



# ASD 2

## *Algorithmique et Structure de Données*

Marc Gaëtano, Hélène Collavizza, Sylvain Schmitz

`gaetano@polytech.unice.fr`

Polytech'Nice-Sophia

Département Sciences Informatiques

930 route des Colles

06903 Sophia Antipolis - France





---

# Tris rapides et rangs

- Présentation
- Files de priorité (tas)
- Tri par tas (*heapsort*)
- Tri rapide (*quicksort*)
- Tri fusion
- Optimalité



# Tris : contexte



- **tri interne** : le tri s'effectue en mémoire centrale. Les tris externes ne sont pas abordés dans ce cours
- **tri de tableau** : un tri qui trie un tableau. Extensible à toute structure de données offrant un accès en temps (quasi)constant à ses éléments (par exemple les `Vectors` en Java)
- **tri générique** : tous les tris présentés ici sont écrits pour trier des tableaux d'objets à clef mais peuvent aisément s'adapter pour des objets totalement ordonnés quelconques



# Tris : propriétés



- **tri comparatif** : fondé sur la comparaison entre les clefs des éléments
- **tri itératif** : basé sur un ou plusieurs parcours itératifs du tableau
- **tri récursif** : basé sur une méthode récursive
- **tri en place** : ne nécessite qu'une quantité constante de mémoire supplémentaire
- **tri stable** : conserve l'ordre relatif des éléments de même clef



# Tris : méthodes quadratiques

## Propriétés

- basées sur le parcours itératif du tableau à trier
- en général deux boucles imbriquées
- complexité en  $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$  dans le pire des cas, mais aussi en moyenne
- implémentation triviale
- optimal pour les tableaux de petite taille ( $< 25$ )

# Tris : méthodes quasi-linéaires

## Les tris comparatifs optimaux

- **tri par tas (ou *heapsort*)** : complexité en  $\Theta(n \lg n)$  (pire des cas et en moyenne)
- **tri rapide (ou *quicksort*)** : complexité en  $\Theta(n \lg n)$  (en moyenne) mais peut dégénérer ( $\Theta(n^2)$ ) dans le pire des cas)
- **tri fusion** : complexité en  $\Theta(n \lg n)$  (pire des cas et en moyenne) mais nécessite un tableau auxiliaire (le tri n'est plus en place)

# Tri par tas : introduction

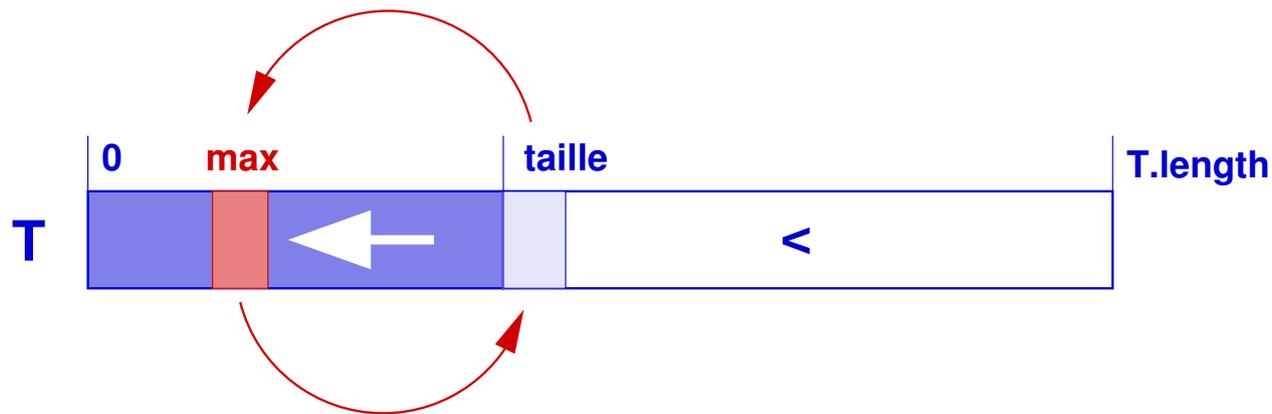


- *heapsort* en anglais
- inventé par Williams en 1964
- basé sur une structure de donnée très utile, la *file de priorité*
- complexité bornée par  $\Theta(n \lg n)$  quelle que soit la configuration initiale du tableau
- tri en place
- mise en œuvre très simple

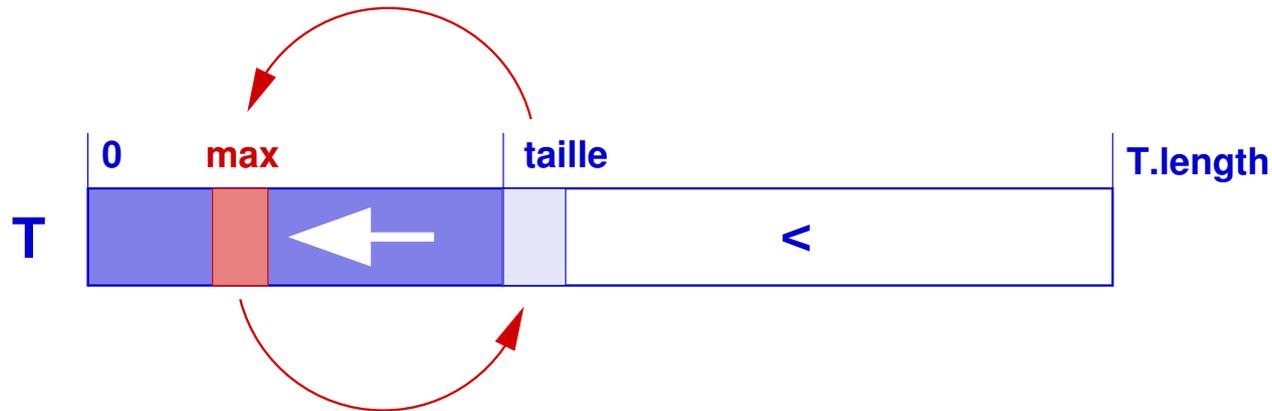


# Tri par tas : tri par sélection

1. Prendre le plus grand élément parmi les `taille` premiers éléments d'un tableau.
2. L'échanger avec le `taille`<sup>ième</sup> élément du tableau.
3. Retour à l'étape 1 en ne considérant que les `taille-1` premiers éléments du tableau.



# Tri par tas : tri par sélection



- Complexité de la recherche du plus grand élément :  $\Theta(taille)$ .
- Complexité globale du tri :  $\Theta(n^2)$ .

**Optimiser la recherche du plus grand élément**

# Tas : arbre parfait

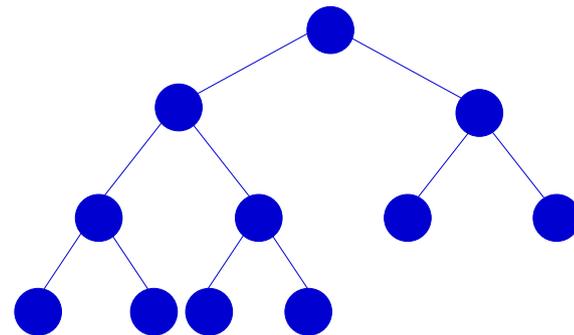
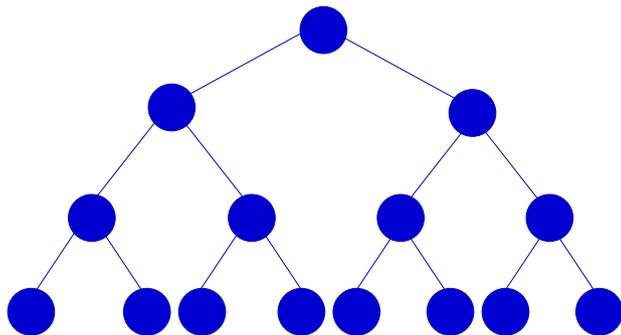
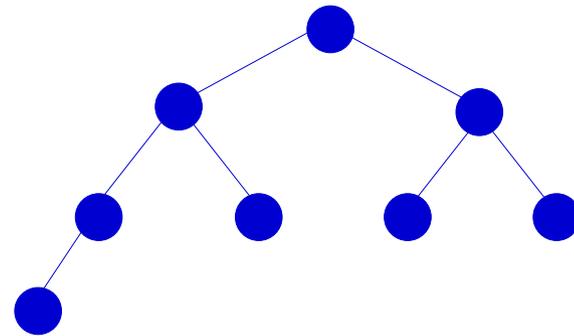
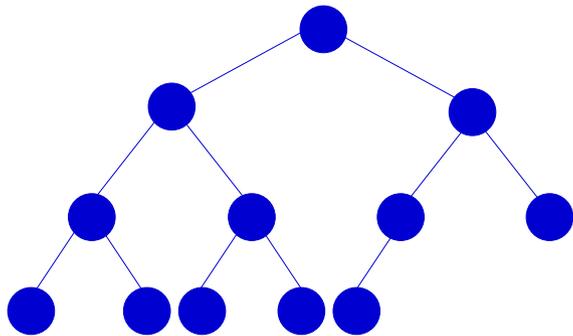
$\mathcal{A}$  est un arbre parfait si

- $\mathcal{A}$  est arbre binaire de hauteur  $h$
- une feuille de  $\mathcal{A}$  a la profondeur  $h$  ou  $h - 1$
- $\mathcal{A}$  possède  $2^{h-1}$  nœuds de profondeur  $h - 1$
- tous les nœuds de profondeur  $h$  sont regroupés le plus à gauche possible

# Tas : arbre parfait

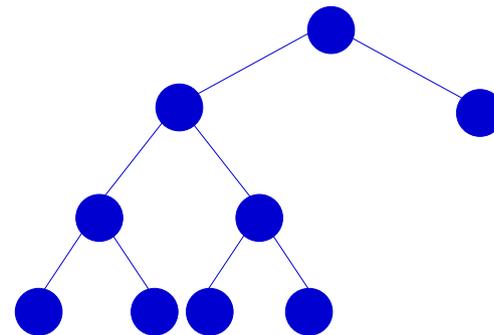
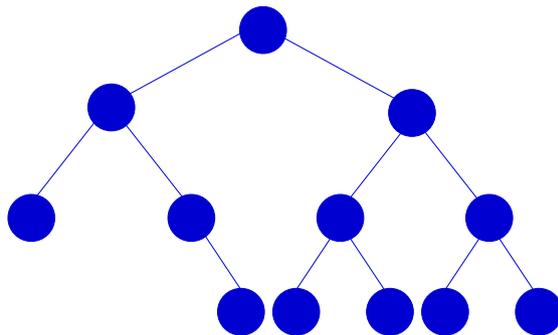
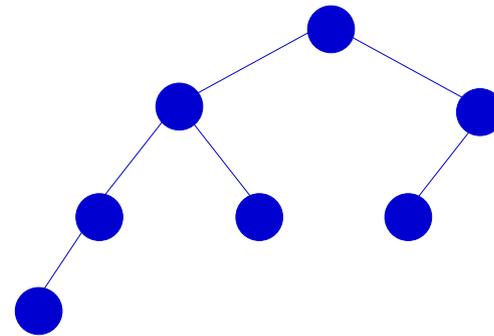
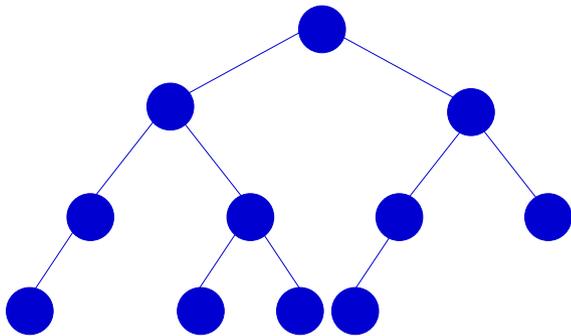


