{Max}(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = n^3$$

Exemple B : Recherche séquentielle

On cherche à déterminer la complexité, en nombre de comparaisons, de l'algorithme de recherche séquentielle. On a

$$\mathrm{Min}(n)=1 \text{ et } \mathrm{Max}(n)=n$$

Calcul de `Moy(n)` : probabilités sur L et X

- soit q la probabilité que X soit dans L
- on suppose que si X est dans L, toutes les places sont équiprobables

D_n^i pour 1<= i <= n : ens. des données où X apparaît à la $i^{ème}$ place

D_n^0 : ens. des données où X est absent

$p(D_n^i) = q/n$ et $p(D_n^0) = 1 - q$

On a $coût(D_n^i) = i$ et $coût(D_n^0) = n$

$$Moy(n) = \sum_{i=0}^{n} p(D_n^i) \cdot coût(D_n^i) = (1-q) \cdot n + \sum_{i=1}^{n} q/n.i$$

$$= (1-q) \cdot n + q/2 \cdot (n+1)$$

Si on sait que X est dans la liste, alors on a $q = 1$

$$Moy(n) = (n + 1) /2$$

Si X a une chance sur deux d'être dans la liste, on a $q = ½$

$$Moy(n) = n/2 + (n + 1)/4 = (3n + 1)/4$$

Comparaisons de deux algorithmes

- La complexité d'un algorithme est fonction de la taille des données.
- Une simple approximation de la fonction de complexité suffit.
- Constantes additives et multiplicatives peu importantes.

A_1 :complexité $M_1(n) = n^2$

A_2 :complexité $M_2(n) = 2n$

A_2 meilleur que A_1 pour presque tous les n ($n>1$).

Idem pour $M_1(n) = 3n^2$ et $M_2(n) = 25n$

Ordre de grandeur asymptotique

Analyse de la complexité de $M_A(n)$: chercher une échelle de comparaison, une fonction avec une rapidité de croissance voisine.

Notation: étant données deux fonctions f et g de N dans $\mathbf{R}^+$,

$$f = 0(g) \text{ ssi } \exists\ c \in \mathbf{R}^+, \exists\ n_0 \in N \text{ tels que}$$
$$\forall\ N > n_0, f(n) \leq c.g(n)$$

Signification:

- Ordre de grandeur asymptotique de f est inférieur ou égal à celui de g
- f est dominée *asymptotiquement* par g.

Ex : $2n = 0(n^2)$, et $2n = 0(n)$

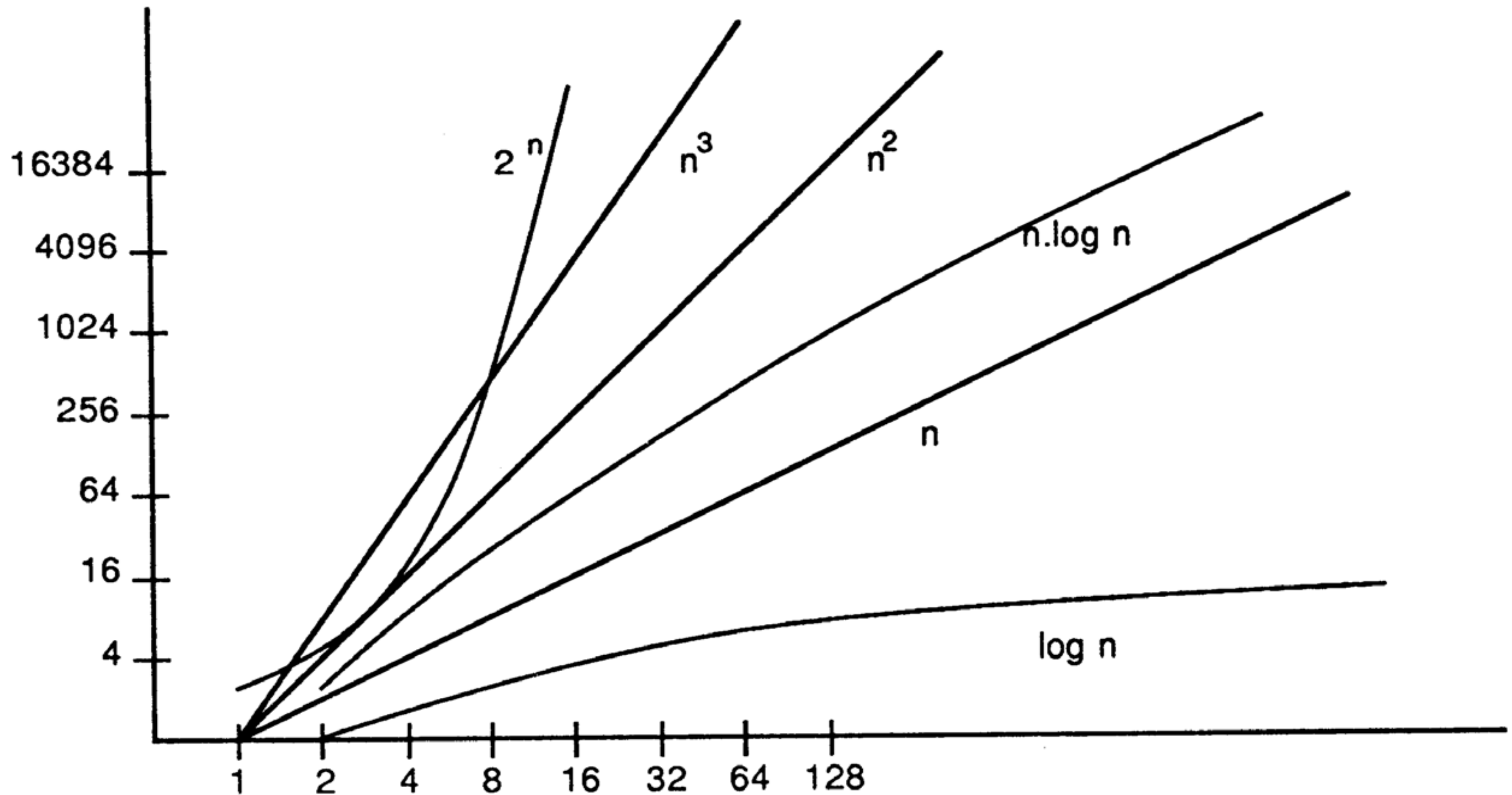
Cette notation donne un ordre de grandeur asymptotique de `f`.
Elle n'est pas suffisante pour comparer les performances des diffèrents algorithmes.
Elle fournit un majorant.
Il faut connaître le plus petit majorant : ordre de grandeur est exactement `h(n)`.

