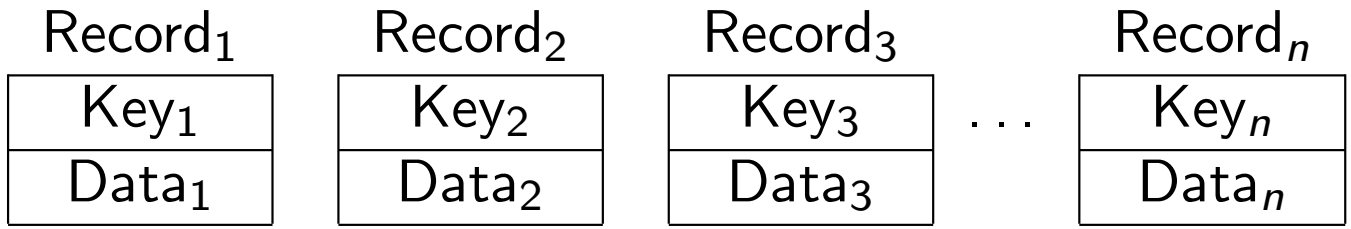
Partie 3

Algorithmes de tri

Plan

Tri

- Un des problèmes algorithmiques les plus fondamentaux.
- En général, on veut trier des enregistrements avec une clé et des données attachées.



- Ici, on va ignorer ces données satellites et se focaliser sur les algorithmes de tri

- Le problème de tri :
  - Entrée : une séquence de n nombres $\langle a_1, a_2, \ldots, a_n \rangle$
  - Sortie : une permutation de la séquence de départ $\langle a'_1, a'_2, \ldots, a'_n \rangle$ telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Applications

Applications innombrables :

- Tri des mails selon leur ancienneté
- Tri des résultats de requête dans un moteur de recherche
- Tri des facettes des objets pour l'affichage dans les jeux 3D
- Gestion des opérations bancaires
- ...

Le tri sert aussi de brique de base pour d'autres algorithmes :

- Recherche binaire dans un tableau trié
- Recherche des éléments dupliqués dans une liste
- Recherche du $k$ème élément le plus grand dans une liste
- ...

Des études montrent qu'environ 25% du temps CPU des ordinateurs est utilisé pour trier

Différents types de tri

- **Tri interne :** tri en mémoire centrale. **Tris externes :** données sur un disque externe.
- **Tri de tableau :** tri qui trie un tableau. Extensible à toutes structures de données offrant un accès en temps (quasi) constant à ses éléments.
- **Tri générique :** peut trier n'importe quel type d'objets pour autant qu'on puisse comparer ces objets.
- **Tri comparatif :** basé sur la comparaison entre les éléments (clés)

Différents types de tri

- **Tri itératif :** basé sur un ou plusieurs parcours itératifs du tableau
- **Tri récursif :** basé sur une procédure récursive
- **Tri en place :** modifie directement la structure qu'il est en train de trier. Ne nécessite qu'une quantité très limitée de mémoire supplémentaire.
- **Tri stable :** conserve l'ordre relatif des éléments égaux (au sens de la méthode de comparaison).

Jusqu'ici

Algorithme	*Complexité*			*En place ?*
	Pire	*Moyenne*	*Meilleure*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui
Bubble-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	non
??		$\Theta(n \log n)$		oui
??	$\Theta(n \log n)$			oui

Tri rapide

- *Quicksort* en anglais
- Inventé par Hoare en 1960
- Dans le top 10 des algorithmes du 20-ième siècle (SIAM)
- L'exemple le plus célèbre de la technique du "diviser pour régner"
- Tri en place, comme tri par insertion, et contrairement au tri par fusion
- Complexité : $\Theta(n^2)$ dans le pire des cas, $\Theta(n \log n)$ en moyenne

QuickSort : principe

Pour trier un sous-tableau $A[p\,..\,r]$:

- Partitionner $A[p\,..\,r]$ en deux sous-tableaux : $A[p..q-1]$ et $A[q+1\,..\,r]$ tels que tout élément de $A[p\,..\,q-1]$ est $\leq A[q]$ et $A[q] <$ à tout élément de $A[q+1\,..\,r]$.

  (diviser)

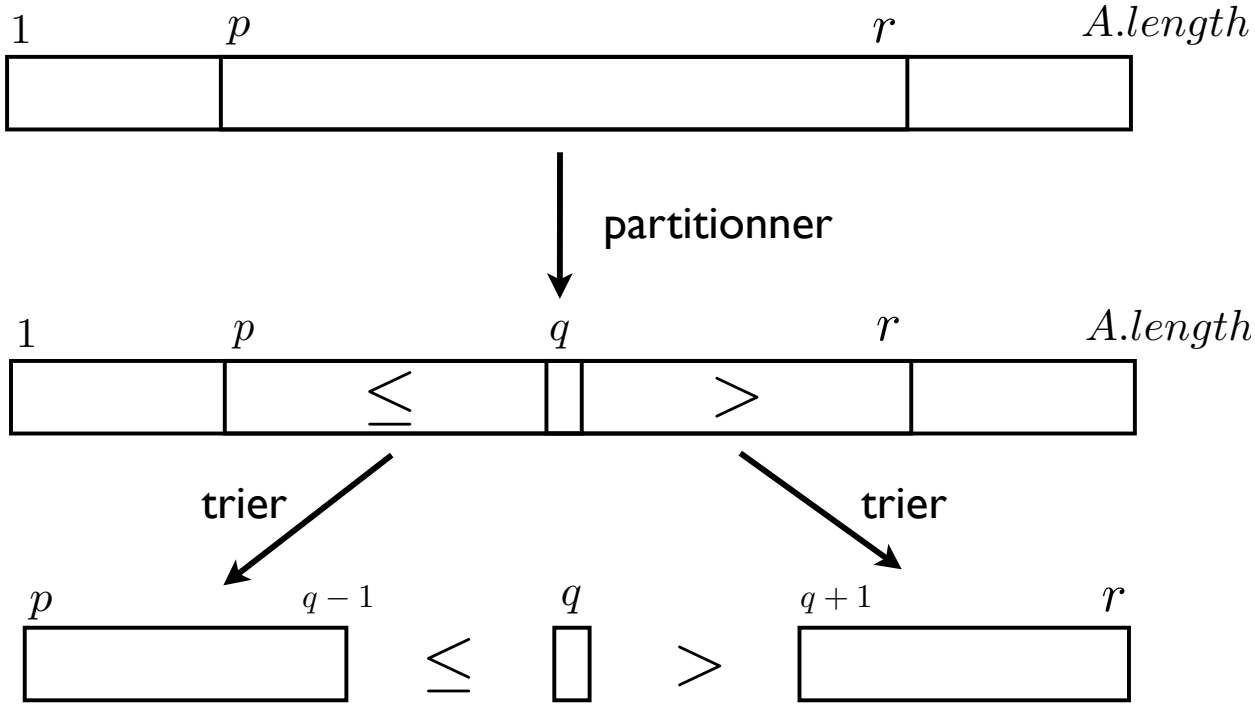
- Appeler récursivement l'algorithme pour trier $A[p\,..\,q-1]$ et $A[q+1\,..\,r]$

  (régner)

Remarques :

- $A[q]$ est appelé le "pivot"
- Par rapport au tri par fusion, il n'y a pas d'opération de combinaison