## Des arbres parfaits



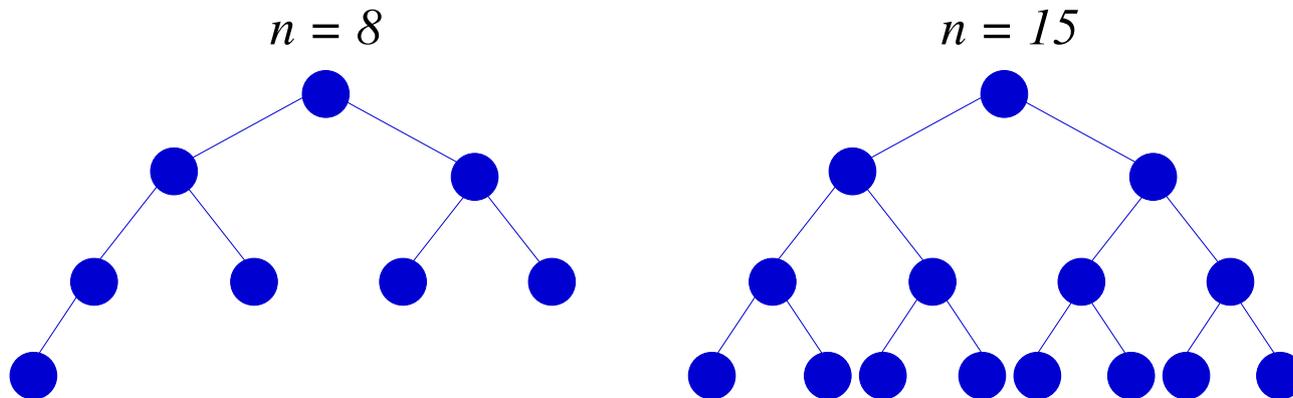
# Tas : arbre parfait

## Des arbres non parfaits



# Tas : arbre parfait

$\mathcal{A}$  un arbre parfait de taille  $n$  et de hauteur  $h$

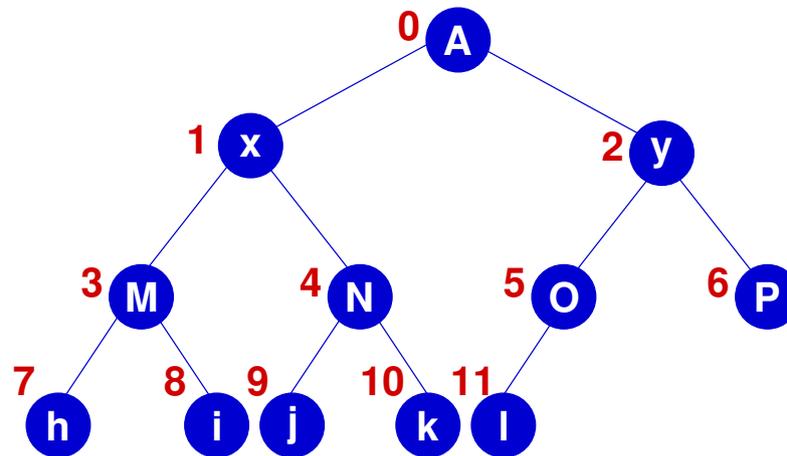


*Exercice*

Montrer que la hauteur  $h$  d'un arbre parfait est  $\lfloor \lg n \rfloor$ .

# Tas : arbre parfait

Représentation compacte à l'aide d'un tableau

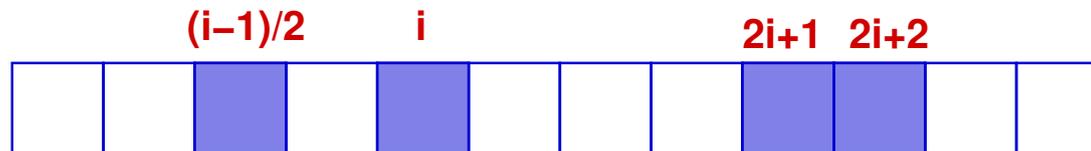
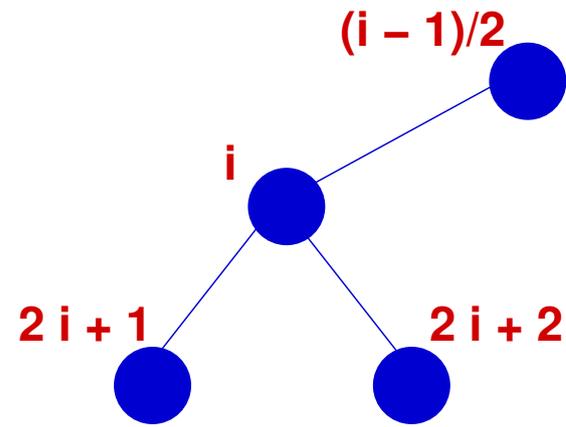


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	x	y	M	N	O	P	h	i	j	k	l				

# Tas : arbre parfait

## Codage des nœuds

- $\text{pere}(i) = (i - 1) / 2$
- $\text{gauche}(i) = 2i + 1$
- $\text{droite}(i) = 2i + 2$



# Tas : arbre parfait

## Représentation

- un tableau  $T$  d'objets à clef et un entier  $N$  égal à la longueur du préfixe du tableau  $T$  occupé par le tas
- $N = 0 \iff$  l'arbre parfait est vide et  
 $N = T.length \iff$  l'arbre parfait est plein

## Opérations triviales sur un arbre parfait

- accès à la racine ( $T[0]$ ), à la dernière feuille ( $T[N - 1]$ ), aux sous-arbres gauche et droit et au père
- ajout (resp. suppression) de la dernière feuille avec  
 $N \leftarrow N + 1$  (resp.  $N \leftarrow N - 1$ )

# Tas : arbre parfait

## *Exercices*

- Représentez l'arbre parfait correspondant au tableau contenant les clefs 53, 8, 57, 1, 90, 17, 89, 27, 35.
- À quel indice du tableau trouve-t-on la racine de l'arbre parfait ? le père de la dernière feuille ?
- Comment savoir si ce dernier est un fils gauche ou un fils droit ? Quel est l'indice de son frère ?
- Quelle est la taille d'un arbre parfait de 100 éléments ?

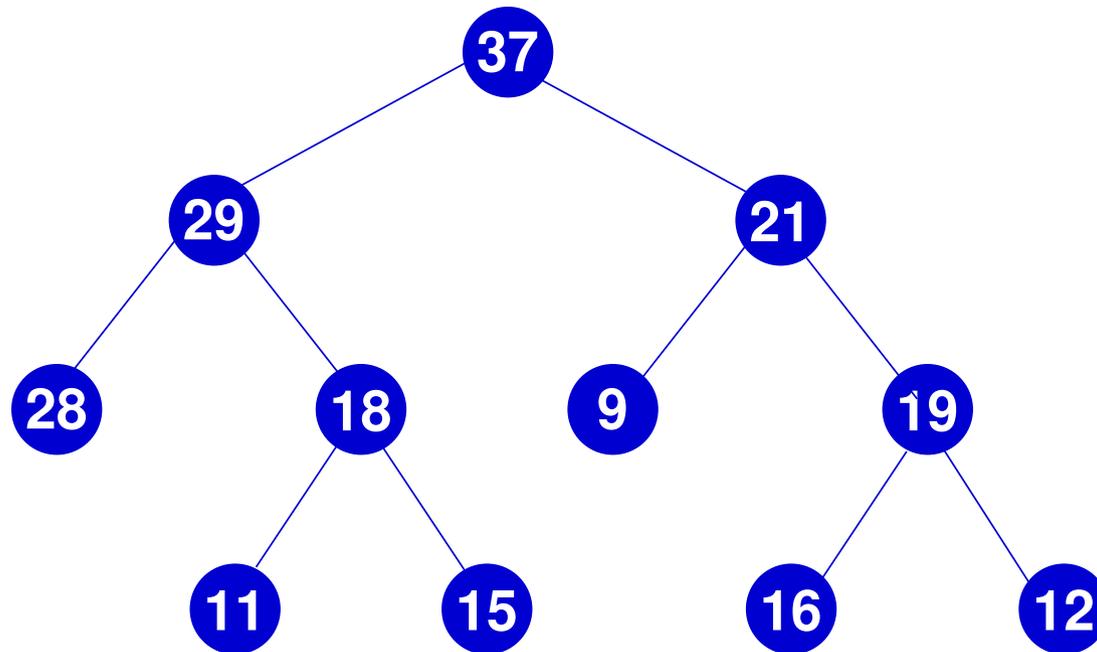
# Tas : arbre ordonné

$\mathcal{A}$  est un arbre partiellement ordonné si

- $\mathcal{A}$  est un arbre binaire
- $\mathcal{A}$  contient des objets à clef
- $\forall \mathcal{A}'$  sous-arbre de  $\mathcal{A}$ ,  $\text{clef}(r(\mathcal{A}'')) \leq \text{clef}(r(\mathcal{A}'))$ ,  $\forall \mathcal{A}''$  sous-arbre de  $\mathcal{A}'$ , où  $r(\mathcal{A})$  est la racine de l'arbre  $\mathcal{A}$  et  $\text{clef}(x)$  la clef de l'objet à clef  $x$

# Tas : arbre ordonné

## Exemple



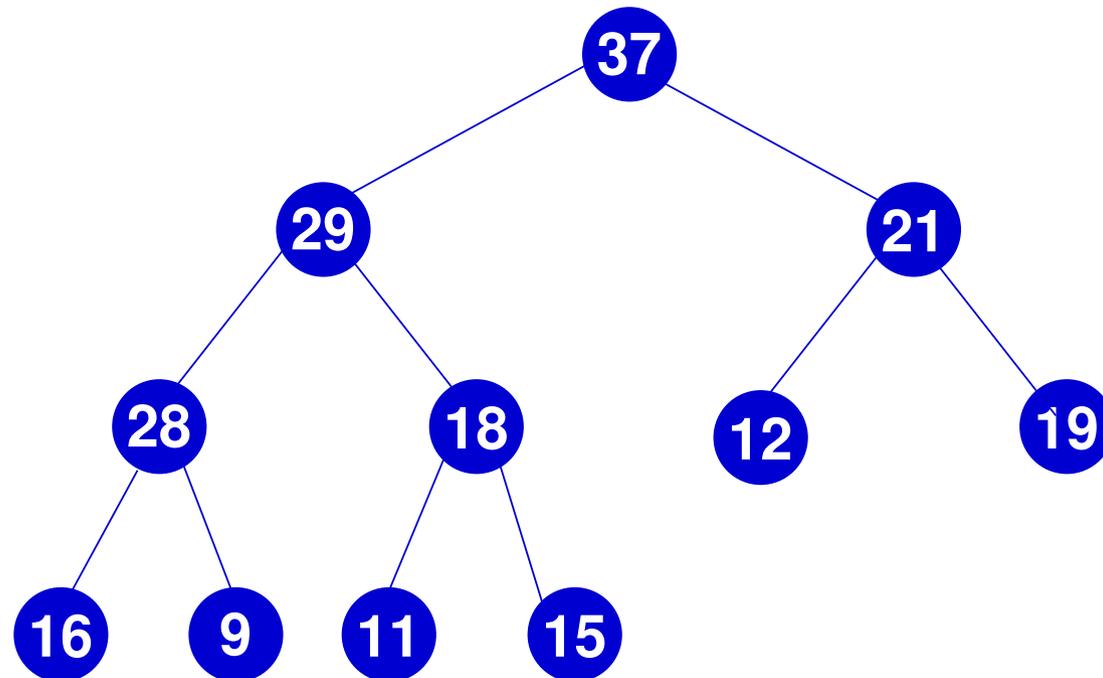
# Tas : file de priorité

## Structure de données fondamentale

- un arbre parfait partiellement ordonné de taille  $n$
- supporte les opérations :
  - *extrême* : retourne l'élément de clef maximum (ou minimum) du tas en  $\Theta(1)$
  - *extraire* : retire et retourne l'élément de clef extrême du tas en  $O(\lg n)$
  - *ajouter* : ajoute un nouvel élément au tas en  $O(\lg n)$