Nouvelle notation:

$f=\Theta(g)$ ssi $f=O(g)$ et $g=O(f)$
i.e. il existe c et d réels positifs et n0 entier tels que
$\forall n>n0,\ d.g(n) \leq f(n) \leq c.g(n)$

On dit que f et g ont même ordre de *grandeur asymptotique.*

Rapidité de croissance de fonctions usuelles



Complexité

Taille

	1	log_2N	N	Nlog2N	N^2	N^3	2^N
$N = 10^2$	~ 1 µs	~ 6.6 µs	0.1 ms	0.66 ms	10 ms	1 s	$4.\ 10^7$ milliards d'années
$N = 10^3$	~ 1 µs	~ 9.9 µs	1 ms	9.9 ms	1 s	16.6 mn	∞
$N = 10^4$	~ 1 µs	~ 13.3 µs	10 ms	0.13 s	1.5 mn	11.5 j	∞
$N = 10^5$	~ 1 µs	~ 16.6 µs	0.1 s	1.64 s	2.7 h	31,7 a	∞
$N = 10^6$	~ 1 µs	~ 19.9 µs	1 s	19.9 s	11.5 j	31.7 milliers d'années	∞

Estimation du temps d'exécution pour différentes tailles des données du problème

(*sur un ordinateur pouvant effectuer 10^6 opérations par seconde*)

Complexité

Temps de calcul	1	$\log_2 N$	N	Nlog2N	N^2	N^3	2^N
1 s	∞	∞	10^6	$6.3\ 10^4$	10^3	10^2	19
1 mn	∞	∞	$6\ 10^7$	$2.8\ 10^6$	$7\ 10^3$	$4\ 10^2$	25
1 h	∞	∞	$36\ 10^8$	$1.3\ 10^8$	$6\ 10^4$	$15\ 10^2$	31
1 jour	∞	∞	$8.6 10^{10}$	$2.7\ 10^9$	$2.9\ 10^5$	$44\ 10^2$	36

Estimation de la taille maximale du problème que l'on peut résoudre en un temps donné
(*sur un ordinateur pouvant effectuer 10^6 opérations par seconde*)

Algorithmes utilisables

Ce sont ceux qui s'exécutent en temps:

- **constant** : cas de certaines méthodes de hachage pour la complexité en moyenne.
- **logarithmique** : par ex. recherche dichotomique ou opérations sur les arbres binaires de recherche.
- **linéaire** : algorithmes qui doivent connaître toutes les données pour résoudre le problème tel que la recherche d'un élément dans un tableau non trié.
- **n.log(n)** : les "bons" algorithmes de tri.