QuickSort : principe

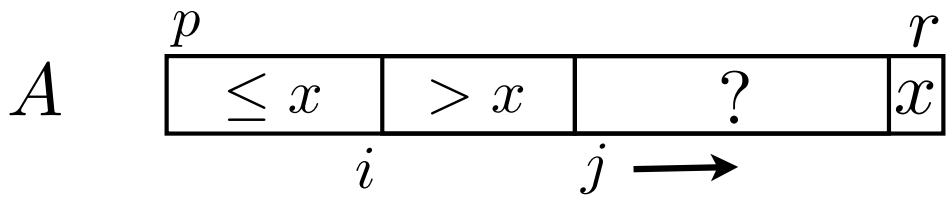


QuickSort Algorithm

```
QuickSort(A, p, r)
1  if p < r
2        q = Partition(A, p, r)
3        QuickSort(A, p, q − 1)
4        QuickSort(A, q + 1, r)
```

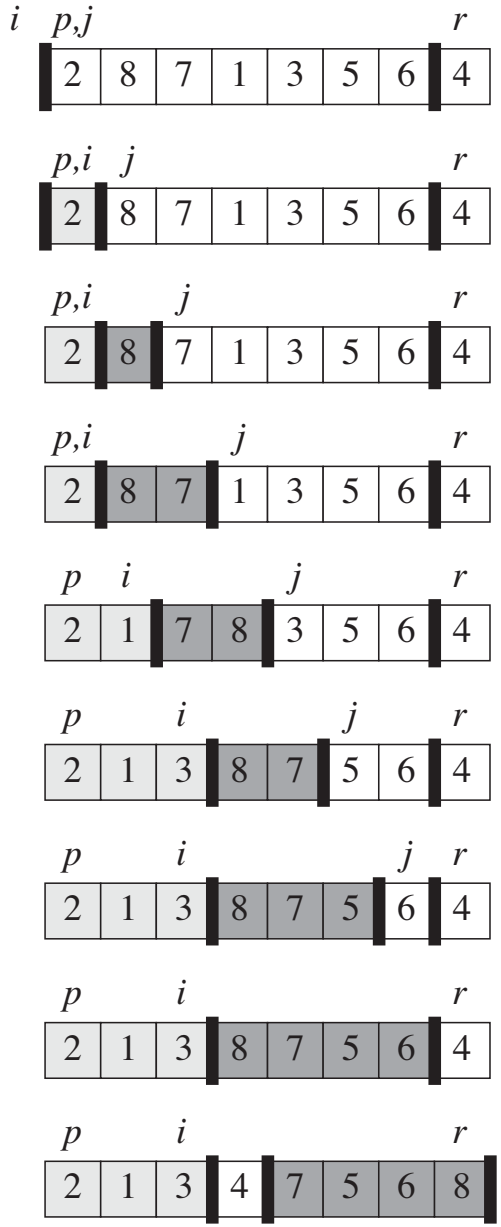
Appel initial : Quicksort$(A, 1, A.length)$

Partition : principe



- On sélectionne le dernier élément $A[r]$ comme le **pivot**
- On initialise un indice i à $p - 1$
- On parcourt le tableau de gauche à droite avec un indice $j = p \textbf{ to } r - 1$
- Si $A[j] \leq A[r]$, on incrémente i et on échange $A[j]$ et $A[i]$
- En sortie de boucle, on échange $A[i + 1]$ et $A[r]$ et on renvoie $i + 1$

Partition : illustration



$A[r]$ est le pivot
$A[p\,..\,i]$ contient des éléments $\leq$ au pivot
$A[i+1\,..\,j-1]$ contient des éléments $>$ que le pivot
$A[j\,..\,r-1]$ est la partie du tableau non encore examinée

Partition : pseudo-code

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

Partition : correction

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

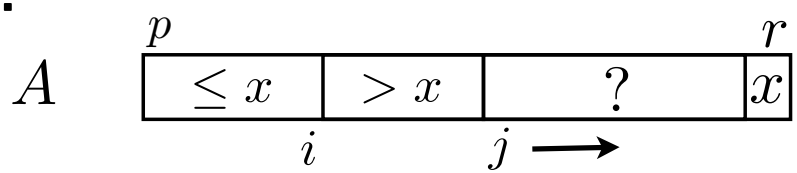
Pré-condition : $\{A[p\,..\,r],$ un tableau de nombres$\}$
Post-condition : $\{A[p\,..\,i] \leq A[i+1] < A[i+2\,..\,r]\}$
Invariant :

1. Les valeurs dans $A[p\,..\,i]$ sont $\leq$ au pivot
2. Les valeurs dans $A[i+1\,..\,j-1]$ sont $>$ que le pivot
3. $A[r] = pivot$

Partition : correction

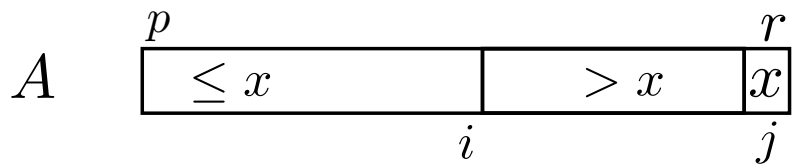
Avant la boucle : $i = p - 1$ et $j = p \Rightarrow A[p\,..\,i]$ et $A[i+1\,..\,j-1]$ sont vides

Pendant la boucle :



- Si $A[j] > x$, on incrémente juste j. Donc si l'invariant était vrai avant l'exécution du corps, il reste vrai après.
- Si $A[j] \leq x$, on échange $A[j]$ et $A[i+1]$ et i et j sont incrémentés. L'invariant reste donc vérifié également

Après la boucle :



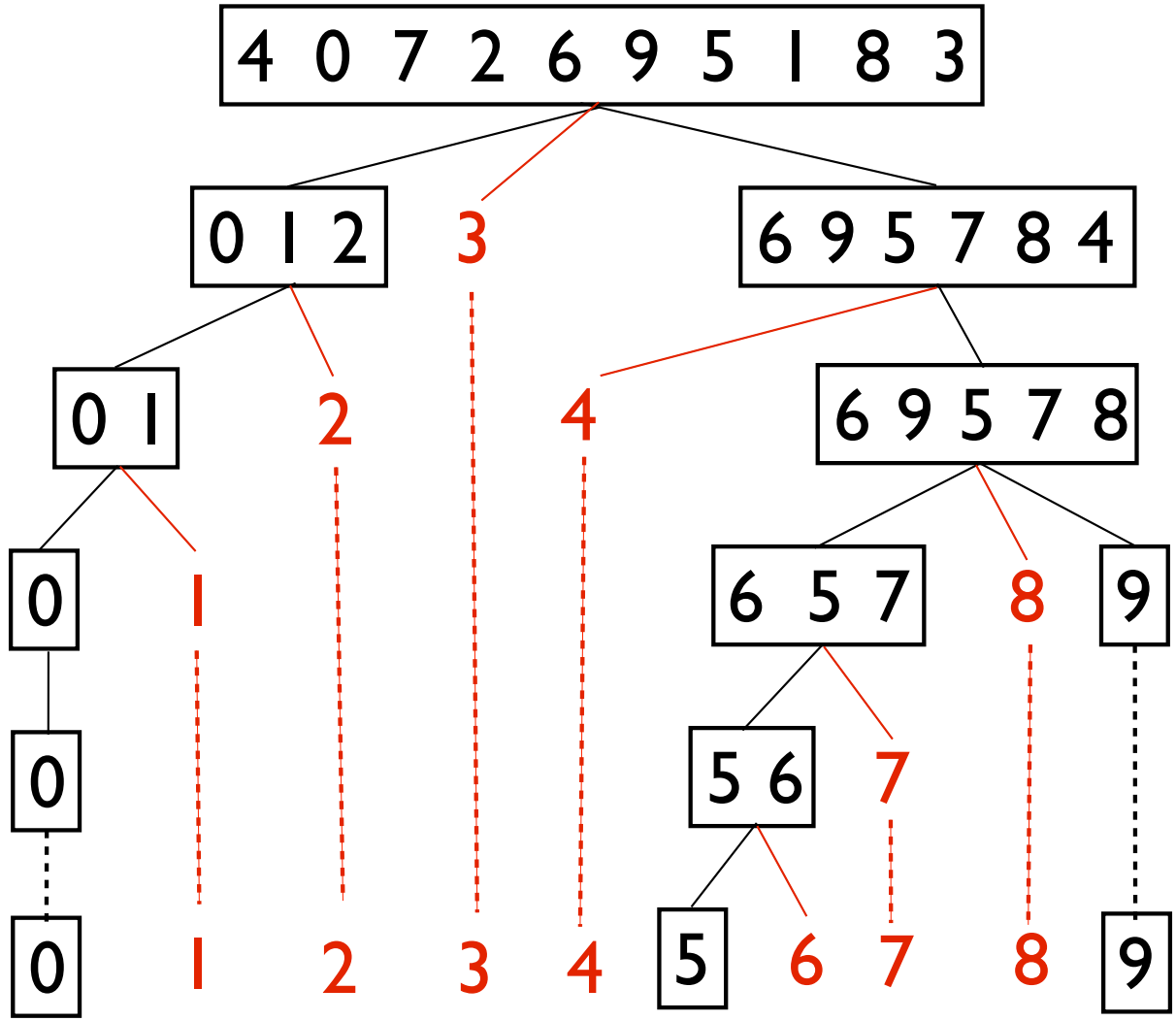
En sortie de boucle, l'invariant est vérifié et on a $j = r$. Echanger $A[i+1]$ et $A[r]$ établit la post-condition.