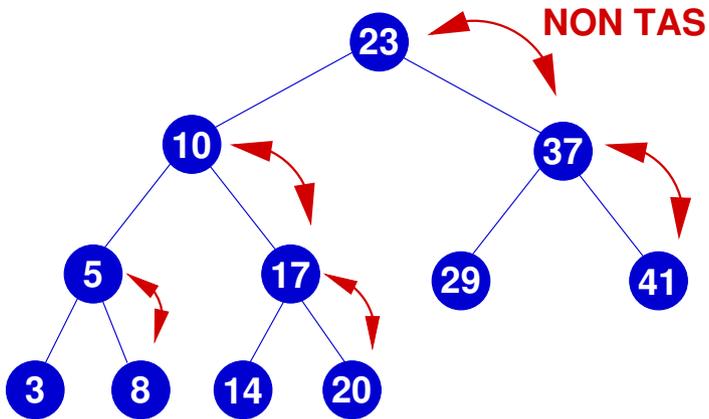
# Tas : file de priorité

## Exemple

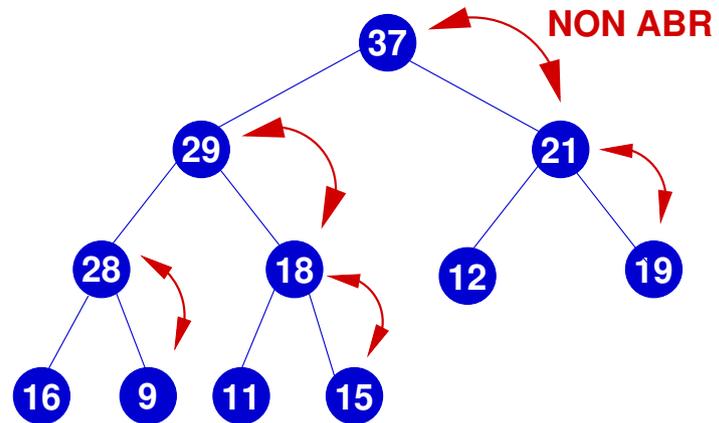


# Tas : file de priorité

Différent d'un ABR



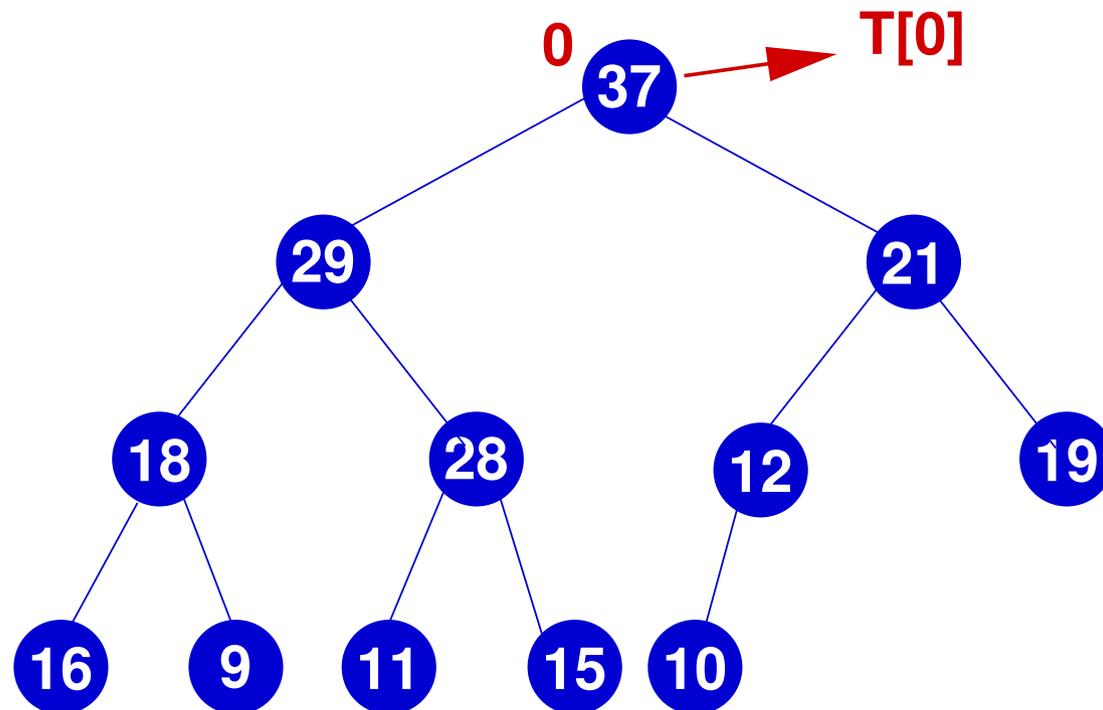
ABR



TAS

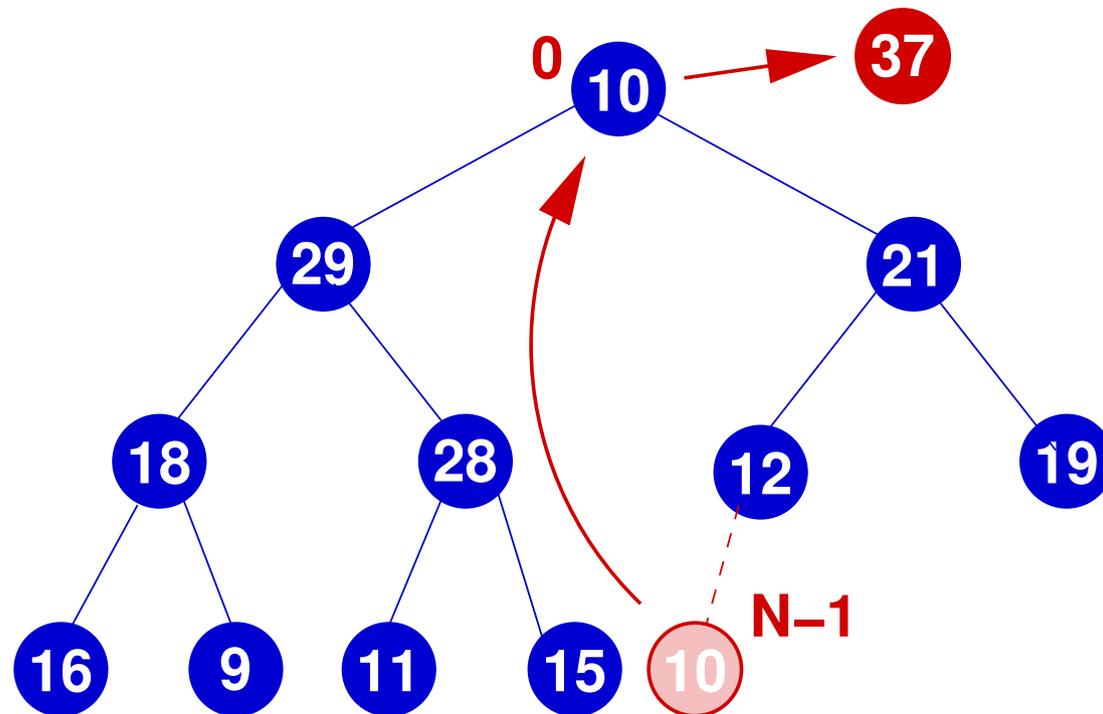
# Tas : accès au maximum

Accès au maximum en  $\Theta(1)$



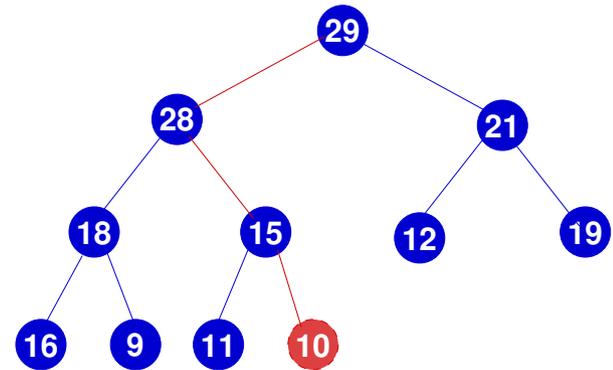
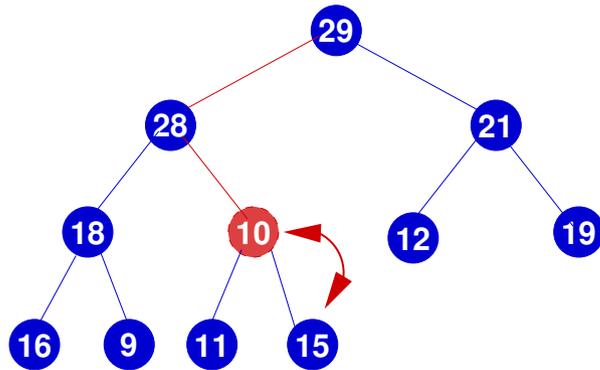
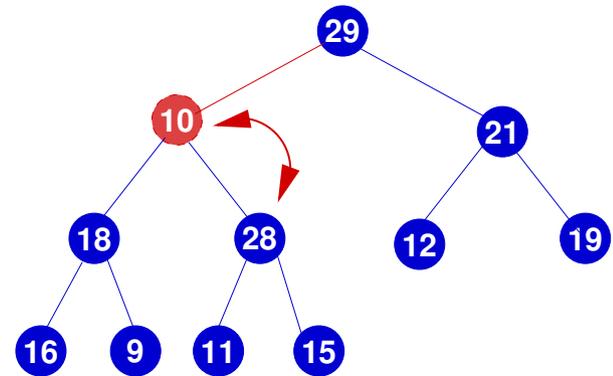
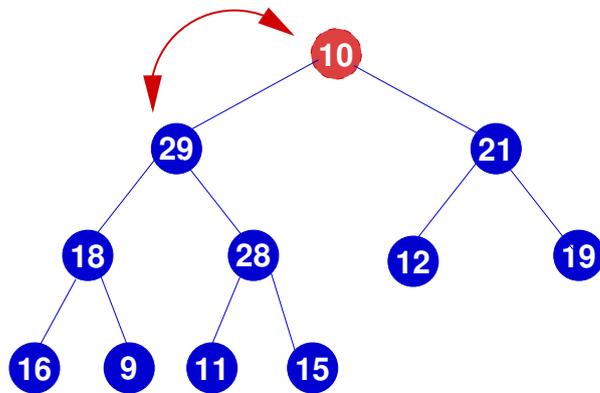
# Tas : extraire le maximum

Accès et remplacement du maximum en  $\Theta(1)$



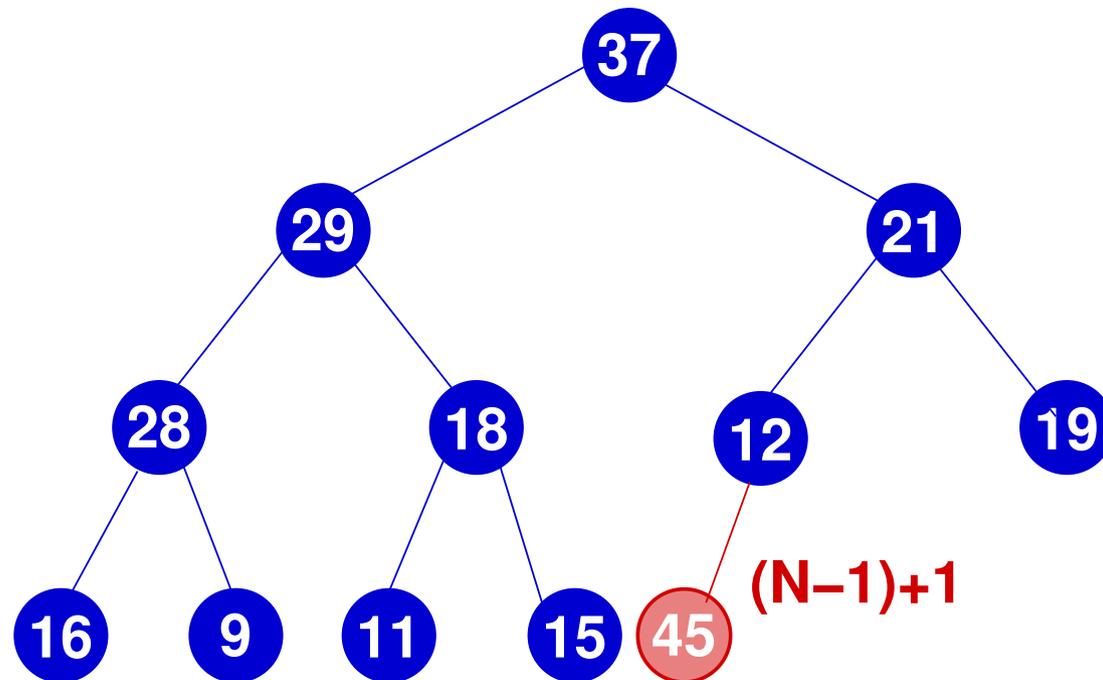
# Tas : tasser

Reconstitution de l'ordre partiel en  $O(\lg n)$



# Tas : ajout d'un élément

Ajout d'une nouvelle dernière feuille en  $\Theta(1)$





# Tas : exercices



- On considère le tableau initial ayant pour éléments 53, 8, 57, 1, 90, 17, 89, 27 et 35 comme représentant un arbre parfait. Tassez chaque sous-arbre de bas en haut jusqu'à obtention d'un tas.
- Quel est le plus grand élément du tableau ? L'extraire du tas, tasser l'arbre résultant, et obtenir le second plus grand élément du tableau.
- En déduire une méthode de tri du tableau initial.