LE TRI

- Méthodes de tri très importantes en pratique
- Beaucoup d'algorithmes.
- Spécification
  - **Informelle** : la donnée est une **liste de n éléments**; à chaque élément est associé une **clé** dont la valeur appartient à un ensemble **totalement ordonné**
  - le résultat est une liste dont les éléments sont une **permutation** des éléments de la liste d'origine telles que les **valeurs des clés soient croissantes**.
  - **Formelle** :

P(td) = td : tableau[T], T muni d'un ordre total.

Q(td, tr) = tr : tableau[T] $\wedge$
perm(tr, td) $\wedge$
$\forall$i, bi(tr) $\leq$ i < bs(tr) => tr[i] $\leq$ tr[i+1]

Algorithmes guidés par la donnée

- Trois étapes :
  1. Découpage du tableau en sous-tableaux
  2. Tri de chacun des sous-tableaux
  3. Interclassement des sous-tableaux triés

- Deux sous-tableaux de taille équivalente
  - Tris par interclassement de Von Neumann

- Deux sous-tableaux de taille 1 et n-1
  - Tris par insertion séquentielle ou dichotomique

Algorithmes guidés par le résultat

- Trois étapes :
  - Découpage du tableau en sous-tableaux tels que tous les éléments du sous-tableau i soient supérieurs à ceux du sous-tableau i-1 et inférieurs à ceux du sous-tableau i+1.
  - Tri de chacun des sous-tableaux
  - Concaténation des sous-tableaux triés
- Deux sous-tableaux de taille équivalente
  - Tri rapide de Hoare
- Deux sous-tableaux de taille 1 et n-1
  - Tris par sélection
    ordinaire, bulle, arborescente, par tas.

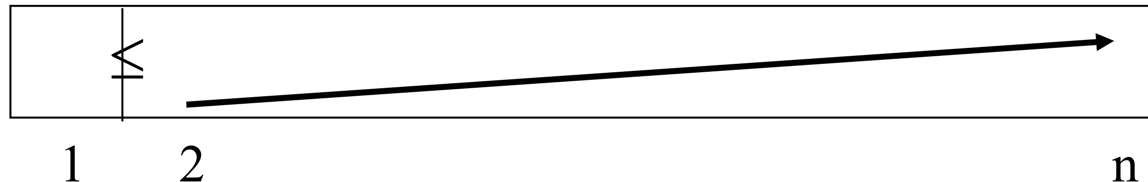
Tri par sélection

donnée:	|101	115	30	63	47	20
sélection:						**20**
placement	20	|115	30	63	47	101
sélection			**30**			
placement	20	30	|115	63	47	101
sélection					**47**	
placement	20	30	47	|63	115	101
sélection				**63**		
placement	20	30	47	63	|115	101
sélection						**101**
placement	20	30	47	63	101	115

Principe :

• découpage en deux sous-tableaux, de taille 1 et n-1; le premier contient le plus petit de tous les éléments.

• Tri du sous-tableau de taille n-1



• On recommence jusqu'à ce que le sous tableau soit réduit à un élément

Sélection ordinaire : version récursive

```
triselect(t:tableau, i:entier)
    si i < n alors //la tranche contient au moins 2 éléments
    /* recherche séquentielle du minimum entre les
    indices i et n */
        j ← i;
        pour k dans [(i+1)..n] faire
        si t[k]<t[j] alors j ← k finsi
        finpour
        // placement du minimum
         t[j]↔ t[i]
        // tri de la fin du tableau
         triselect(t, i+1)
     finsi
fin triselect
```

Sélection ordinaire : version itérative

```
/* on suppose t et n donnés */
pour i dans [1..(n-1)] faire
      j ← i
      pour k dans [(i+1)..n] faire
            si t[k]<t[j]
                  alors j ← k
            finsi
      finpour
      t[j]↔ t[i]
finpour;
```

Coût de la sélection ordinaire

- version récursive ou version itérative.
- deux types d'opérations
  - comparaisons entre clés
  - transferts (ou échanges) d'éléments

1) nombre de comparaisons

$\text{Max}(n) = n-1 + \text{Max}(n-1)$

$\text{Max}(1) = 0$ et $\text{Max}(2) = 1$

$\text{Max}(n) = \text{Max}(1) + \sum_{i = 1..n-1} i = n(n-1)/2$

Complexité en nombre de comparaisons

$\text{Moy}(n) = \text{Max}(n) = n(n-1)/2 = \Theta(n^2)$

2) nombre d'échanges

Max(n) = Moy(n)

Max(n) = 1 + Max(n-1)

Max(1) = 0   Max(2) =1

On obtient donc:

Max(n) = Moy(n) = n-1

Ce nombre doit être multiplié par trois pour obtenir le nombre de transferts (un échange = trois transferts).