Algorithme complet

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

Illustration



Complexité de Partition

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

$$T(n) = \Theta(n)$$

Complexité de QuickSort

```
QuickSort(A, p, r)
1  if p < r
2        q = Partition(A, p, r)
3        QuickSort(A, p, q − 1)
4        QuickSort(A, q + 1, r)
```

- Pire cas : *(quand se produit-il ?)*
  - $q = p$ ou $q = r$
  - Le partitionnement transforme un problème de taille n en un problème de taille $n - 1$

$$T(n) = T(n-1) + \Theta(n)$$

  - Même complexité que le tri par insertion :

$$T(n) = \Theta(n^2)$$

Complexité de QuickSort

```
QuickSort(A, begin, end)
1  if begin < end
2        q = Partition(A, begin, end)
3        QuickSort(A, begin, q − 1)
4        QuickSort(A, q + 1, end)
```

- Meilleur cas :
  - $q = \lfloor n/2 \rfloor$
  - Le partitionnement transforme un problème de taille n en deux problèmes de taille $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor - 1$ respectivement

$$T(n) = 2T(n/2) + \Theta(n)$$

  - Même complexité que le tri par fusion :

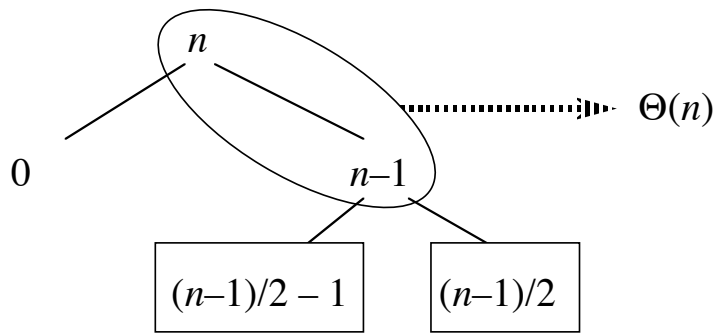
$$T(n) = \Theta(n \log n)$$

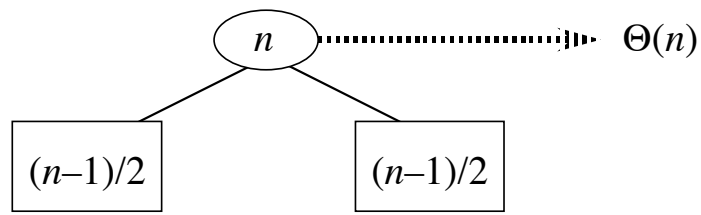
Complexité moyenne de QuickSort : intuitivement

- Complexité moyenne identique à la complexité du meilleur cas

$$T(n) = \Theta(n \log n)$$

- Intuitivement :
  - En moyenne, on s'attend à une alternance de "bons" et de "mauvais" partitionnements
  - La complexité d'un mauvais partitionnement suivi d'un bon est identique à la complexité d'une bon partitionnement directement (seule la constante est modifiée).





Complexité moyenne de QuickSort : mathématiquement

Modèle mathématique :

- Nombre de comparaisons pour le partitionnement : $n+1$
- Probabilité que le pivot soit à la position k : $1/n$
- Tailles des sous-tableaux dans ce cas-là : $k-1$ et $n-k$
- Les sous-tableaux sont aussi triés aléatoirement

Le nombre *moyen* de comparaisons utilisées par le quicksort est donné par la récurrence suivante :

$$\begin{aligned} C_1 &= 0 \\ C_n &= n-1+\sum_{k=1}^{n}\frac{1}{n}(C_{k-1}+C_{n-k}) \text{ (si } n>1) \end{aligned}$$

Formulation analytique

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

Par symétrie :

$$C_n = n - 1 + \frac{2}{n} \sum_{k=1}^{n} C_{k-1}$$

En multipliant par n :

$$nC_n = n(n-1) + 2 \sum_{k=1}^{n} C_{k-1}$$

En soustrayant la même formule pour $n-1$:

$$nC_n - (n-1)C_{n-1} = 2(n-1) + 2C_{n-1}$$

En rassemblant les termes :

$$nC_n = (n+1)C_{n-1} + 2(n-1)$$

On divise par $n(n+1)$:

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2(n-1)}{n(n+1)}$$

Téléscopage :

$$\begin{aligned}
\frac{C_n}{n+1} &= \frac{C_{n-1}}{n} + \frac{2(n-1)}{n(n+1)} = \frac{C_{n-2}}{n-1} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\
&= \frac{C_1}{2} + \frac{2\cdot 1}{2\cdot 3} + \cdots + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\
&= \sum_{k=2}^{n} \frac{2(k-1)}{k(k+1)} = \sum_{k=2}^{n} \frac{2}{(k+1)} - \sum_{k=2}^{n} \frac{2}{k(k+1)}
\end{aligned}$$

En négligeant la deuxième somme devant la première, on obtient :

$$C_n = 2(n+1)\sum_{k=1}^{n} \frac{1}{k} - 3(n+1)$$

En utilisant l'approximation de la série harmonique du transparent 128 :

$$C_n \sim 2n \ln n \in \Theta(n \log n),$$

Variantes de Quicksort

- Choix du pivot :
  - Prendre un élément au hasard plutôt que le dernier.
  - Prendre la médiane de 3 éléments
  - Diminue drastiquement les chances d'être dans le pire cas

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  swap(A[end], A[i])
3  return PARTITION(A, p, r)
```

```
MEDIAN-OF-3-PARTITION(A, p, r)
1  i = MEDIAN(A, p, ⌊(p + r)/2⌋, r)
2  swap(A[r], A[i])
3  return PARTITION(A, p, r)
```

Variantes de QUICKSORT

- Petits sous-tableaux
  - QUICKSORT est trop lourd pour des petits tableaux
  - Utiliser un tri naïf (par ex., par insertion) sur les sous-tableaux de longueur inférieure à k ($k \approx 20$).

