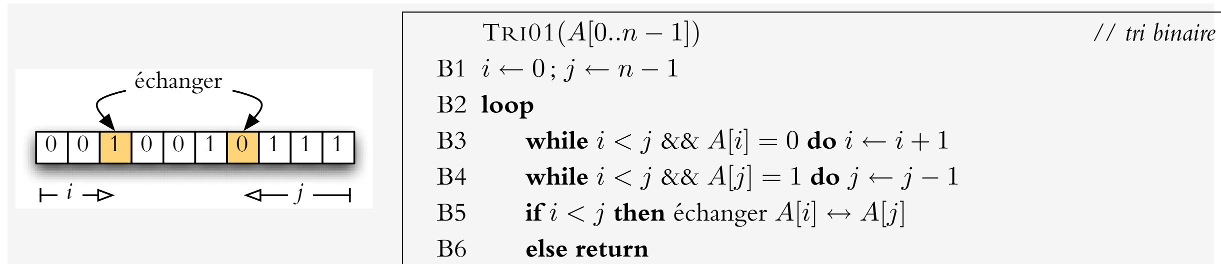
11 Tri rapide

11.1 Tri binaire

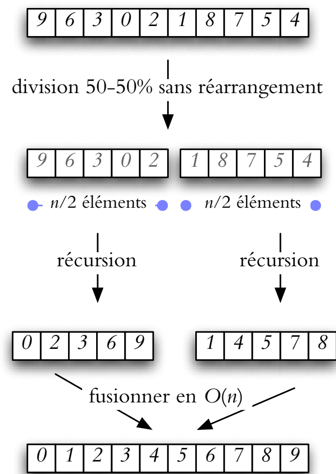
Supposons qu'il y a juste deux clés possibles (0 et 1) dans un tableau à trier. Alors on peut performer le tri en un temps linéaire à l'aide de deux indices qui balayent à partir des extremités vers le milieu.



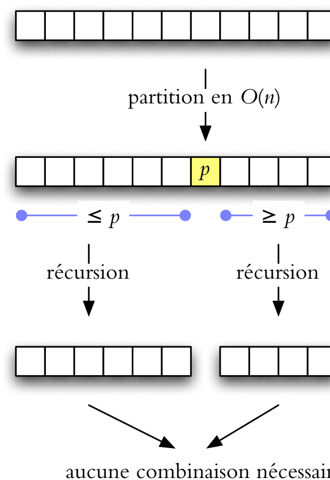
Exercice 11.1. On peut rendre l'exécution plus efficace si on a des sentinelles aux deux extrémités : $A[0] = 0$ et $A[n-1] = 1$. ▶ Montrez une version améliorée de TRI01 qui échange des éléments au besoin en un pré-traitement pour assurer une telle configuration de «serres-livres», et emploit des conditions plus simples dans les boucles intérieures. ♠

11.2 Tri rapide

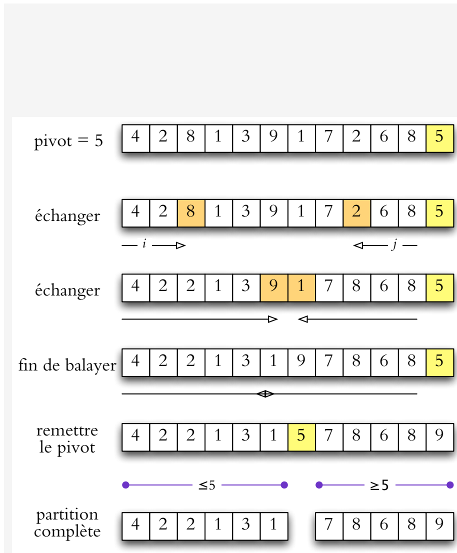
W(fr)



Le **tri par fusion** utilise la logique de «diviser pour régner» : le tableau est divisé en deux sous-tableaux (en temps $O(1)$) qui sont triés ensuite dans des appels récursifs, et on combine les résultats (fusion) en un temps linéaire.



En **tri rapide**, on choisit un **pivot** p, et à l'aide des échanges d'éléments, on place les éléments inférieurs à p à la gauche, et ceux supérieurs à p à la droite du tableau. Après une telle partition, on peut procéder avec des appels récursifs aux deux sous-tableaux gauche et droit. Notez que le pivot n'est pas nécessairement la médiane : les sous-tableaux gaches et droits résultants peuvent avoir des tailles très différentes. La partition même suit la logique du tri binaire.