La complexité en nombre de transferts est en Θ(n)

Tri par insertion

donnée:	101	|**115**	30	63	47	20
1ère insertion:	101	115	|**30**	63	47	20
2ème insertion:	30	101	115	|**63**	47	20
3ème insertion:	30	63	101	115	|**47**	20
4ème insertion:	30	47	63	101	115	|**20**
5ème insertion:	20	30	47	63	101	115

A l'issue de la $i^{ème}$ insertion, la liste est triée jusqu'à la barre verticale et le nouvel élément à insérer est indiqué en gras.

Schéma d'algorithme récursif

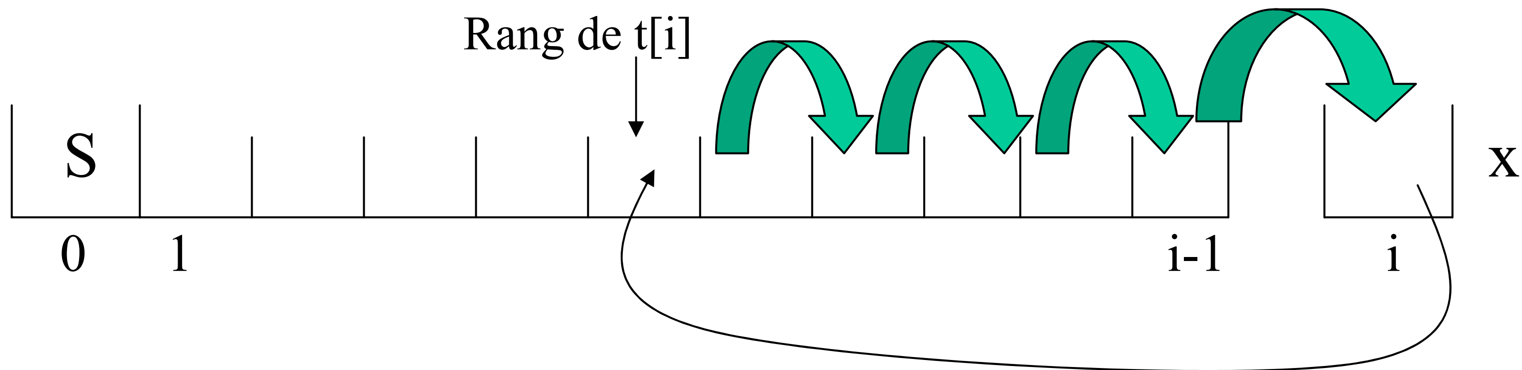
```
triinsert(t:tableau;i:entier):
    si i>1
        alors
        triinsert(t,i-1)
        --insérer t[i] à son rang
        --  parmi t[1],…,t[i-1]
```

- le tri de la totalité de la liste est effectué par triinsert(t, n)

- variantes suivant la méthode employée pour rechercher le rang de t[i].

Insertion séquentielle

- Recherche séquentielle de la place où insérer le nouvel élément.
- Recherche par la fin pour pouvoir faire le décalage
- Utilisation d'une "sentinelle".



```
triinsert(t:tableau;i:entier)
  si i>1 alors
    triinsert(t,i-1); --tri du début de la liste
      --recherche du rang de t[i]
     k ← i-1 ; x ← t[i];
     tantque t[k] > x faire t[k+1] ← t[k];
          k ← k-1 fintantque
      --on a t[k] ≤ x; la place de x est k+1
      t[k+1] ← x
  finsi
fin triinsert
```

Coût de l'insertion séquentielle

a) *Nombre de comparaisons*

$\mathrm{Max}(n) = \mathrm{Max}(n-1) + n$

$\mathrm{Max}(2) = 2$; $\mathrm{Max}(1) = 0$

d'où

$$\mathrm{Max}(n) = n.(n+1)/2 - 1$$

Le nombre de comparaisons est donc dans le pire des cas en $\Theta(n^2)$.
On montre que la complexité en moyenne est également en $\Theta(n^2)$.

b) *Nombre de transferts également* $\Theta(n^2)$.