```
QUICKSORT(A, p, r)
1  if r − p + 1 ≤ CUTOFF
2        INSERTIONSORT(A, p, r)
3        return
4  q = PARTITION(A, p, r)
5  QUICKSORT(A, p, q − 1)
6  QUICKSORT(A, q + 1, r)
```

Conclusion sur QuickSort

- Rapide en moyenne $\Theta(n \log n)$
- Pire cas en $\Theta(n^2)$ mais très improbable avec choix du pivot bien fait
- Bonne performance au niveau du cache
- Tri en place (mais utilise de la mémoire pour la trace récursive)
- Pas stable
- En pratique souvent un peu plus rapide que Merge-Sort
- Complexité en espace $O(\log n)$ si bien implémenté (récursif terminal, en développant d'abord la partition la plus petite)

Jusqu'ici

Algorithme	*Complexité*			*En place ?*
	Pire	*Moyenne*	*Meilleure*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui
Bubble-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	non
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	oui
??	$\Theta(n \log n)$			oui

Tri par tas : introduction

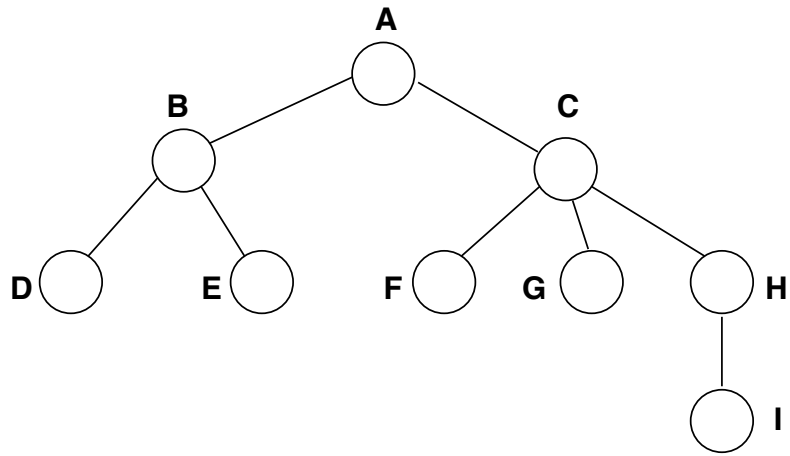
- *Heapsort* en anglais
- inventé par Williams en 1964
- basé sur une structure de données très utile, le *tas*
- complexité bornée par $\Theta(n \log n)$ (dans tous les cas)
- tri en place
- mise en oeuvre très simple

- Suite du cours :
  - Introduction aux arbres
  - Tas
  - Tri par tas

Arbres : définition

- Définition : Un arbre[2] (*tree*) T est un graphe dirigé (N, E), où :
  - N est un ensemble de nœuds, et
  - $E \subset N \times N$ est un ensemble d'arcs,

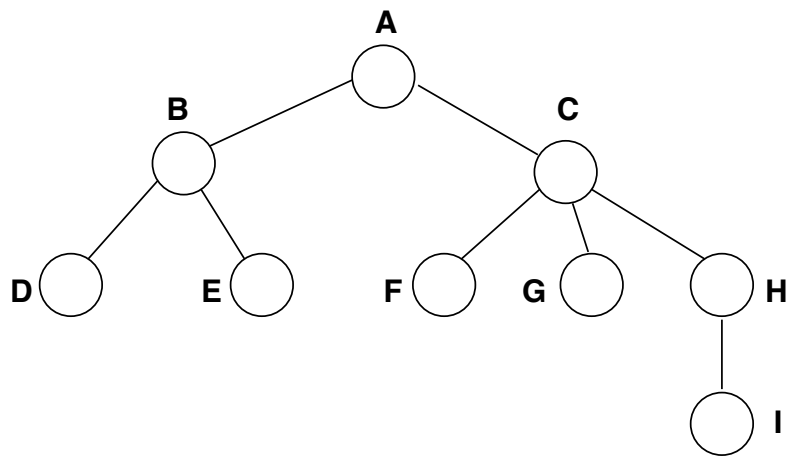
  possédant les propriétés suivantes :
  - T est connexe et acyclique
  - Si T n'est pas vide, alors il possède un nœud distingué appelé racine (*root node*). Cette racine est unique.
  - Pour tout arc $(n_1, n_2) \in E$, le nœud n_1 est le parent de n_2.
    - La racine de T ne possède pas de parent.
    - Les autres nœuds de T possèdent un et un seul parent.



2. En théorie des graphes, on parlera d'un arbre enraciné.

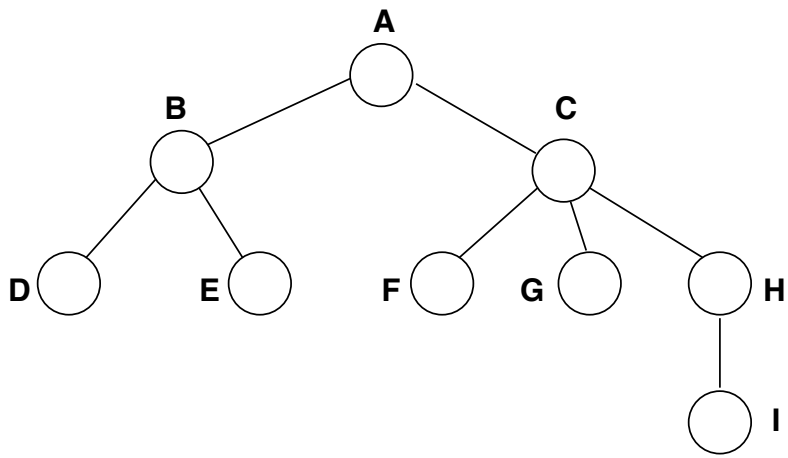
Arbres : terminologie

- Si n_2 est le parent de n_1, alors n_1 est le fils (*child*) de n_2.
- Deux nœuds n_1 et n_2 qui possèdent le même parent sont des frères (*siblings*).
- Un nœud qui possède au moins un fils est un nœud interne.
- Un nœud externe (c'est-à-dire, non interne) est une feuille (*leaf*) de l'arbre.
- Un nœud n_2 est un ancêtre (*ancestor*) d'un nœud n_1 si n_2 est le parent de n_1 ou un ancêtre du parent de n_1.
- Un nœud n_2 est un descendant d'un nœud n_1 si n_1 est un ancêtre de n_2.



Arbres : terminologie

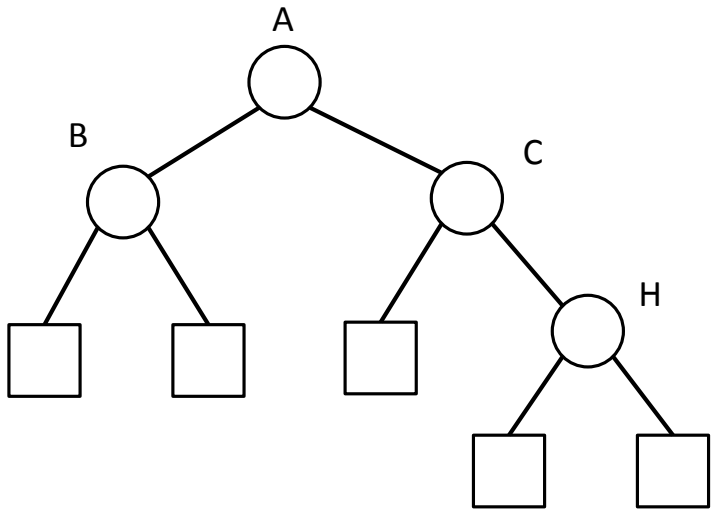
- Un chemin est une séquence de nœuds $n_1, n_2, \ldots, n_m$ telle que pour tout $i \in [1, m-1]$, (n_i, n_{i+1}) est un arc de l'arbre.
  Remarque : Il n'existe jamais de chemin reliant deux feuilles distinctes.
- La hauteur (*height*) d'un nœud n est le nombre d'arcs d'un plus long chemin de ce nœud vers une feuille. La *hauteur de l'arbre* est la hauteur de sa racine.
- La profondeur (*depth*) d'un nœud n est le nombre d'arcs sur le chemin qui le relie à la racine.



Arbre binaire

- Un arbre ordonné est un arbre dans lequel les ensembles de fils de chacun de ses nœuds sont ordonnés.
- Un arbre binaire est un arbre ordonné possédant les propriétés suivantes :
  - Chacun de ses nœuds possède au plus deux fils.
  - Chaque nœud fils est soit un fils gauche, soit un fils droit.
  - Le fils gauche précède le fils droit dans l'ordre des fils d'un nœud.