L'idée principale est la **partition** autour d'un pivot.

Algo $\textsc{Quicksort}(A[0..n-1], g, d)$ // *tri de* $A[g..d-1]$
- Q1 **if** $d - g \leq 1$ **then return** // *cas de base*
- Q2 $i \leftarrow \textsc{Partition}(A, g, d)$
- Q3 $\textsc{Quicksort}(A, g, i)$
- Q4 $\textsc{Quicksort}(A, i+1, d)$

Algo $\textsc{Partition}(A, g, d)$ // *partition de* $A[g..d-1]$
- P1 chosir le pivot $p \leftarrow A[d-1]$
- P2 $i \leftarrow g-1$; $j \leftarrow d-1$
- P3 **loop** // *condition terminale à P6*
- P4 **do** $i \leftarrow i+1$ **while** $A[i] < p$
- P5 **do** $j \leftarrow j-1$ **while** $j > i$ et $A[j] > p$
- P6 **if** $i \geq j$ **then** sortir de la boucle (à P8)
- P7 échanger $A[i] \leftrightarrow A[j]$
- P8 échanger $A[i] \leftrightarrow A[d]$
- P9 **return** i

Pour trier un tableau $A[0..n-1]$ en ordre croissant, on exécute $\textsc{Quicksort}(A, 0, n-1)$. C'est un tri en place.

11.3 Performances

Soit $m = d - g$, le nombre des éléments dans le sous-tableau à trier, avec $m > 1$. La partition (Lignes P3–P7) se fait en un temps $\Theta(m)$. Le temps de calcul est donc

$$T(m) = \Theta(m) + T(i) + T(m-1-i).$$

La récurrence dépend de l'indice i du pivot.

	pivot i	**récurrence** $T(n)$	**solution** $T(n)$
Meilleur cas	$(n-1)/2$	$2 \cdot T\big((n-1)/2\big) + \Theta(n)$	$\Theta(n \log n)$
Pire cas	$0, n-1$	$T(n-1) + \Theta(n)$	$\Theta(n^2)$
Moyen cas	aléatoire	$\mathbb{E}T(n) = 2\mathbb{E}T(i) + \Theta(n)$	$\Theta(n \log n)$

Le pire cas arrive (entre autres) quand on a un tableau trié au début !

Exercice 11.2. On a aussi $O(n \log n)$ quand la partition regroupe au moins une fraction fixé (p.e., $\alpha = 10\%$) des éléments à la fois. ▶ Démontrez que le tri rapide prend $O(n \log n)$ si la position i du pivot satisfait $\min\{i, m-1-i\} \geq m/3$ lors de la partition de sous-tableaux de taille m.
En général, supposons qu'il existe un $\alpha \in (0, 1/2]$ tel que $\min\{i, m-1-i\} \geq \alpha m$ lors de la partition de sous-tableaux de taille m. On a donc $T(n) \leq O(n) + T(n\alpha) + T\big(n(1-\alpha)\big)$ pour presque tout n. ▶ Démontrez que $T(n) = O(n \log n / \log(1-\alpha)^{-1})$.
Remarque : en conséquence, on a la borne $T(n) = O(\alpha^{-1} n \log n)$ après développement en série de Taylor : $\ln(1-\alpha) = -\alpha(1 - o(1))$ quand $\alpha \to 0$. ♠

11.4 Génie algorithmique

Petits sous-tableaux. Le **tri par insertion** est plus rapide que quicksort quand $d - g$ est petit ($g \geq d - \ell^*$ avec $\ell^* = 5..20$). En Ligne Q1, c'est mieux donc de faire le tri par insertion pour tels petits tableaux. En fait, on peut juste **ignorer** les petits sous-tableaux entièrement (retourner si $g \geq d - \ell^*$ en Ligne Q1). À la fin, il faut parcourir le tableau entier selon tri par insertion en $\Theta(n\ell^*) = \Theta(n)$.

Choix du pivot. Deux choix performent très bien en pratique : médiane ou aléatoire.