Insertion séquentielle: version itérative (java)

```
//Algorithme de tri par insertion

public void trier(int debut, int fin) {
int k = fin ;
   //@maintaining estTrie(k+1, fin) ;
   //@decreasing k+1 ;
   while (k >= debut) {
              insererDecaler(k+1, fin) ;
         k-- ;
   } // while
} // trier(int, int)
```

```
/** Algorithme d'insertion séquentielle d'un élément
 * @param debut indice de début de la tranche
 * @param fin indice de fin de la tranche
 * l'élément à insérer est en debut-1
*/
protected void insererDecaler(int debut, int fin) {
int val = get(debut-1);
int k = debut;
    /*@ maintaining ( k == debut & estTrie(debut,fin)) ||
    @ k>debut && get(k-1)<=val && k<=fin+1 &&
    estTrie(debut,fin);
    */
    //@ decreasing fin-k+1;
    while ((k<=fin) && elements[k]< val) {
            elements[k-1] = elements[k];
            k++;
    } // while
    elements[k-1] = val;
} // insererDecaler(int, int)
```

Insertion dichotomique

- On peut faire de la recherche dichotomique dans une liste triée.
- Au cours du tri par insertion, les i-1 premiers éléments sont triés et on cherche le rang du ième.
- S'il y a des éléments égaux, on insère après.

```
triinsertdicho(t:tableau;i:entier)
  si i>1 alors
    triinsertdicho(t,i-1);
      si t[i-1]>t[i] alors
      -- la fonction "rang" est donnée plus loin;
      -- elle rend l'indice où doit être inséré t[i]
      -- entre les indices 1 et i-1, si t[i]<t[i-1]
      k ← rang(t,1, i-1, t[i]);
       -- il reste à faire les transferts;
      x ← t[i];
      pour j dans [(i-1)..k] faire
            t[j+1] ← t[j] finpour;
       t[k] ← x
      finsi
  finsi
fin triinsertdicho
```

```
rang(t:tableau;p, q, x:entier) = r : entier;
      si p = q alors r ← p
            sinon
            m ← (p+q)div 2 ;
            si x < t[m]
                  alors r ← rang(t,p,m,x)
                  sinon r ← rang(t,m+1,q,x)
            finsi
      finsi
fin rang
```

Complexité:

- Nombre de comparaisons : Θ(n*log(n)).
- Nombre de transferts : $\Theta(n^2)$.

Le tri rapide (quicksort)

Basé sur le paradigme "diviser pour régner"

Diviser: Le tableau $t[p .. r]$ est partitionné (réarrangé) en deux sous tableaux non vides $t[p .. q]$ et $t[q+1 .. r]$ tels que chaque élément de $t[p .. q]$ soit inférieur ou égal à chaque élément de $t[q+1 .. r]$. L'indice q est calculé pendant le partitionnement.

Régner: Les deux sous tableaux $t[p .. q]$ et $t[q+1 .. r]$ sont triés par des appels récursifs à la procédure principale du tri rapide.

Combiner: Comme les sous tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombiner: le tableau $t[p .. r]$ est maintenant trié.

Procédure implémentant le tri rapide

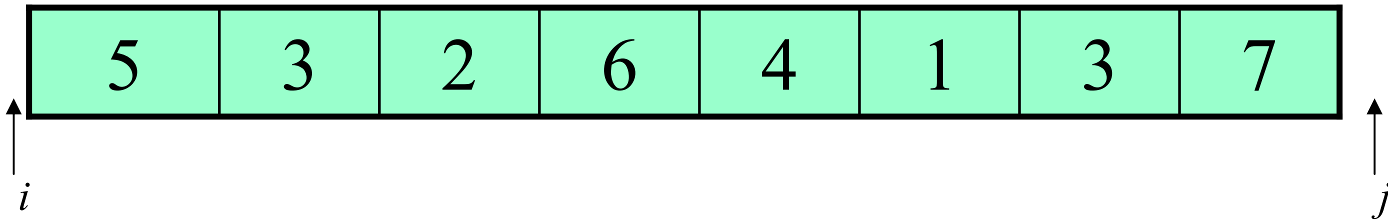
```
Tri_rapide(t, p, r)
   1 si p<r
   2 alors q ← partitionner(t, p, r)
   3       Tri_rapide(t, p, q)
   4       Tri_rapide(t, q+1, r)
```

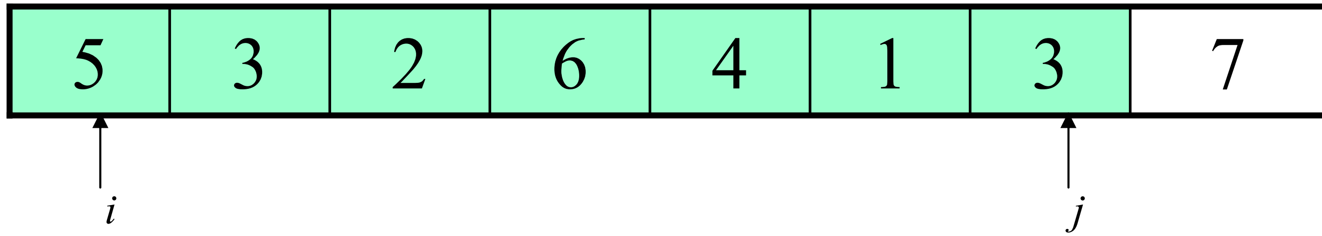
Pour trier un tableau A entier, l'appel initial est:

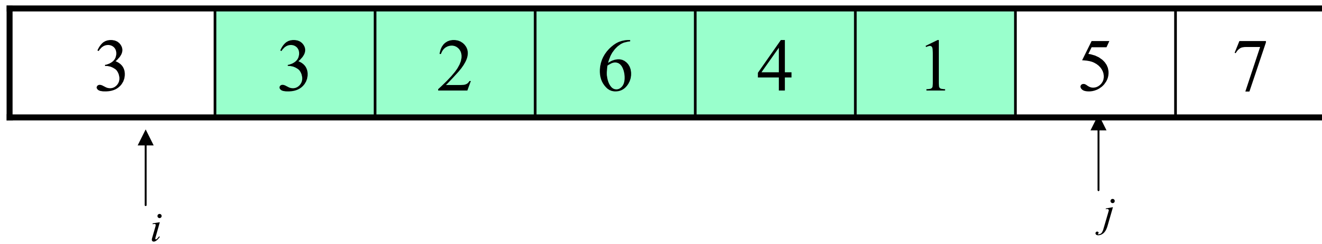
Tri_rapide(A, 1, longueur(A)).

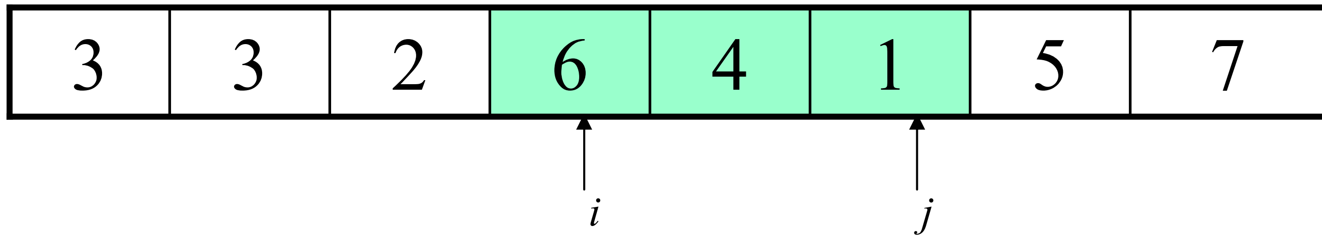
Partitionner le tableau

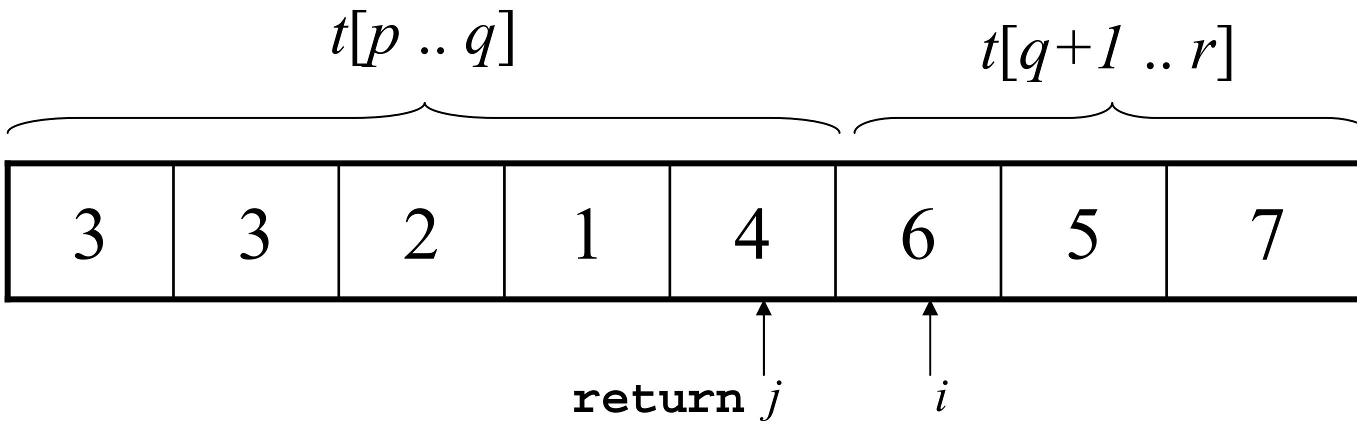
```
Partitionner(t, p, r)
 1  x ← t[p]
 2  i ← p-1
 3  j ← r+1
 4  tantque (i<j) faire
 5     répéter j ← j-1
 6     jqa t[j]≤x
 7     répéter i ← i+1
 8     jqa t[i]≥x
 9     si i<j
10        alors échanger t[i] ↔ t[j]
11        sinon retourner j
```