- Un arbre binaire entier ou propre (*full* or *proper*) est un arbre binaire dans lequel tous les nœuds internes possèdent exactement deux fils.
- Un arbre binaire parfait est un arbre binaire entier dans lequel toutes les feuilles sont à la même profondeur.

Propriétés des arbres binaires entiers



- Le nombre de nœuds externes est égal au nombre de nœuds internes plus 1.
- Le nombre de nœuds internes est égal à $\frac{n-1}{2}$, où n désigne le nombre de nœuds.
- Le nombre de nœuds à la profondeur (ou niveau) i est $\leq 2^i$.
- La hauteur h de l'arbre est $\leq$ au nombre de nœuds internes.
- Le lien entre hauteur et nombre de nœuds peut être résumé comme suit :

$$n \in \Omega(h) \text{ et } n \in O(2^h) \text{ (ou } h \in O(n) \text{ et } h \in \Omega(\log n))$$

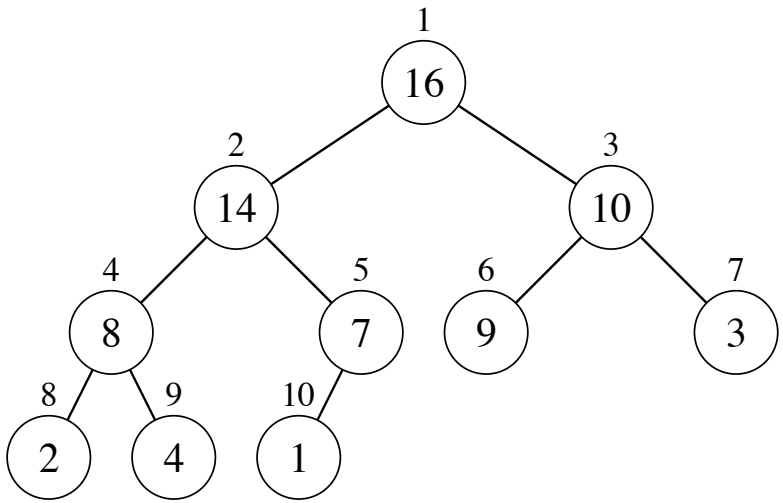
Tas : définition

Un arbre binaire complet est un arbre binaire tel que :

- Si h dénote la hauteur de l'arbre :
  - Pour tout $i \in [0, h-1]$, il y a exactement 2^i nœuds à la profondeur i.
  - Une feuille a une profondeur h ou $h-1$.
  - Les feuilles de profondeur maximale (h) sont "tassées" sur la gauche.

Un tas binaire (binary heap) est un arbre binaire complet tel que :

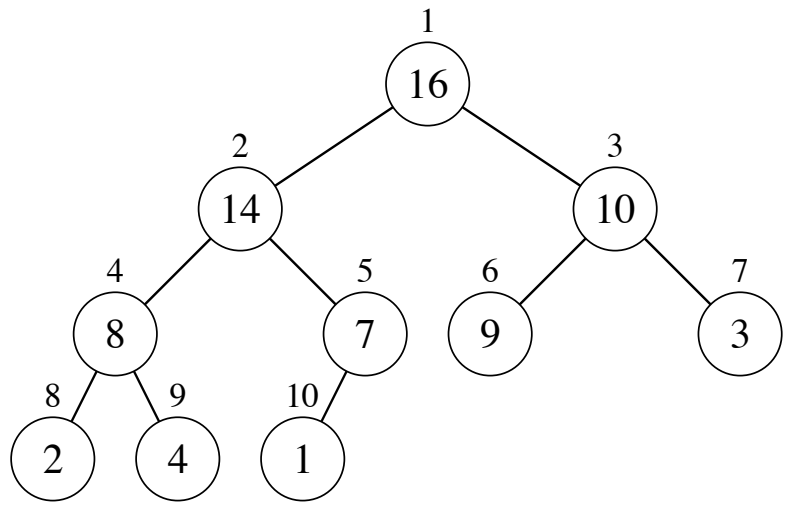
- Chacun de ses nœuds est associé à une clé.
- La clé de chaque nœud est supérieure ou égale à celle de ses fils (propriété d'ordre du tas).



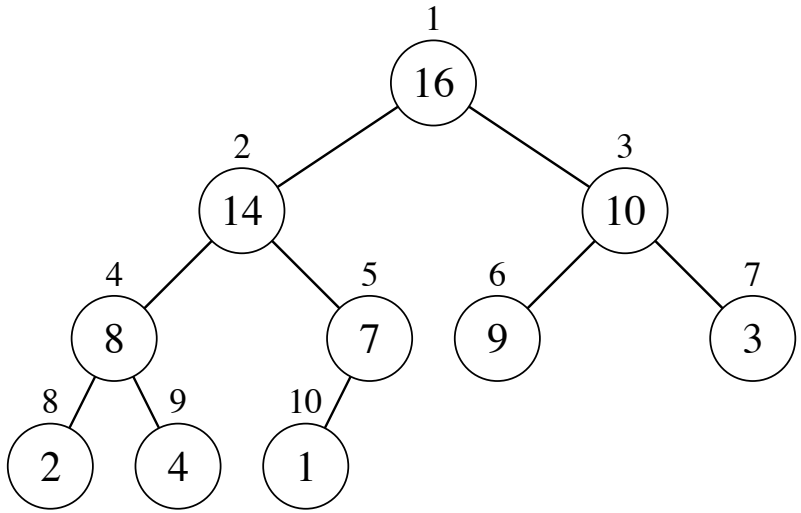
Propriété d'un tas

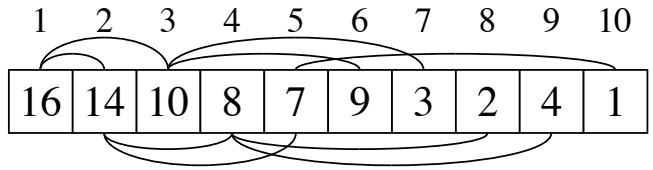
- Soit T un arbre binaire complet contenant n entrées et de hauteur h :
  - n est supérieur ou égal à la taille de l'arbre parfait de hauteur $h-1$ plus un, soit $2^{h-1+1} - 1 + 1 = 2^h$
  - n est inférieur ou égal à la taille de l'arbre parfait de hauteur h, soit $2^{h+1} - 1$

$$
\begin{aligned}
2^h \leq n \leq 2^{h+1} - 1 \quad &\Leftrightarrow \quad 2^h \leq n < 2^{h+1} \\
&\Leftrightarrow \quad h \leq \log_2 n < h + 1 \\
&\Leftrightarrow \quad h = \lfloor \log_2 n \rfloor
\end{aligned}
$$



Implémentation par un tableau





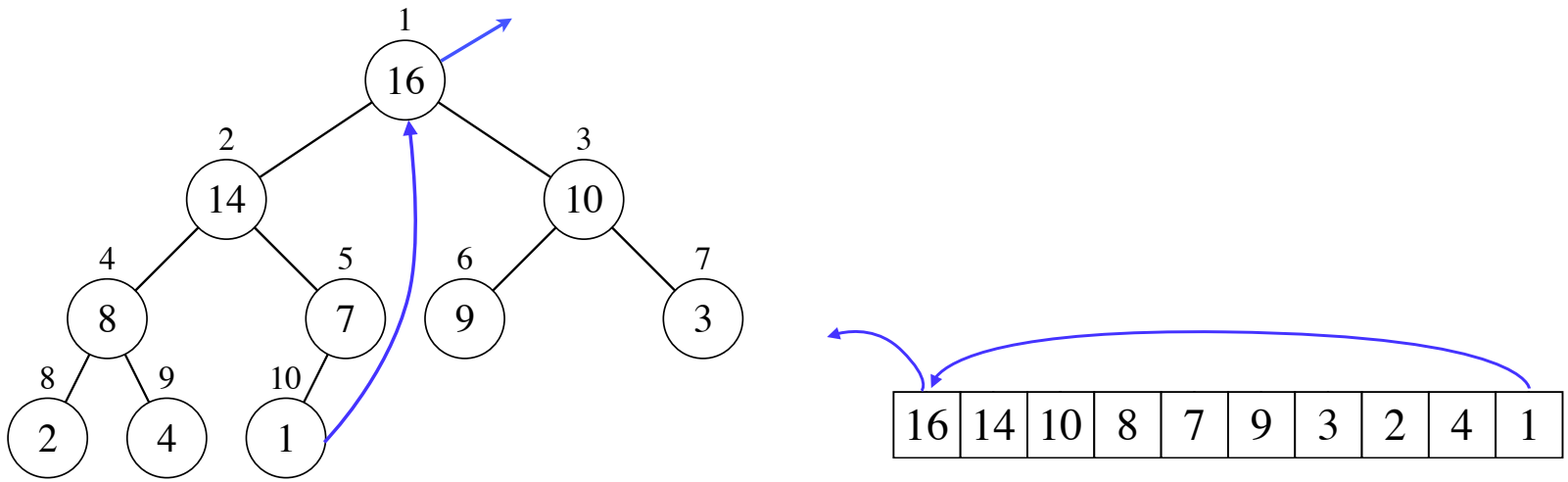
Un tas peut être représenté de manière compacte à l'aide d'un tableau A.

- La racine de l'arbre est le premier élément du tableau.
- $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
- $\textsc{Left}(i) = 2i$
- $\textsc{Right}(i) = 2i + 1$

Propriété d'ordre du tas : $\forall i, A[\textsc{Parent}(i)] \geq A[i]$

Principe du tri par tas

- On construit un tas à partir du tableau à trier → BUILD-MAX-HEAP(A).
- Tant que le tas contient des éléments :
  - On extrait l'élément au sommet du tas qu'on place dans le tableau trié et on le remplace par l'élément le plus à droite

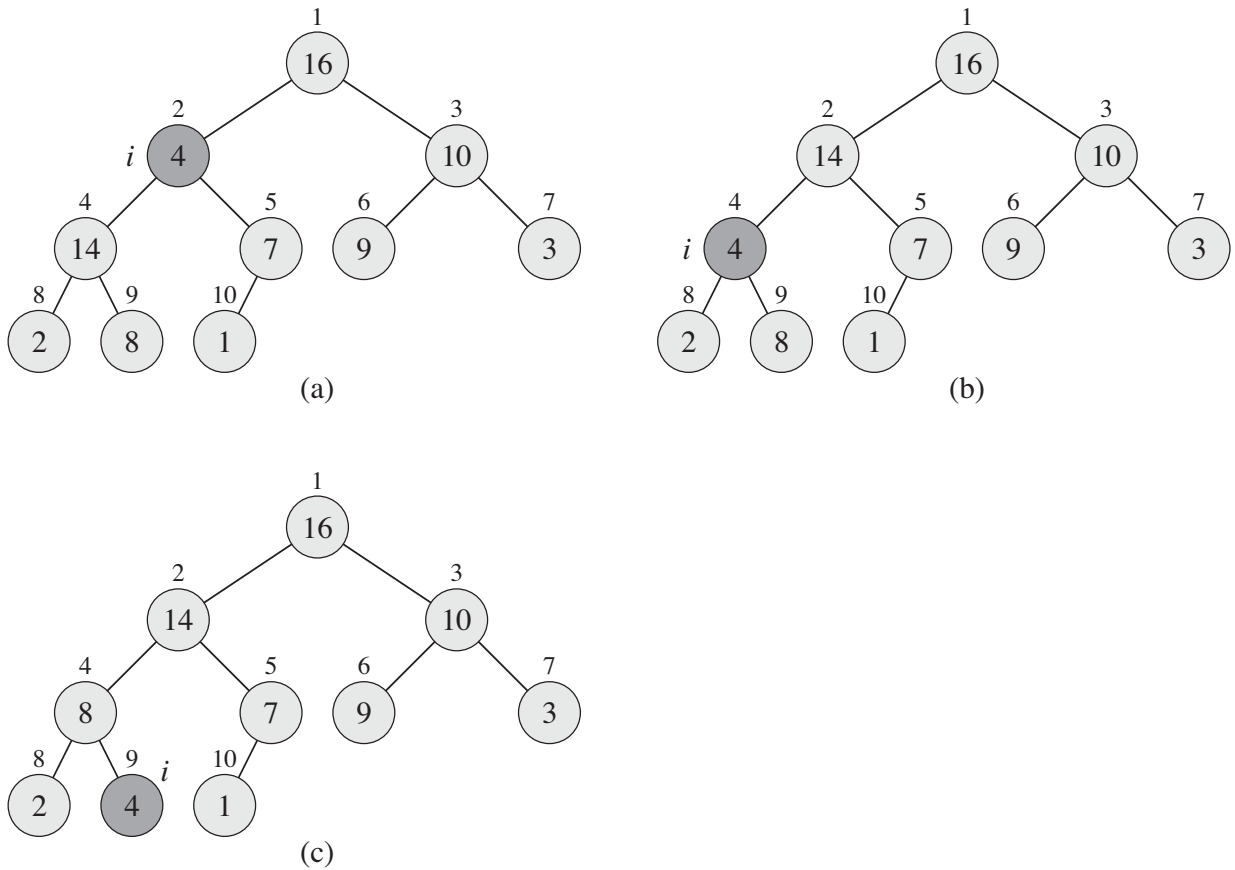


  - On rétablit la propriété de tas en tenant compte du fait que les sous-arbres de droite et de gauche sont des tas → MAX-HEAPIFY($A, 1$)

(Tout se fait dans le tableau initial → en place)

Max-Heapify

- Procédure Max-Heapify(A, i) :
  - Suppose que le sous-arbre de gauche du nœud i est un tas
  - Suppose que le sous-arbre de droite du nœud i est un tas
  - But : réarranger le tas pour maintenir la propriété d'ordre du tas
- Ex : Max-Heapify($A, 2$)



Max-Heapify

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size ∧ A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size ∧ A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap(A[i], A[largest])
10        MAX-HEAPIFY(A, largest)
```