Médiane de trois

```
P1.1  if d ≥ g + 2 then
P1.2      if A[g] > A[d − 2] then échanger A[g] ↔ A[d − 2]
P1.3      if A[d−1] > A[d−2] then échanger A[d−1] ↔ A[d−2]
P1.4      if A[g] > A[d − 1] then échanger A[g] ↔ A[d − 1]
P1.5  p ← A[d − 1]                    // A[g] ≤ A[d − 1] ≤ A[d − 2]
```

et on se sert des **sentinelles** qui sont maintenant en place à $A[g]$, $A[d-2]$:

```
P2'  i ← g; j ← d − 2
P5'      do j ← j − 1 while A[j] > p
```

Aléatoire

```
P1.1  k ← g + rnd(d − g)
P1.2  p ← A[k]
P1.3  if k ≠ d − 1 then
P1.4      A[k] ← A[d − 1]
P1.5      A[d − 1] ← p
```

Profondeur de la pile d'exécution. En une implantation efficace, on se sert de la position terminale du deuxième appel récursif (Ligne Q4).

```
     Algo QUICKSORT_ITER(A[0..n − 1], g, d)        // tri de A[g..d − 1]
QI1  while d − g > 1 do
QI2      i ← PARTITION(A, g, d)
QI3      QUICKSORT_ITER(A, g, i)
QI4      g ← i + 1                     // boucler au lieu de l'appel récursif
```

La profondeur de la pile d'exécution dépend donc du nombre d'appels récursifs en Ligne QI3 ce qui est $\Theta(n)$ au pire (p.e., on a $i = d$ toujours). On peut facilement modifier le code pour toujours faire l'appel récursif avec le plus court entre $A[g..i-1]$ et $A[i+1..d-1]$ qui assure que la profondeur maximale est $\Theta(\log n)$.

11.5 Sélection

Supposons qu'on veut trouver le k-ème plus petit élément dans un tableau $A[0..n-1]$. Il existe des algorithmes qui le font en temps $\Theta(n)$ au pire cas. Ici, on se sert plutôt de la partition autour d'un pivot pour achever un temps de calcul linéaire *en moyen cas* (voir Théorème 11.1 ci-bas), mais $\Theta(n^2)$ au pire. En pratique, l'algorithme par partition est souvent plus performant que l'algorithme avec un temps linéaire théoriquement guaranti.

Idée de clé : après avoir appellé $i \leftarrow \text{PARTITION}(A, 0, n)$, on trouve le k-ème élément en $A[0..i-1]$ si $k < i$ ou en $A[i+1..n-1]$ si $k > i$. En même temps, on réorganise le tableau pour que $A[k]$ soit le k-ème plus petit élément.

```
Algo SELECTION(A[0..n − 1], g, d, k)
S1 if d − g ≤ 2 then                                          // cas de base : 1 ou 2 éléments
S2     if d = g + 2 et A[d − 1] < A[g] then échanger A[g] ↔ A[d − 1]          // 2 éléments
S3     return A[k]
S4 i ← PARTITION(A, g, d)
S5 if k = i then return A[k]                                          // on l'a trouvé
S6 if k < i then return SELECTION(A, g, i, k)                     // continuer à la gauche
S7 if k > i then return SELECTION(A, i + 1, d, k)                  // continuer à la droite
```

Comme c'est une **récursion terminale**, on peut transformer le code en forme itérative très facilement.

Théorème 11.1. *Avec un pivot aléatoire, algorithme* SELECTION *fait* $\big(2+o(1)\big)n$ *comparaisons en moyenne.*

11.6 Moyen cas

Théorème 11.2. *Soit* $D(n)$ *le nombre moyen de comparaisons lors du tri de* n *éléments avec pivotage aléatoire. Alors,*

$$\frac{D(n)}{n} \sim \big(2+o(1)\big)H_n \sim 2\ln n \approx 1.39\lg n.$$

Lemme 11.3. *On a* $D(0) = D(1) = 0$*, et*

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

Démonstration. Supposons que le pivot est le i-ème plus grand élément de A. Le pivot est comparé à $(n-1)$ autre éléments pour la partition. Les deux partitions sont de tailles i et $(n-1-i)$. Or, i prend les valeurs $0, 1, \ldots, n-1$ avec la même probabilité. ∎

Preuve de Théorème 11.2. Par Lemme 11.3,

$$\begin{aligned} nD(n)-(n-1)D(n-1) &= \Big(n(n-1)+2\sum_{i=0}^{n-1}D(i)\Big) - \Big((n-1)(n-2)+2\sum_{i=0}^{n-2}D(i)\Big) \\ &= 2(n-1)+2D(n-1). \end{aligned}$$