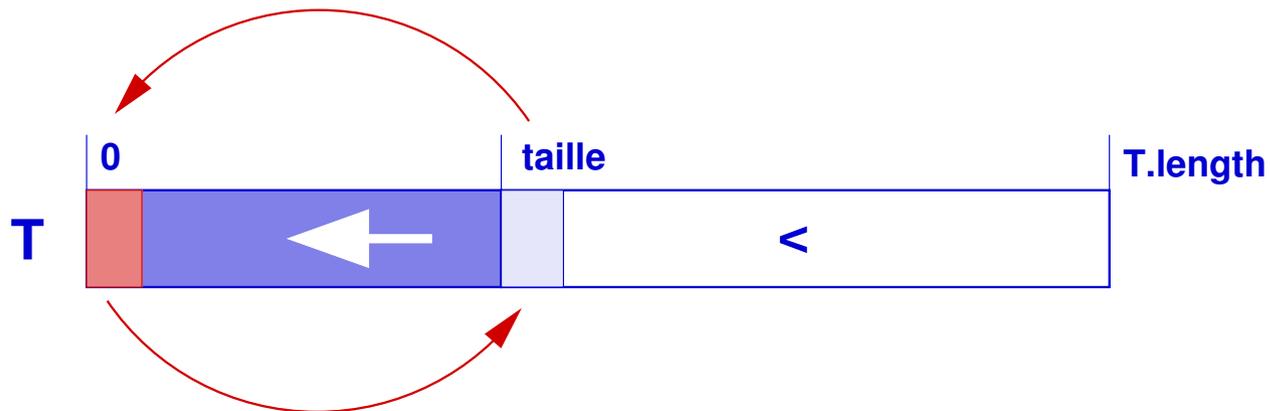
# Tri par tas : principe

- **Construction du tas initial :**

- arbre parfait : rien à faire, c'est le tableau !
- ordre partiel : tasser l'arbre de bas en haut

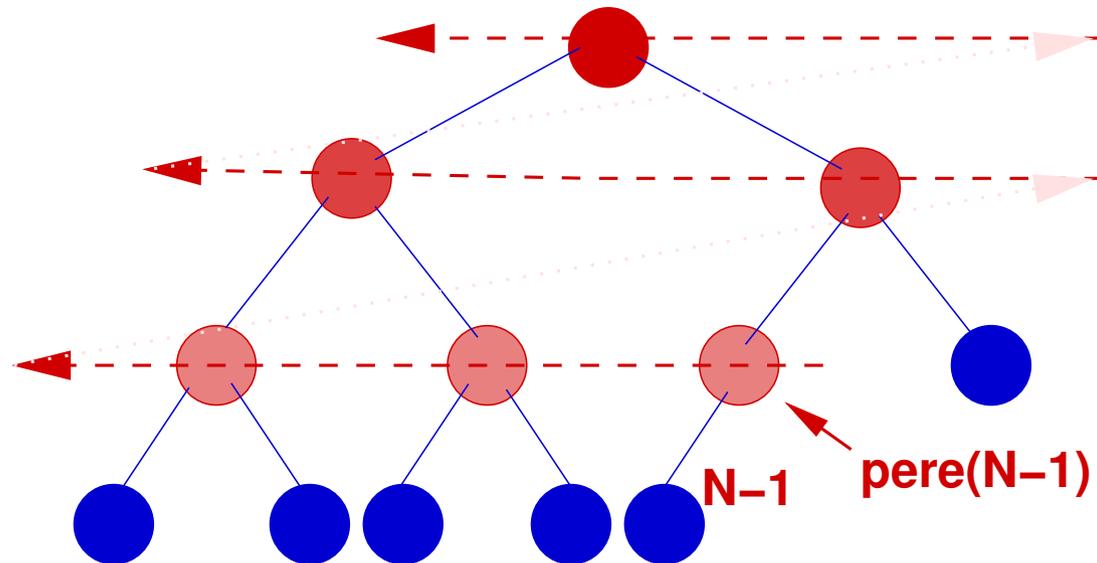
- **le tri proprement dit :**

tant que le tas n'est pas vide faire :



# Tri par tas : le tas initial

Tasser tous les nœuds des feuilles vers la racine



# Tri par tas : complexité

## Complexité de la construction du tas initial

Soit  $\mathcal{A}$  un arbre parfait de taille  $n$  (et de hauteur  $\lfloor \lg n \rfloor$ )

- un sous-arbre de hauteur  $h$  a sa racine à la profondeur  $\lfloor \lg n \rfloor - h$
- $\implies \exists$  au plus  $2^{\lfloor \lg n \rfloor - h}$  sous-arbres de hauteur  $h$
- $\exists \alpha, 0 \leq \alpha < 1 \mid \lfloor \lg n \rfloor = \lg n - \alpha$
- $\implies 2^{\lfloor \lg n \rfloor - h} = \frac{n}{2^{h+\alpha}} \leq \frac{n}{2^h}$
- $\implies \exists$  au plus  $\lceil \frac{n}{2^h} \rceil$  sous-arbres de hauteur  $h$

# Tri par tas : complexité

## Complexité de la construction du tas initial

- tasser un arbre de hauteur  $h$  s'effectue en  $\Theta(h)$
- $\implies$  la complexité de la construction du tas initial est bornée par

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} \Theta(h) = \Theta\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) \leq \Theta\left(n \sum_{h=0}^{\infty} h k^h\right) = \Theta\left(n \frac{k}{(1-k)^2}\right)$$

avec  $k = \frac{1}{2}$ , soit  $\Theta(2n) = \Theta(n)$

# Tri par tas : complexité

## Complexité du tri par tas

Pour trier un tableau de taille  $n$

- construction initiale du tas bornée par  $\Theta(n)$
- le tri proprement dit borné par  $n \Theta(\lg n)$
- **le tri est en  $\Theta(n \lg n)$**

**en moyenne et dans le pire des cas**

# Tri par tas : exercices



## *Feuille de TD8*

1. Implémentation des classes `TasDecroissant`, `TasCroissant` **et** `TrisTas`.
2. Extraction des éléments de clef `k`.
3. Tas *d*-aires.

# Tri rapide : introduction

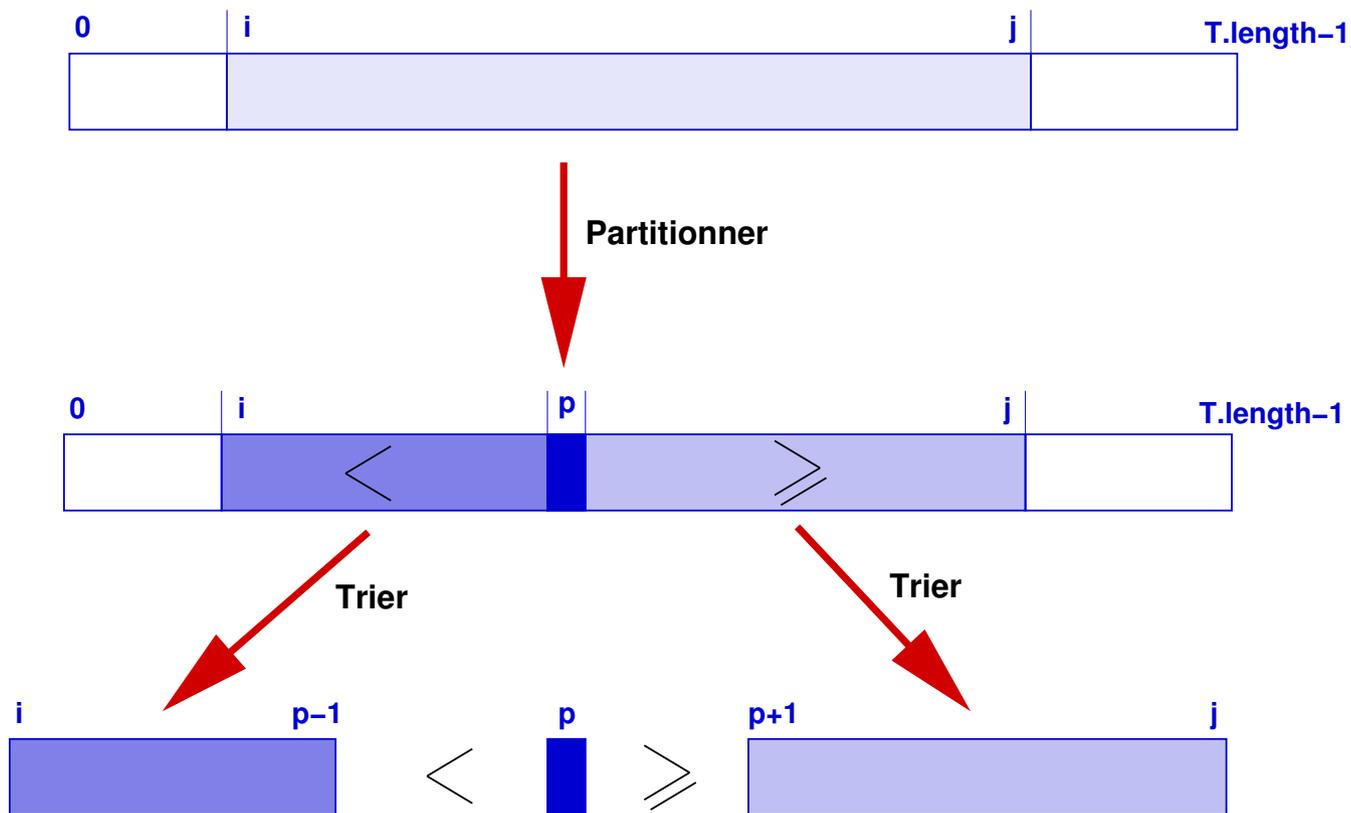


- *quicksort* en anglais
- inventé par Hoare en 1960
- appelé aussi *le tri des bijoutiers* ou *tri de perles*
- l'exemple le plus célèbre de la technique *diviser pour régner*
- basé sur des méthodes de partitionnement de tableau
- tri en place
- un dérivé intéressant : la recherche d'un rang statistique



# Tri rapide : principe

## Tri récursif



# Tri rapide : partition

## Partition d'une tranche de tableau

Un algorithme conservatif qui déplace les éléments d'une tranche  $T[i..j]$  et qui retourne l'indice  $p$



de telle façon que

$$\forall g, i \leq g < p, \forall d, p < d \leq j, \text{clef}(T[g]) < \text{clef}(T[p]) \leq \text{clef}(T[d])$$