$t[p .. r]$
(a)
5 3 2 6 4 1 3 7
i
j


(b)
5 3 2 6 4 1 3 7
i
j

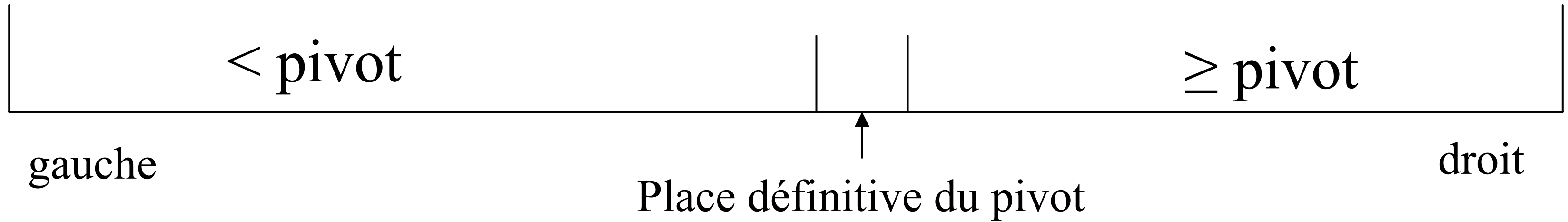

(c)
3 3 2 6 4 1 5 7
i
j


(d)
3 3 2 6 4 1 5 7
i
j

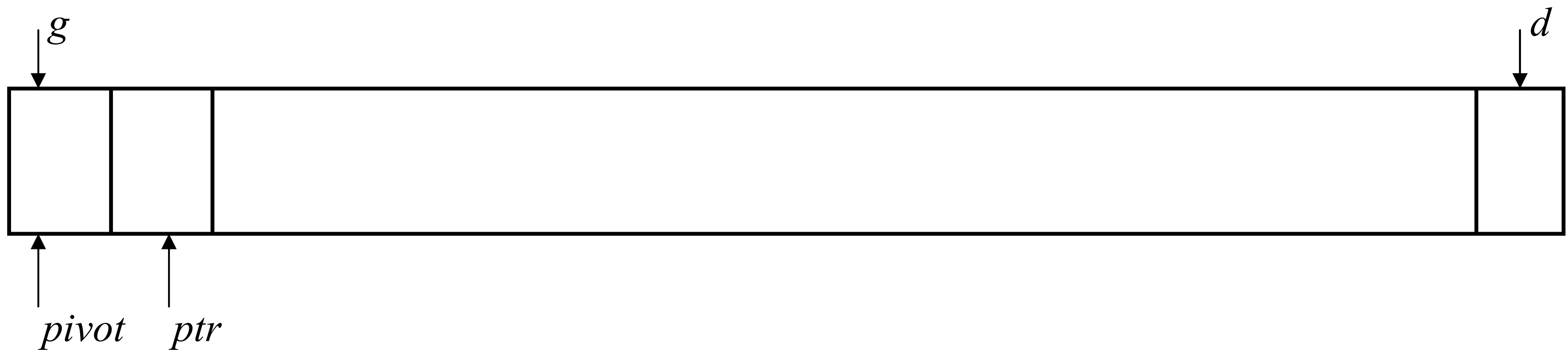


- code un peu délicat
- les indices `i` et `j` n'indexent jamais le sous-ensemble `t[p..r]` hors des limites.
- Utilisation de `t[p]` comme élément pivot `x` est essentielle.
- Temps d'exécution de Partitionner sur un tableau `t[p..r]` est $\Theta(n)$ avec $n = r - p+1$