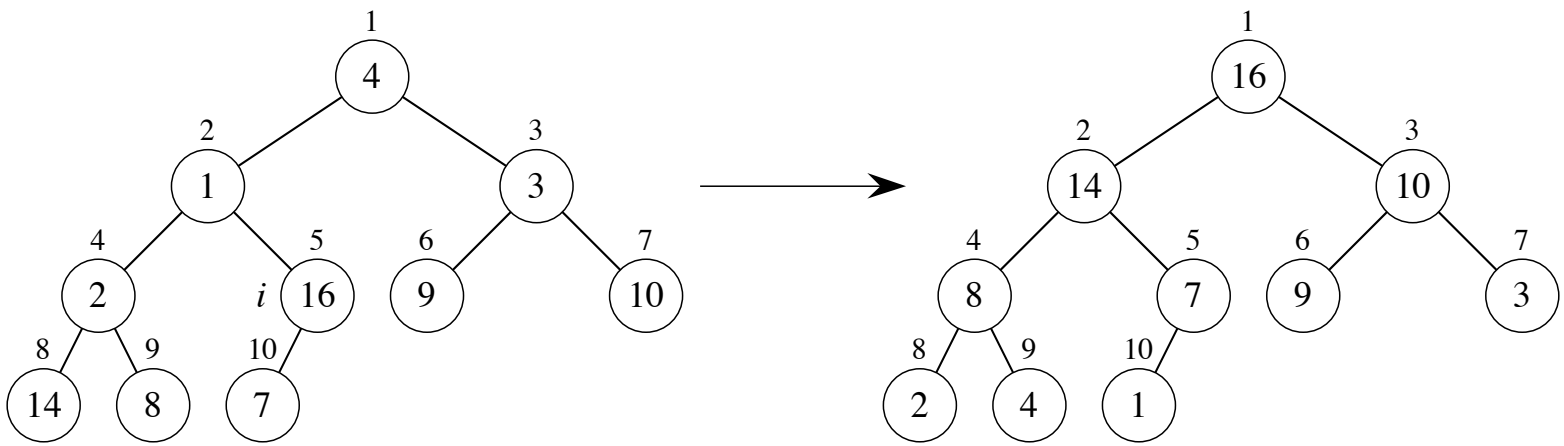
- Complexité ? La hauteur du nœud : $T(n) = O(\log n)$ (Transp. 193)