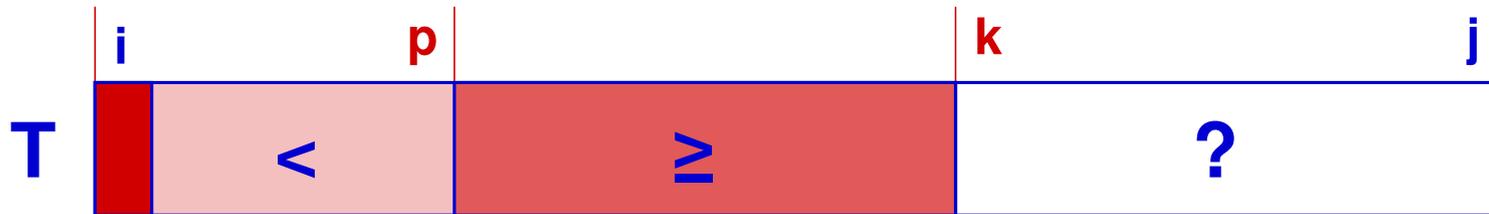
sous les conditions

$$i \leq p \leq j$$

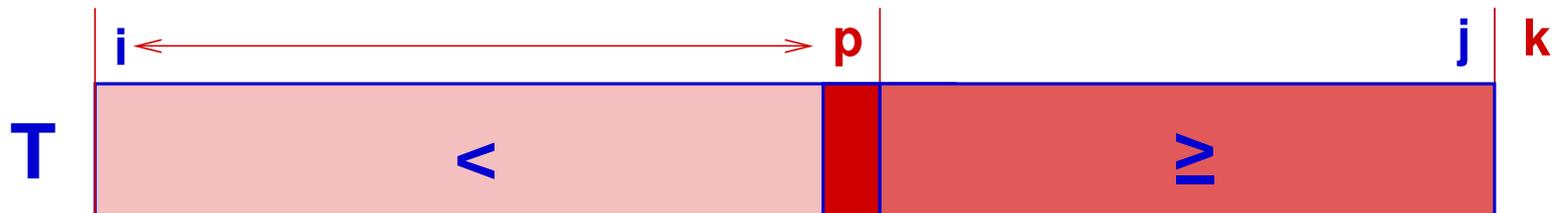
# Tri rapide : partition

## Partitionner

- invariant de boucle ( $T[i]$  est choisi comme pivot) :

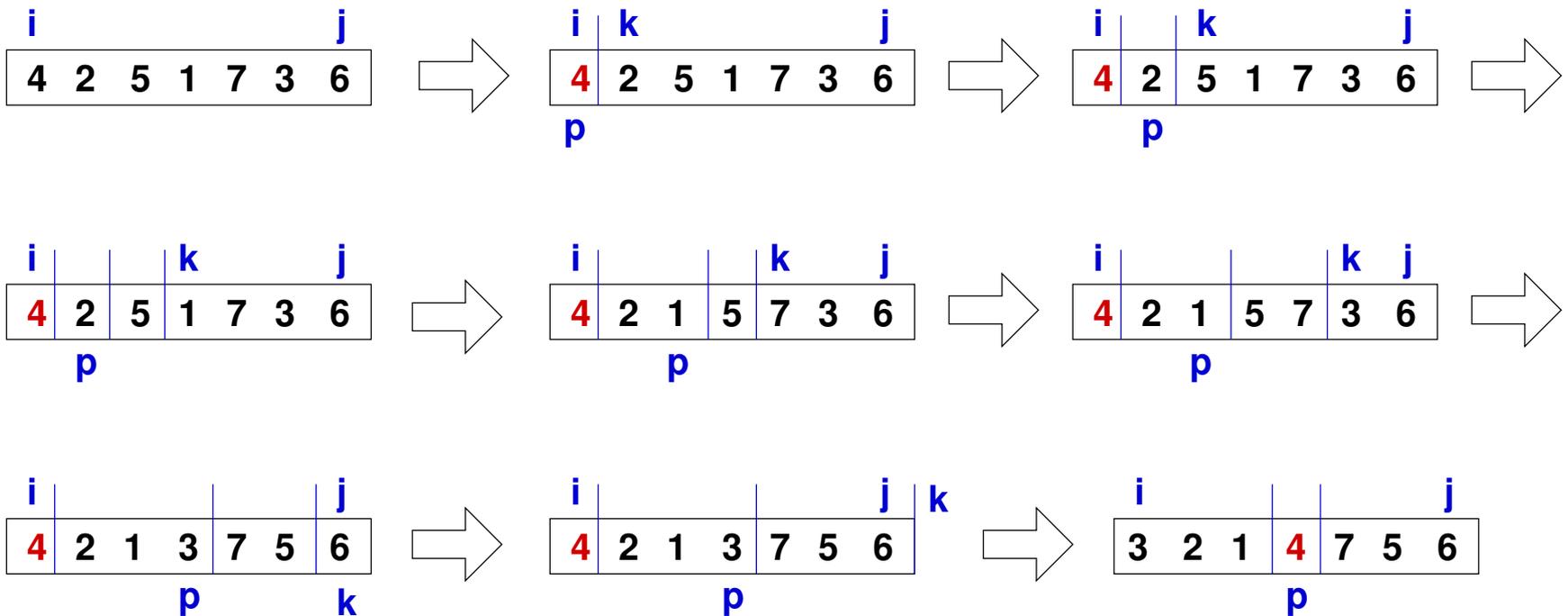


- traitement final :



# Tri rapide : partition

## Exemple



## Exercice

Écrivez la méthode de partition qui prend en paramètres les indices  $i$  et  $j$ , partitionne  $T[i..j]$ , et retourne l'indice  $p$ .

# Tri rapide : le tri



## Algorithme récursif sur une tranche de tableau

- la tranche de tableau a au plus un élément : ne fait rien
- la tranche de tableau a au moins deux éléments :
  - partitionner la tranche de tableau
  - trier récursivement les deux tranches issues du partitionnement
- le tri : appliquer sur la tranche égale au tableau entier



# Tri rapide : méthodes

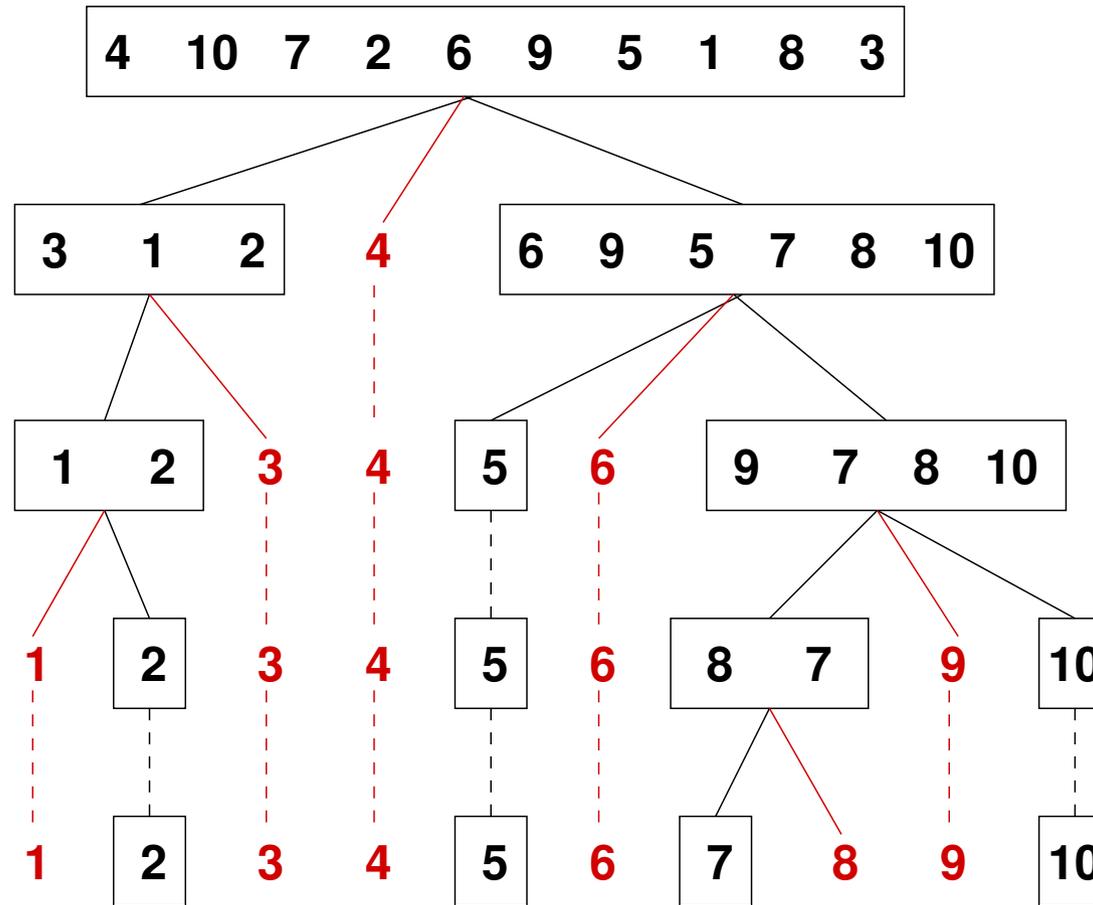
## Méthode principale

```
public void trier() {  
    trier(0,T.length - 1);  
}
```

## Méthode récursive

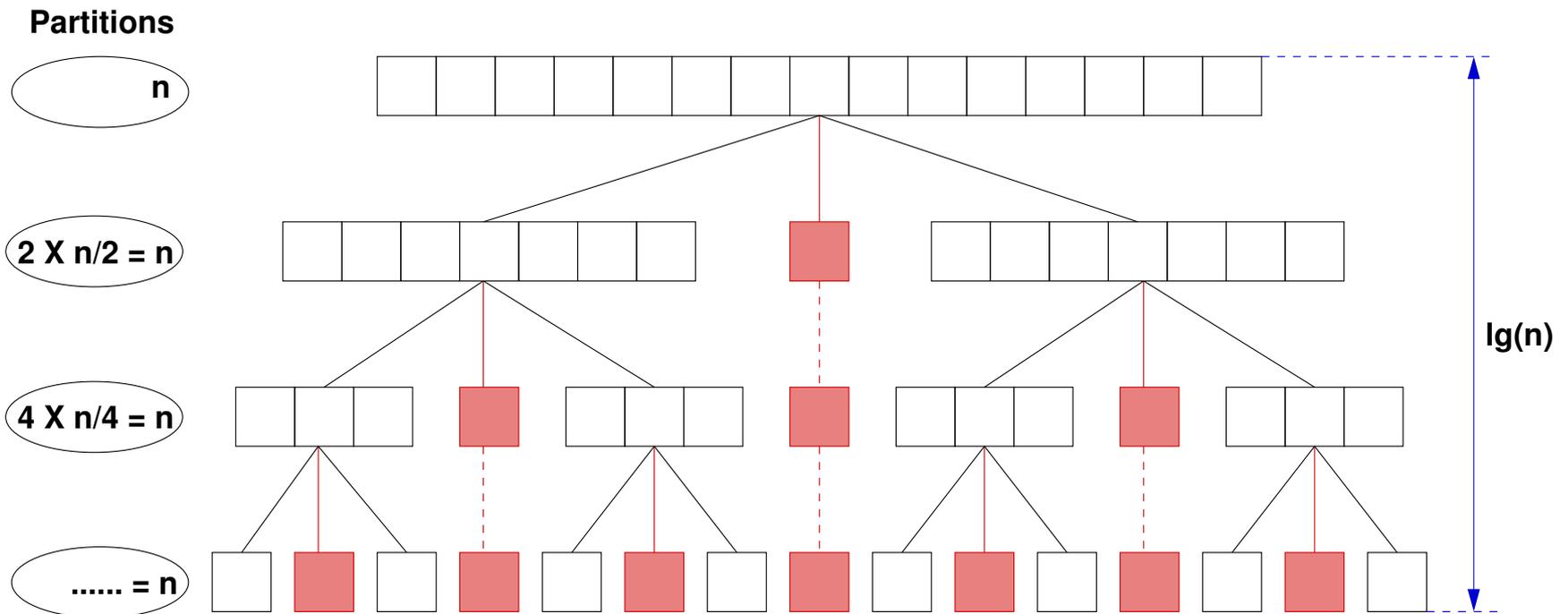
```
private void trier(int i, int j) {  
    if ( i < j ) {  
        int p = partitionner(i,j);  
        trier(i,p-1);  
        trier(p+1,j);  
    }  
}
```