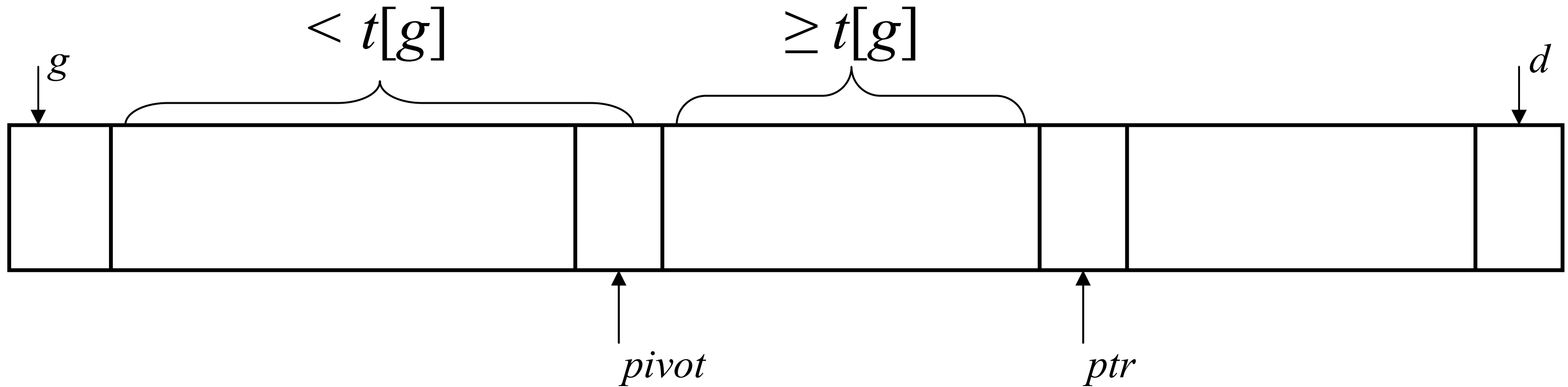
avec pivot "explicite"



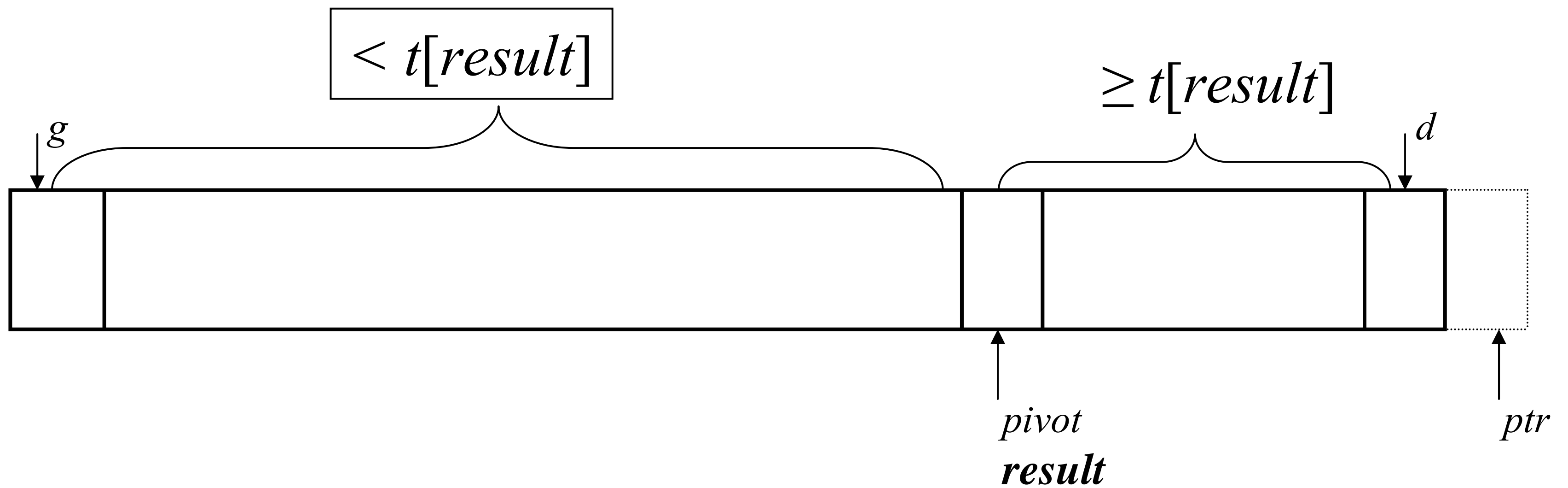
- Initialement



- Invariant



- Final



101	213	20	123	47	79	195
47	79	20	101	123	213	195
20	47	79	101	123	213	195
20	47	79	101	123	213	195
20	47	79	101	123	195	213

Les pivots sont entourées d'un cercle; les sous tableaux sont encadrés

Conclusion sur le tri rapide

- Complexité en nombre de comparaisons:
  - $\Theta(n.\log n)$ en moyenne
  - $\Theta(n^2)$ au pire

- pile auxiliaire de taille maximum $\log_2 n$

- En dessous d'un certain seuil, il devient plus avantageux d'utiliser la version itérative d'une méthode de tri simple, comme la sélection ordinaire.

- Expérimentalement, la valeur du seuil est de l'ordre de 10.