Construction d'un tas

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

(Invariant : chaque nœud i, i+1,. . . , n est la racine d'un tas)

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

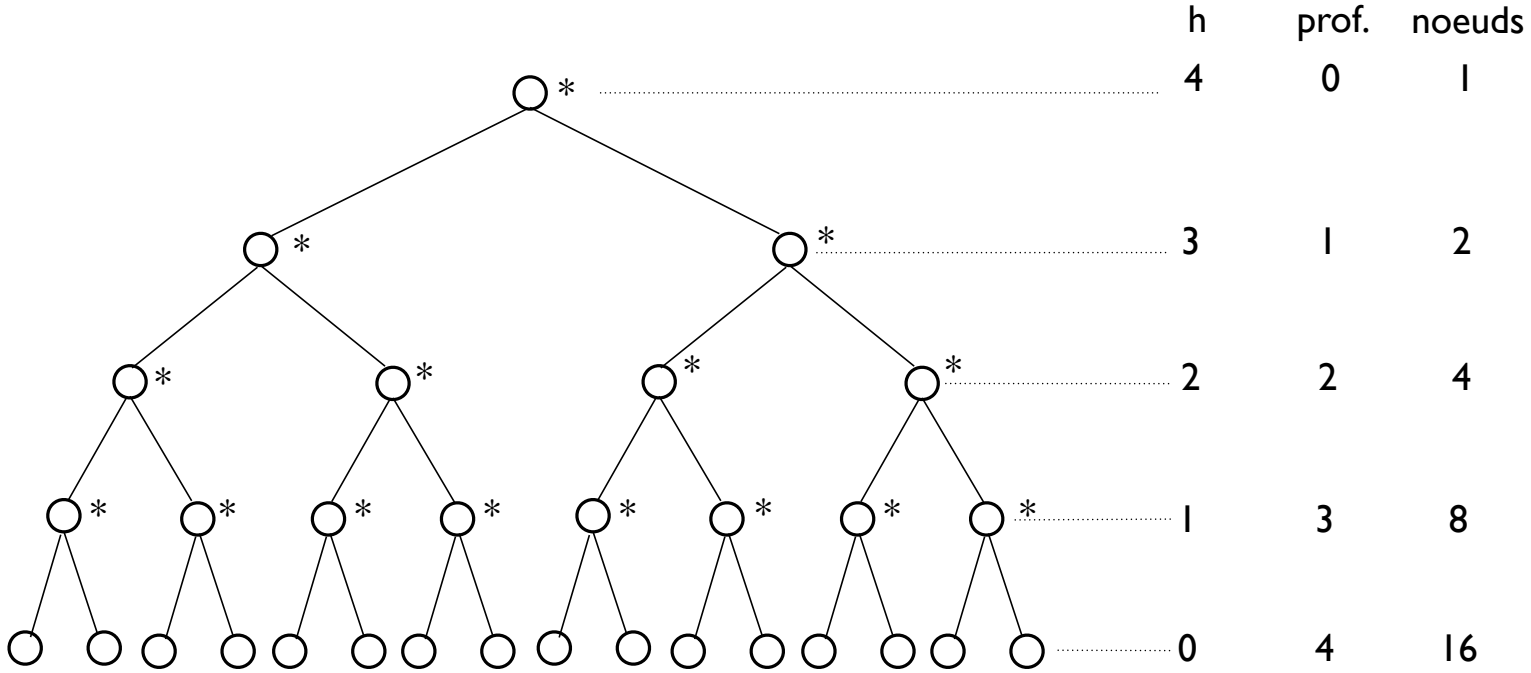


- La tableau initial est interprété comme un arbre binaire complet
- On tasse les nœuds internes de bas en haut et de droite à gauche

Complexité de Build-Max-Heap

- Borne simple :
  - $O(n)$ appels à Max-Heapify, chacun étant $O(\log n) \Rightarrow O(n \log n)$.
- Analyse plus fine :
  - Pour simplifier l'analyse, on suppose que l'arbre binaire est parfait.
  - On a donc $n = 2^{h+1} - 1$ pour un $h \geq 0$, qui est aussi la hauteur de l'arbre résultant

Complexité de BUILD-MAX-HEAP



* Noeuds sur lesquels on doit appeler Max-Heapify

- Il y a 2^i nœuds à la profondeur i ($=$ hauteur $h - i$).
- On doit appeler MAX-HEAPIFY sur chacun d'eux
- Chaque appel est au pire $\Theta(h - i)$.
- Nombre d'opérations en fonction de h au pire cas :

$$T(h) = \sum_{i=0}^{h-1} 2^i \Theta(h-i) = \Theta(\sum_{i=0}^{h-1} 2^i (h-i))$$

Complexité de Build-Max-Heap

- En utilisant les théorèmes des transparents 120 et 123, on obtient :

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1,$$

$$\sum_{i=0}^{n} i2^i = (n-1)2^{n+1} + 2.$$

D'où on tire que :

$$\begin{aligned} T(h) &\in \Theta(h(2^h - 1) - (h-2)2^h - 2) \\ &= \Theta(2^{h+1} - h - 2) \end{aligned}$$

- Puisque $h \in \Theta(\log n)$ pour un tas (transp. 192), on a (au pire cas) :

$$T(n) \in \Theta(n).$$

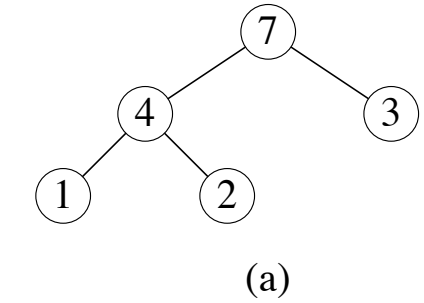
Tri par tas : algorithme

```
HEAP-SORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3        swap(A[i], A[1])
4        A.heap-size = A.heap-size − 1
5        MAX-HEAPIFY(A, 1)
```

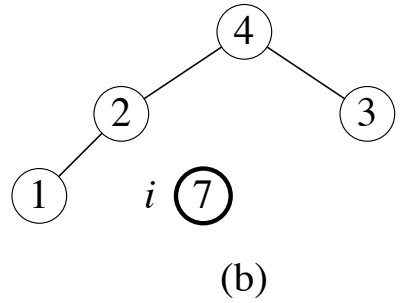
Invariant :
$A[1\,..\,i]$ est un tas contenant les i éléments les plus petits de $A[1\,..\,A.length]$ et $A[i+1\,..\,A.length]$ contient les $n-i$ éléments les plus grands de $A[1\,..\,A.length]$ triés.