# Tri rapide : exemple



# Tri rapide : complexité

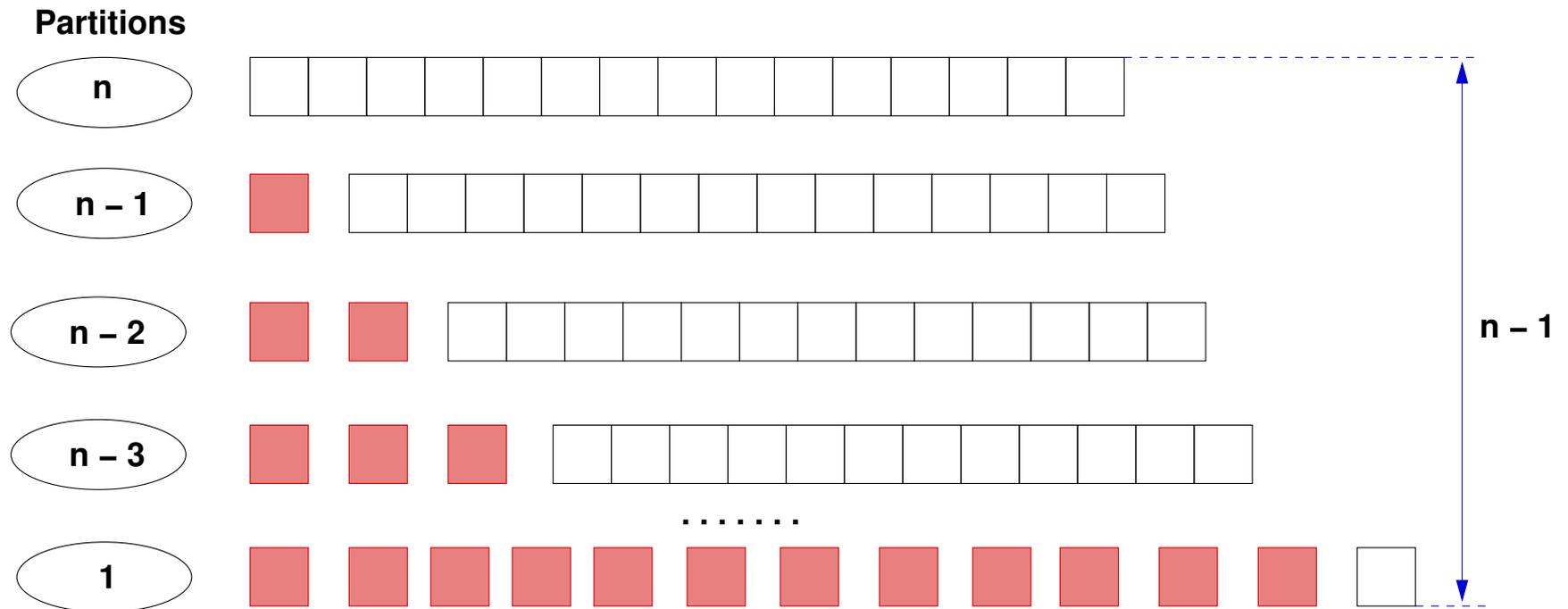
## Meilleur des cas



$\Rightarrow$  complexité =  $\Theta(n \lg n)$

# Tri rapide : complexité

## Pire des cas



⇒ complexité =  $\Theta(n(n + 1)/2) = \Theta(n^2)$

# Tri rapide : complexité

## *Exercices*

Soient les éléments de clefs 11, 13, 16, 17, 5, 19 et 25.

- Dans quel ordre les mettre pour que le tri se fasse en  $\Theta(n^2)$  ?
- Pour qu'il se fasse en  $\Theta(n \lg n)$  ?

# Tri rapide : complexité



- **meilleur des cas** : la méthode de partitionnement produit toujours deux tranches de taille égale

$$C(n) = 2C\left(\frac{n}{2}\right) + \Theta(n) \implies C(n) = \Theta(n \lg n)$$

- **pire des cas** : la méthode de partitionnement produit toujours une tranche de longueur 0

$$C(n) = C(n - 1) + \Theta(n) \implies C(n) = \Theta(n^2)$$

- **en moyenne** : ?



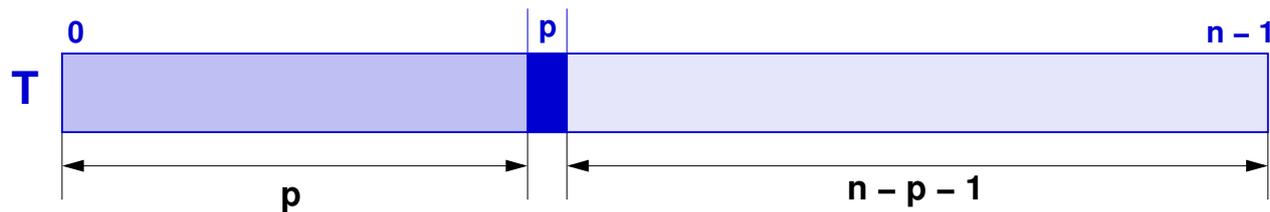
# Tri rapide : complexité

## Hypothèses de calcul de la complexité en moyenne

- on évalue  $C_n$  le nombre moyen de comparaisons entre les objets à clef durant le tri d'un tableau de taille  $n$
- on évalue `trier` qui utilise `partitionner` qui effectue exactement  $k - 1$  comparaisons sur une tranche de longueur  $k$
- les clefs des éléments du tableau sont toutes distinctes
- les places des éléments dans le tableau sont équiprobables

# Tri rapide : complexité

Après l'appel à partitionner on a



- le nombre moyen de comparaisons effectuées par les deux appels récurifs est  $C_p + C_{n-p-1}$
- la moyenne des quantités précédentes quand  $p$  varie est  $\frac{1}{n} \sum_{p=0}^{n-1} (C_p + C_{n-p-1})$
- partitionner effectue  $n - 1$  comparaisons

# Tri rapide : complexité

La complexité en moyenne du tri rapide est

$$C_n = n - 1 + \frac{1}{n} \sum_{p=0}^{n-1} (C_p + C_{n-p-1})$$

soit, par symmétrie

$$C_n = n - 1 + \frac{2}{n} \sum_{p=0}^{n-1} C_p$$

*Exercice*

Montrer que cette expression se réduit à

$$C_n \approx 2nH_n (\approx 1,38 n \lg n)$$

# Tri rapide : améliorations

## Choix du pivot

Choix de l'élément médian parmi trois

## Petites tranches

Utilisation d'un tri naïf (insertion) sur les tranches de longueur inférieure à  $k$  ( $k \approx 20$ )

## Duplication des clefs

Nécessité d'une méthode de partition adaptée

Amélioration de près de 25% en moyenne sur le temps d'exécution par rapport à la version initiale

# Tri rapide : exercices



*Feuille de TD8*

4. Implémentation des classes `PartitionTernaire` et `TriRapideTernaire`.
5. Implémentation de la classe `PresqueTrier`.



# Rang : définition

## Le $k$ -ième rang statistique d'un tableau

- l'élément de rang  $k$
- le  $k$ -ième plus petit élément
- l'élément d'indice  $k - 1$  dans le tableau trié

## sous l'hypothèse

- toutes les clefs sont distinctes
- sinon, notion d'*un* rang statistique possible parmi  $r$

# Rang : méthodes

## Méthode naïve

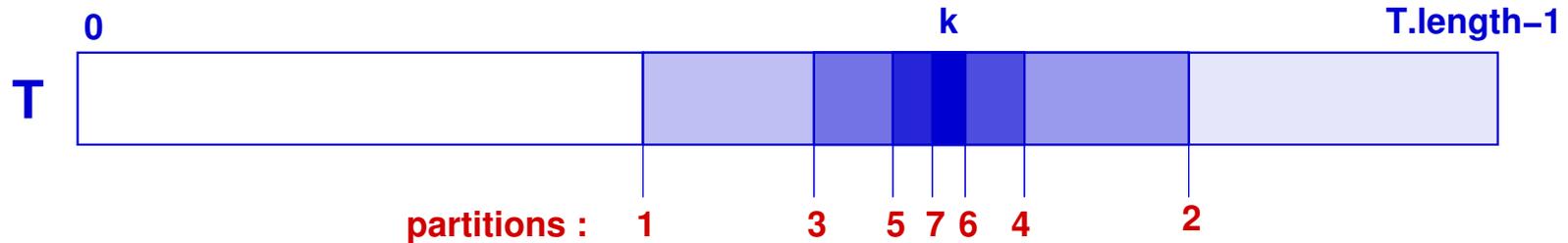
On trie le tableau en  $\Theta(n \lg n)$  et on trouve le rang en  $\Theta(1)$

## Méthode rapide

Algorithme destructif basé sur le partitionnement

- les éléments du tableau sont conservés mais déplacés
- *moitié* du tri rapide
- à la fin, l'élément de rang  $k$  occupe la place qu'il occuperait dans le tableau trié, à l'indice  $k - 1$

# Rang : méthode rapide



- partitionner récursivement la tranche qui contient l'indice  $k$
- on a trouvé  $k$  quand :
  - la tranche à partitionner est de longueur 1
  - l'indice du pivot est  $k$  (seulement avec la méthode `partitionner` présentée dans ce cours)

# Rang : méthodes

## Méthode principale

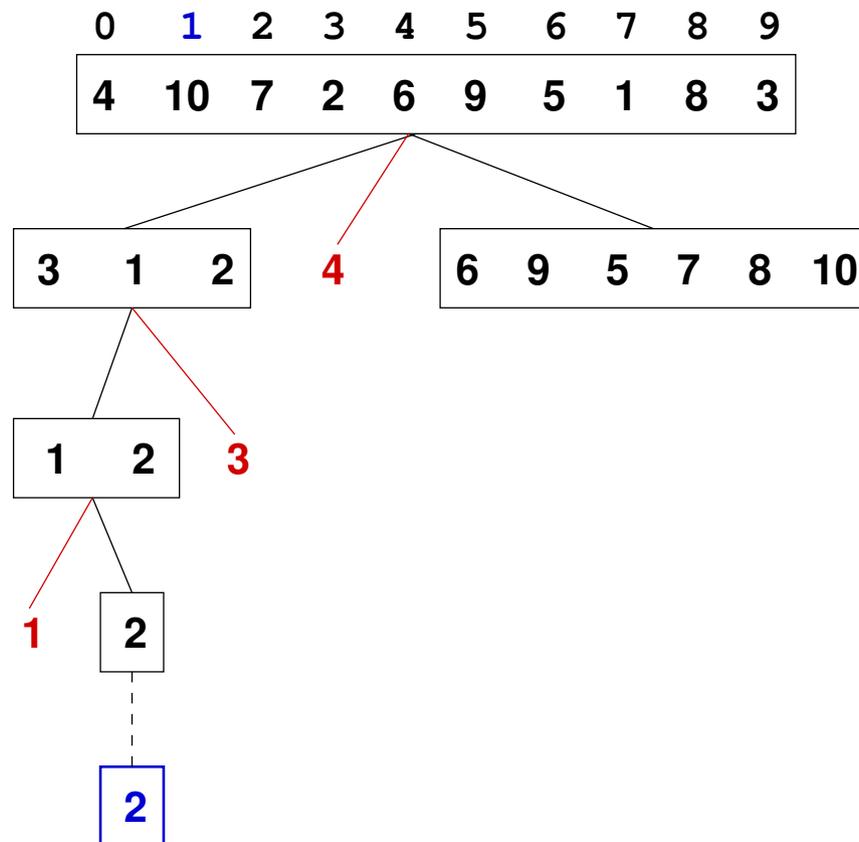
```
public ObjetAClef rang(int k) {  
    return rang(0,T.length - 1,k - 1);  
}
```

## Méthode récursive

```
private ObjetAClef rang(int i, int j, int k) {  
    if ( i == j ) return T[i];  
    int p = partitionner(i,j);  
    if ( k < p ) return rang(i,p-1,k);  
    if ( k > p ) return rang(p+1,j,k);  
  
    return T[p];  
}
```

# Rang : exemple

## Recherche de l'élément de rang 2



# Rang : complexité



- **meilleur des cas** : la méthode de partitionnement produit toujours deux tranches de taille égale

$$C_n = C_{\frac{n}{2}} + \Theta(n) \implies C_n = \Theta(n)$$

- **pire des cas** : la méthode de partitionnement produit toujours une tranche de longueur 0 et le rang est dans la dernière tranche

$$C_n = C_{n-1} + \Theta(n) \implies C_n = \Theta(n^2)$$

- **en moyenne** : ?



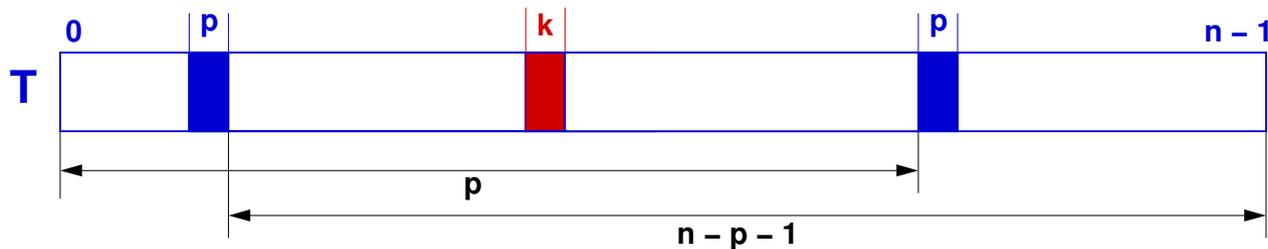
# Rang : complexité

## Hypothèses de calcul de la complexité en moyenne

- on évalue  $C_n$  le nombre moyen de comparaisons entre les objets à clef durant la recherche d'un rang dans un tableau de taille  $n$
- on évalue `rang` qui utilise `partitionner` qui effectue exactement  $k - 1$  comparaisons sur une tranche de longueur  $k$
- les clefs des éléments du tableau sont toutes distinctes
- équiprobabilité de la place des éléments

# Rang : complexité

Après l'appel à partitionner on a



- le nombre moyen de comparaisons effectuées par l'un des deux appels récurifs est soit  $C_p$  soit  $C_{n-p-1}$
- la moyenne des quantités précédentes quand  $p$  varie est  $\frac{1}{n} \left( \sum_{p=0}^{k-1} C_{n-p-1} + \sum_{p=k+1}^{n-1} C_p \right)$
- partitionner effectue  $n - 1$  comparaisons

# Rang : complexité

La complexité en moyenne de la recherche est

$$C_n = n - 1 + \frac{1}{n} \left( \sum_{p=0}^{k-1} C_{n-p-1} + \sum_{p=k+1}^{n-1} C_p \right)$$

soit, par symétrie

$$C_n \leq n - 1 + \frac{2}{n} \sum_{p=\lceil \frac{n}{2} \rceil}^{n-1} C_p$$

On montre par récurrence sur  $n$  que  $C_n \leq O(n)$

# Rang : exercices



## *Exercices*

- Comment pourrait-on modifier le tri par sélection pour obtenir un algorithme de recherche de l'élément de rang  $k$  ?
- Quelle est sa complexité dans le pire des cas ?
- Qu'en est-il si l'on utilise un tas pour optimiser la sélection ?

## *Feuille de TD8*

6. Implémentation de la classe `RangEnTrois`.

 7. Construction d'un oléoduc.

# Tri fusion : introduction

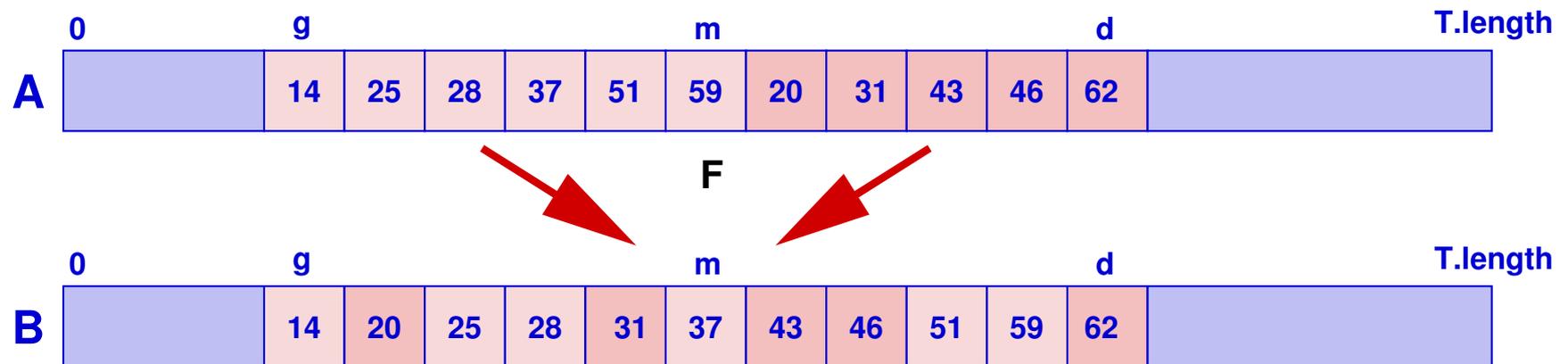


- *merge sort* en anglais
- inventé par John von Neumann en 1945
- un autre exemple standard de la technique *diviser pour régner*
- basé sur des méthodes de fusion de listes ou de tableaux
- nécessite un tableau auxiliaire (non en place)
- complexité de  $\Theta(n \lg n)$  en moyenne, dans le pire et le meilleur des cas



# Méthode de fusion

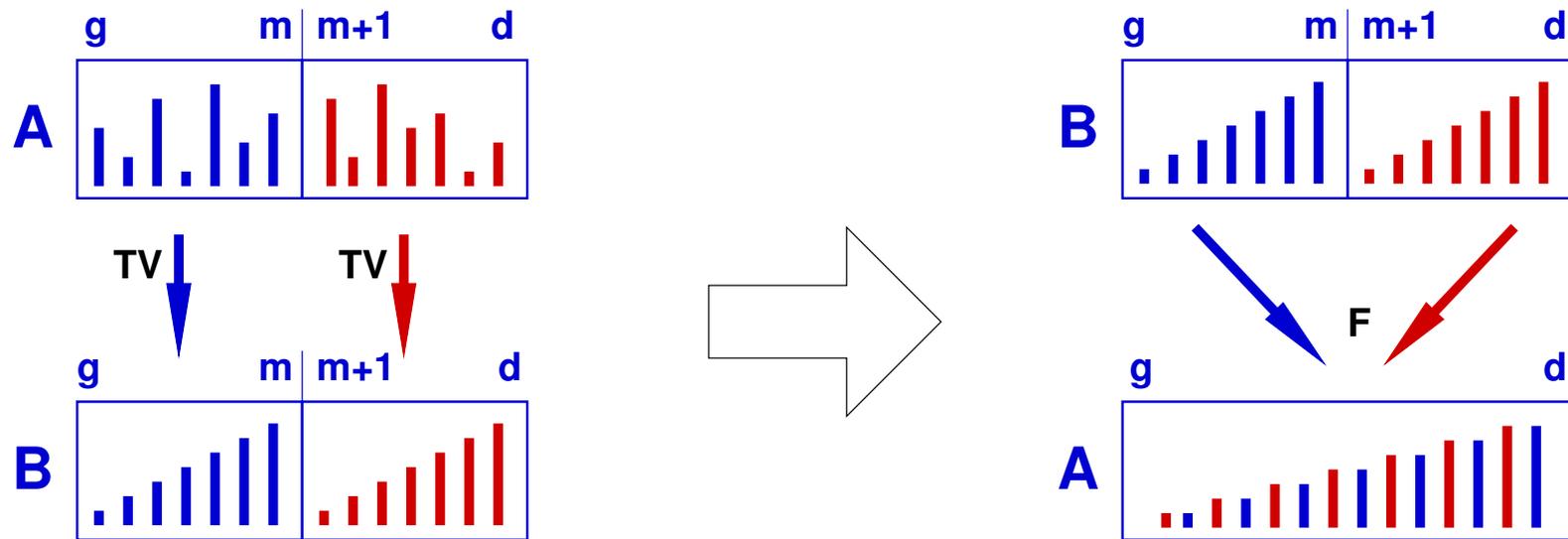
Un algorithme itératif **F** qui fusionne deux tranches triées  $A[g..m]$  et  $A[m + 1..d]$  en une tranche triée auxiliaire  $B[g..d]$  avec la complexité  $\Theta(d - g + 1)$



sous les conditions  $g \leq m < d$

# Tri fusion : principe

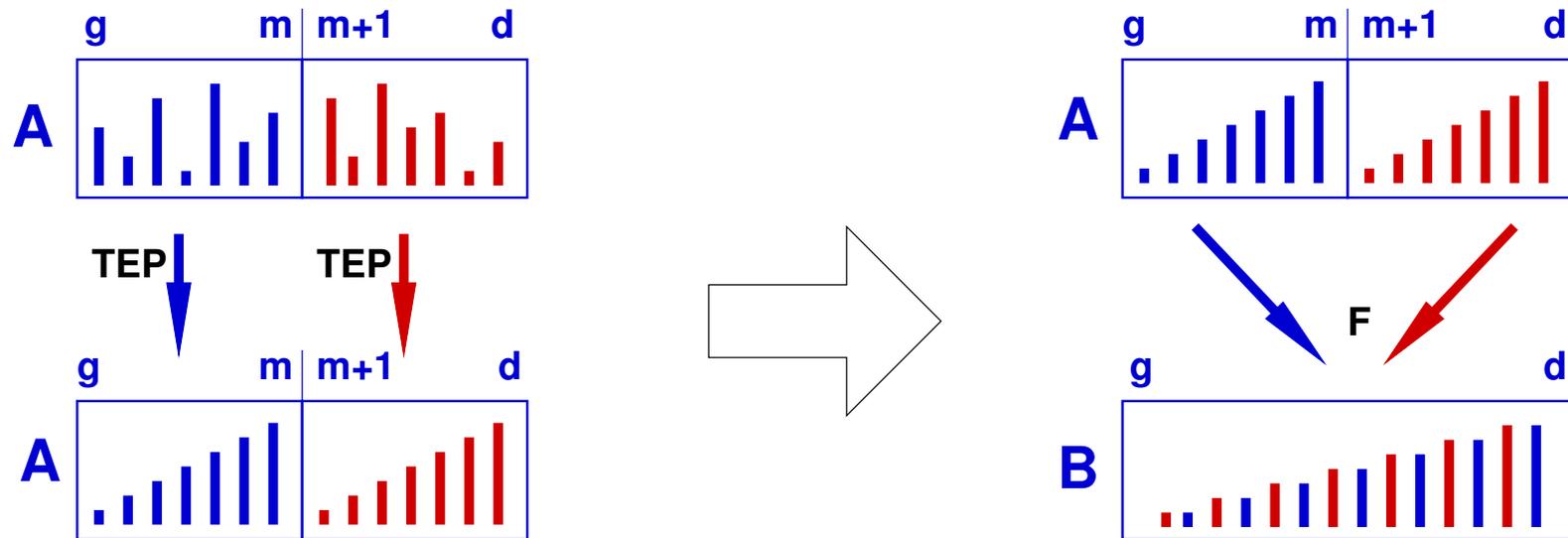
Tri En Place de  $A[g..d]$  à l'aide de  $B[g..d]$  (TEP)



où TV est le Tri de  $A[i..j]$  Vers  $B[i..j]$  et F la fusion

# Tri fusion : principe

Tri de  $A[g..d]$  Vers  $B[g..d]$  (TV)



où TEP est le Tri En Place de  $A[i..j]$  à l'aide de  $B[i..j]$   
et F la fusion