Tri par tas : illustration

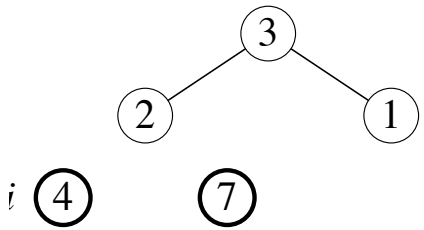
Tableau initial : $A = [7, 4, 3, 1, 2]$



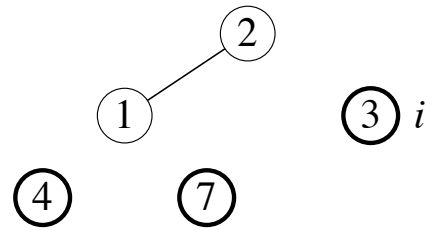
(a)



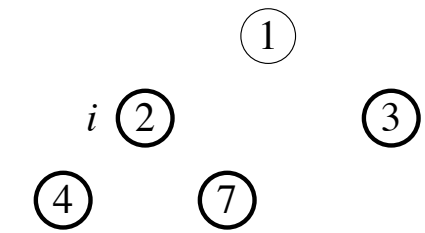
(b)



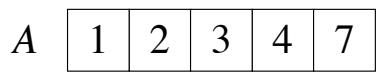
(c)



(d)



(e)



Complexité de Heap-Sort

```
Heap-Sort(A)
1  Build-Max-Heap(A)
2  for i = A.length downto 2
3      swap(A[i], A[1])
4      A.heap-size = A.heap-size − 1
5      Max-Heapify(A, 1)
```

- Build-Max-Heap : $O(n)$
- Boucle **for** : $n - 1$ fois
- Echange d'éléments : $O(1)$
- Max-Heapify : $O(\log n)$

Total : $O(n \log n)$ (pour le pire cas et le cas moyen)

Le tri par tas est cependant généralement battu par le tri rapide

Résumé

Algorithme	*Complexité*			*En place ?*
	Pire	*Moyenne*	*Meilleure*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui
Bubble-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	non
Quick-Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	oui
Heap-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)^*$	oui

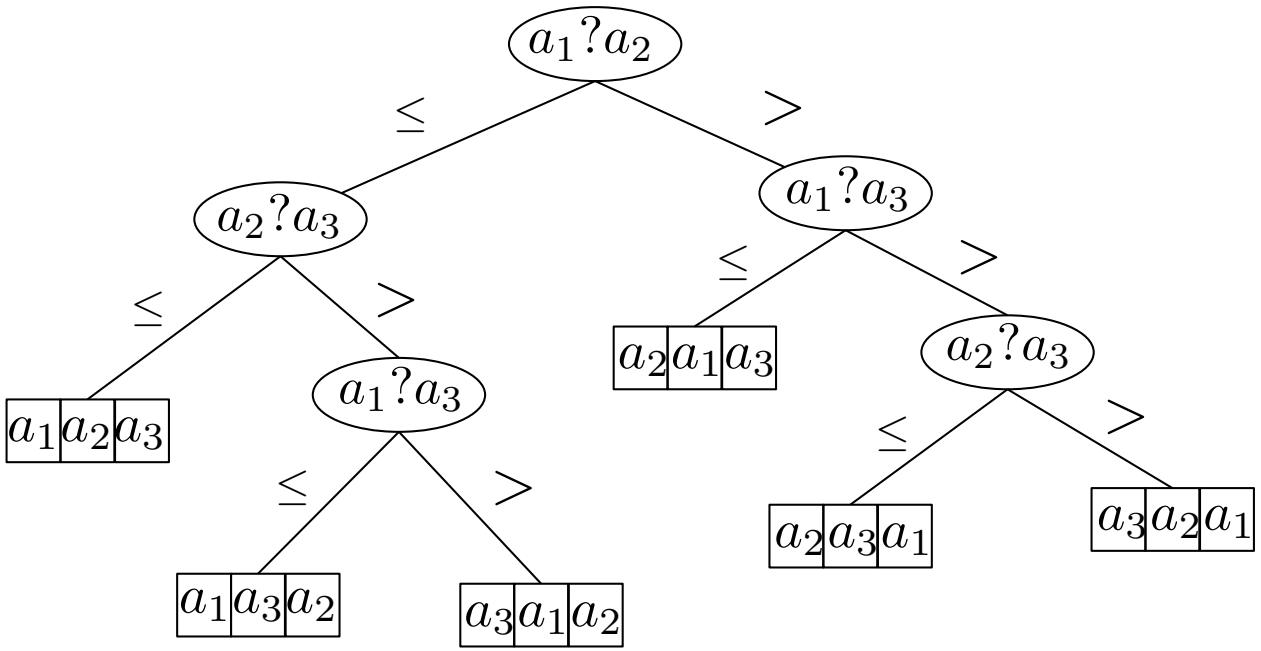
* pas montré dans ce cours

Peut-on faire mieux que $O(n \log n)$?

- Non, si on se restreint aux tri *comparatifs*, c'est-à-dire :
  - Aucune hypothèse sur les éléments à trier
  - Nécessité de les comparer entre eux
- Complexité d'un problème algorithmique versus complexité d'un algorithme
- Dans ce cas, un algorithme de tri est :
  - Une suite de comparaisons d'éléments suivant une certaine méthode
  - Un processus qui transforme un tableau $[e_0, e_1, \ldots, e_{n-1}]$ en un autre tableau $[e_{\sigma_0}, e_{\sigma_1}, \ldots, e_{\sigma_{n-1}}]$ où $(\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ est une permutation de $(0, 1, \ldots, n-1)$.

Arbre de décision : exemple

Un algorithme de tri = un arbre binaire de décision (entier)



(arbre de décision pour le tri par insertion du tableau $[e_0, e_1, e_2]$)

Exercice : construire l'arbre pour le tri par fusion

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 cas pour le tri
- branche la plus courte : le meilleur cas pour le tri
- hauteur moyenne de l'arbre : la complexité en moyenne du tri

Arbre de décision : propriété

- Un arbre binaire de hauteur h a au plus 2^h feuilles (cf transp. 190)
- Le nombre de feuilles de l'arbre de décision est exactement $n!$ où n est la taille du tableau à trier
  (par l'absurde : si moins que $n!$ certains tableaux ne seraient pas correctement triés)
- On a donc :

$$n! \leq 2^h \Rightarrow \log(n!) \leq h$$

- Formule de Stirling :

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n})) \Rightarrow n! \geq (\frac{n}{e})^n$$

$$h \geq \log(n!) > \log((\frac{n}{e})^n) = n\log n - n\log e \Rightarrow h = \Omega(n\log n)$$

Le problème du tri comparatif est $\Omega(n\log n)$

Ce qu'on a vu

- Catégorisation des algorithmes de tri
- QuickSort (tri en place en $\Theta(n \log n)$)
- Analyse du cas moyen d'un algorithme
- Notre première structure de données : le tas
- HeapSort (tri en place en $\Theta(n \log n)$)
- Borne inférieure sur les tris comparatifs

- Illustration :
  - `http://www.sorting-algorithms.com/`
  - `http://www.youtube.com/user/algorythmics`

Ce qu'on n'a pas vu

- Analyse du meilleur cas du tri par tas
- Invariant pour le tri par tas
- Méthodes de tri linéaire
- Méthodes de sélection : trouver l'élément de rang i