# Tri fusion : exercices

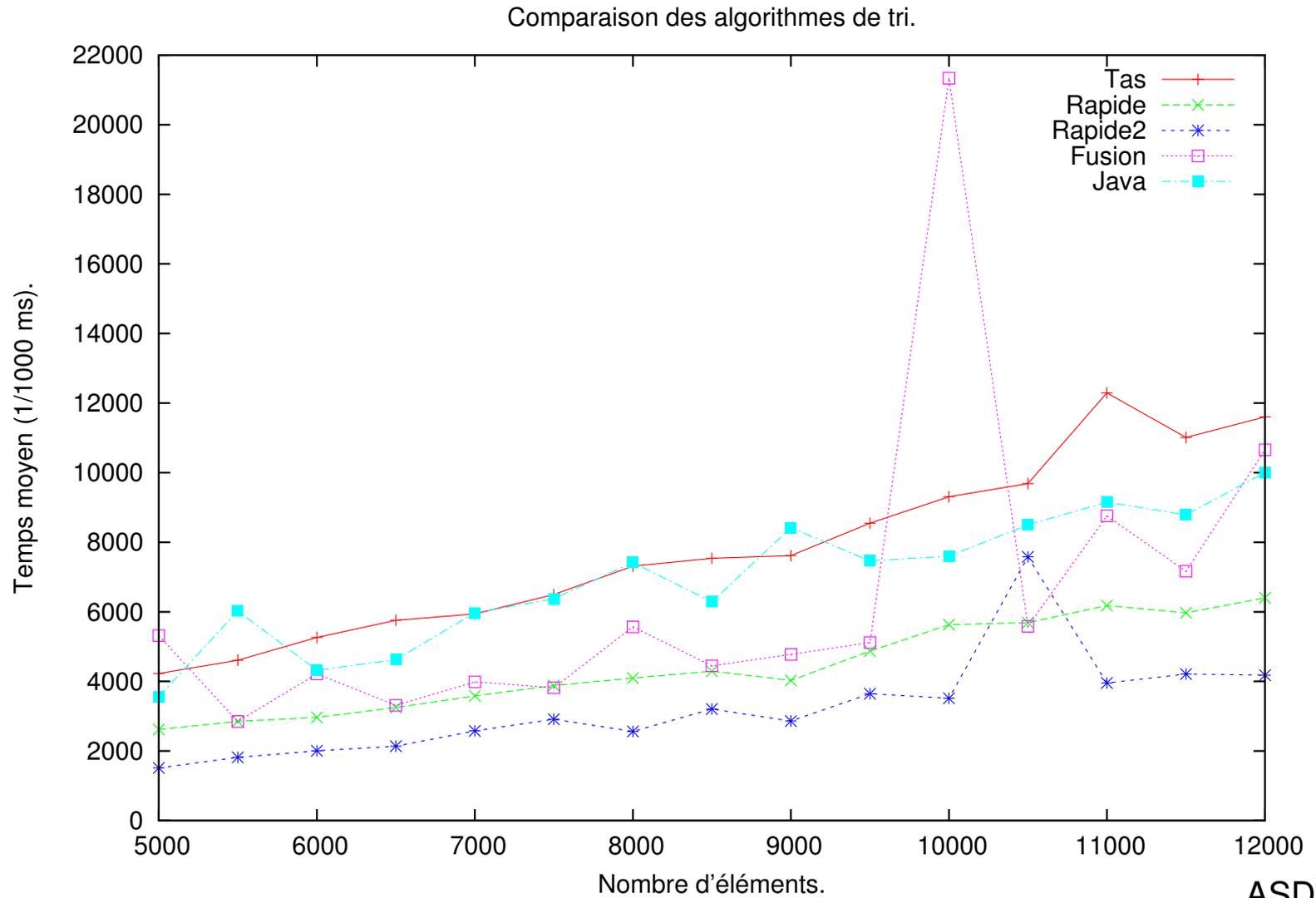
## *Exercices*

- Dans quel cas un TEP n'a-t'il rien à faire ? Que doit faire un TV dans ce même cas ?
- Trier la table contenant initialement les clefs 53, 8, 57, 1, 90, 17, 89, 27 et 35.
- L'ordre initial des éléments a-t'il un impact sur le temps d'exécution du tri ? Que dire des cas  $n = 2^m$  et  $n = 2^m + 1$  ?

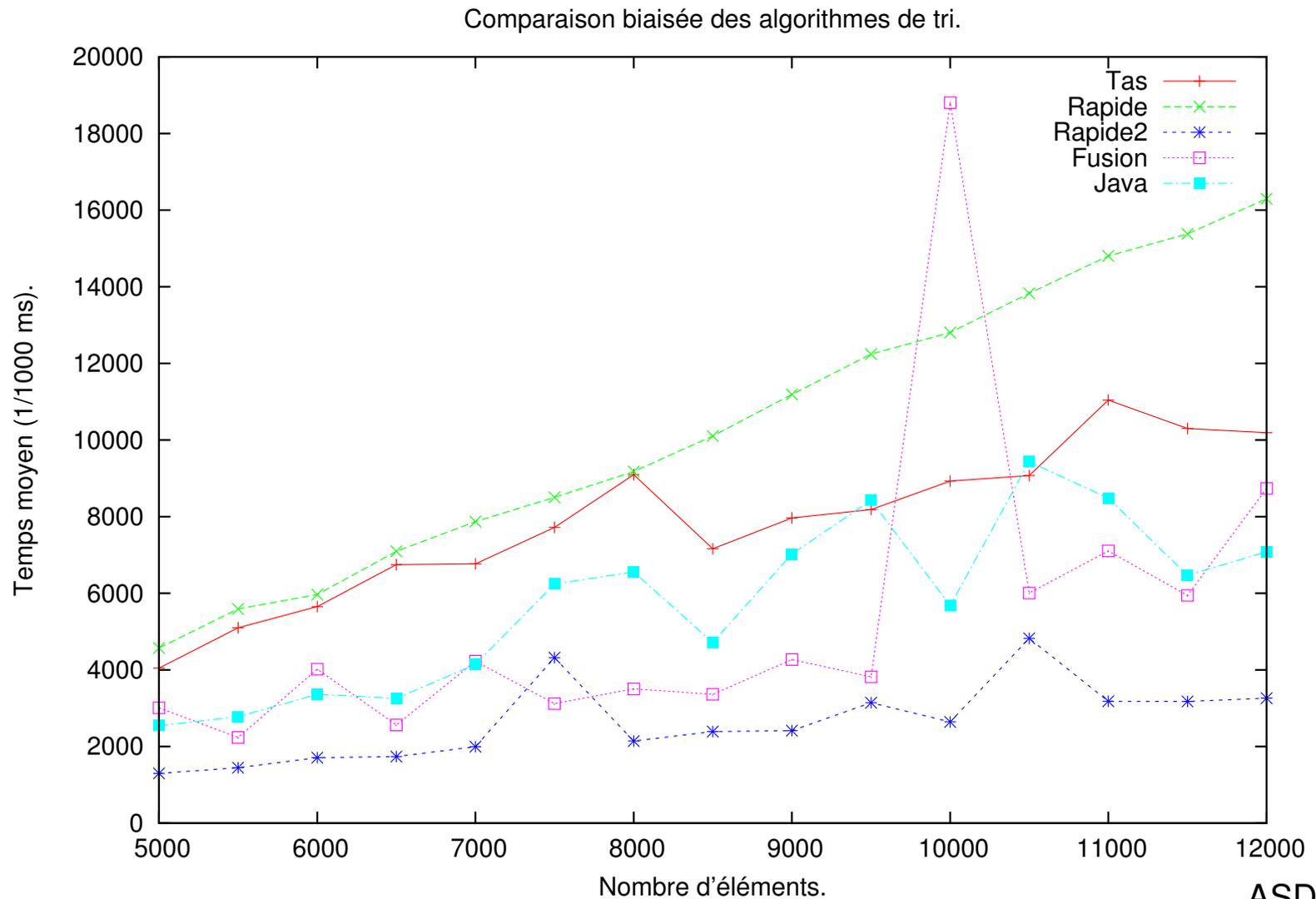
## *Feuille de TD8*

8. Implémentation de la classe `TriFusion`.

# Comparaison des tris



# Comparaison des tris



# Optimalité des tris comparatifs

## Tri comparatif

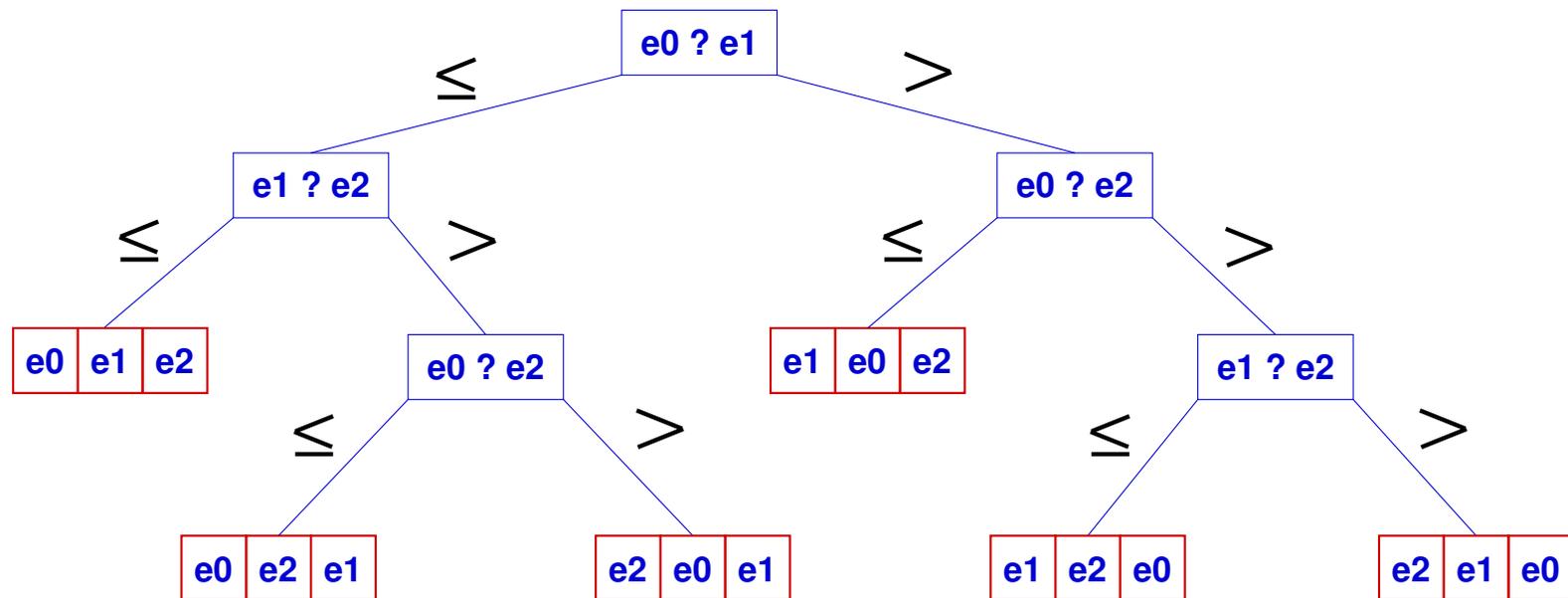
Aucune hypothèse sur les clefs des éléments à trier,  
nécessité de comparer les clefs entre elles

## Algorithme de tri

- une suite de comparaisons de clefs suivant une certaine méthode
- un processus qui transforme un tableau  $[e_0, e_1, \dots, e_{n-1}]$  en un autre tableau  $[e_{\sigma_0}, e_{\sigma_1}, \dots, e_{\sigma_{n-1}}]$  où  $(\sigma_0, \sigma_1, \dots, \sigma_{n-1})$  est une permutation de  $(0, 1, \dots, n - 1)$

# Arbre de décision : exemple

Un algorithme de tri = un arbre binaire de décision



Arbre de décision du tri par insertion du tableau  $[e_0, e_1, e_2]$

# Arbre de décision : exercices



- Dessinez l'arbre de décision du tri par sélection du tableau  $[e_0, e_1, e_2]$ .
- Combien un arbre de décision a-t'il de feuilles ?



# Arbre de décision : définition

**Un algorithme de tri = un arbre binaire de décision**

- **feuille de l'arbre** : une permutation des éléments du tableau initial
- **tri** : le chemin de la racine à la feuille correspondant au tableau trié
- **hauteur de l'arbre** : le pire des cas pour le tri
- **bassesse de l'arbre** : le meilleur des cas pour le tri
- **hauteur moyenne de l'arbre** : la complexité en moyenne du tri

# Arbre de décision : propriétés



- un arbre binaire de hauteur  $h$  a au plus  $2^h$  feuilles (récurrence)
- le nombre de feuilles de l'arbre de décision est  $n!$  où  $n$  est la taille du tableau à trier
- d'où  $n! \leq 2^h \implies \lg(n!) \leq h$ . On a alors :

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \implies n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \lg(n!) > \lg\left(\left(\frac{n}{e}\right)^n\right) = n \lg n - n \lg e \implies h = \Omega(n \lg n)$$

**Un tri comparatif est en  $\Omega(n \lg n)$  dans le pire des cas**



# Tris rapides : synthèse



- un **tri comparatif** ne peut pas avoir une complexité en moyenne ou dans le pire des cas meilleure que  $\Theta(n \lg n)$
- le **tri par tas** est un tri comparatif optimal qui ne dégénère pas (sa complexité est  $\Theta(n \lg n)$  dans tous les cas)
- le **tri rapide** est un tri comparatif optimal qui peut dégénérer mais qui est meilleur (par un petit facteur constant) en moyenne que le tri par tas
- le **tri fusion** est un un tri comparatif optimal qui ne dégénère pas mais qui nécessite un tableau annexe