D'où on a

$$\frac{D(n)}{n+1} = \frac{D(n-1)}{n}+\frac{2n-2}{n(n+1)} = \frac{D(n-1)}{n}+\frac{4}{n+1}-\frac{2}{n}.$$

Avec $E(n) = \frac{D(n)-2}{n+1}$, on a $E(n) = E(n-1)+\frac{2}{n+1}$, donc $E(n) = E(0)+\frac{2}{2}+\frac{2}{3}+\cdots+\frac{2}{n+1} = 2H_{n+1}-4$, où $H_n = \sum_{i=1}^{n} 1/i = \ln n+\gamma+o(1)$ est le n-ème nombre harmonique ($\gamma = 0.5772\cdots$ est la constante d'Euler-Mascheroni).
En retournant à $D(n) = 2+(n+1)E(n)$, on a alors $D(n) = 2(n+1)H_{n+1}-4n-2 < 2nH_{n+1}$.
Donc le nombre de comparaisons en moyenne est tel que $\frac{D(n)}{n} < 2H_{n+1} = O(\log n)$. ∎

Exercice 11.3. ▶ Démontrez Théorème 11.1. ♠

11.7 Permutation aléatoire en place

Au lieu de choisir un pivot au hasard, on peut randomiser l'ordre par un réarrangement au début, et procéder par pivotage déterministe. L'algorithme suivant met les éléments d'un tableau dans un ordre aléatoire *en place*, selon la distribution uniforme sur toutes ($n!$) permutations.

```
    SHUFFLE(A[0..n−1])                    // met les éléments de A dans un ordre aléatoire
P1  for i ← 0, 1, ..., n−1 do
P2      j ← i + rnd(n−i)         // rnd(m) donne un nombre entier (pseudo-)aléatoire de {0, 1, ..., m−1}
P3      échanger A[i] ↔ A[j]     // échange de A[i] et un élément du suffixe A[i..n−1] ; i = j est possible
```

11.8 Le minimum de comparaisons dans un tri

Un **tri par comparaison** n'utilise que des comparaisons entre les éléments d'un tableau (genre $A[i] < A[j]$) pour le trier. L'exécution de l'algorithme ne dépend que l'ordre initial du tableau. On définit l'**arbre de décision** qui montre la séquence de comparaisons pour toute exécution possible. Un nœud interne de l'arbre contient une comparaison entre les éléments de A ; il a toujours deux enfants qui correspondent au branchement de l'exécution selon le résultat de la comparaison. Tout nœud externe correspond à une permutation ce qui est l'ordre final des éléments. La Figure 1 montre l'exemple du tri par insertion de $A[0..3]$.

Théorème 11.4. *Pour tout algorithme déterministe de tri par comparaison, il existe un arrangement initiel de n éléments distincts qui prend au moins $\lg(n!)$ comparaisons à trier.*

Démonstration. Par définition, chaque chemin de la racine jusqu'à un nœud externe correspond à la séquence de comparaisons performées pour établir l'ordre final. Le tableau doit être trié à la sortie, et chaque ordre est possible : il faut avoir au moins un nœud externe pour toute permutation. Le nombre de nœuds externes dans cet arbre binaire est donc $\geq n!$. En conséquence, la hauteur de l'arbre est au moins $\lg(n!)$ (voir Théorème 8.3). Dans d'autres mots, il existe un ordre à l'entrée pour lequel l'algorithme fait au moins $\lceil \lg(n!) \rceil$ comparaisons. ■

Selon Théorème 11.2, le tri rapide est seulement 39% plus lente en moyenne que le tri le-plus-rapide-imaginable qui ferait $\lg n!$ comparaisons au pire par Théorème 11.4.

REMARQUE. Le théorème montre qu'aucun algorithme déterministe ne peut trier n éléments avec moins que $\lg(n!) \sim n \lg n$ comparaisons ; donc le temps de calcul doit être $\Omega(n \log n)$ au pire. Le tri par fusion utilise ce nombre minimal de comparaisons. Par contre, on peut exploiter parfois la distribution des éléments dans le tableau et dépasser la borne : c'est le cas avec le tri binaire qui prend $\Theta(n)$ temps car il utilise le fait qu'il y a juste deux clés différentes parmi les éléments.

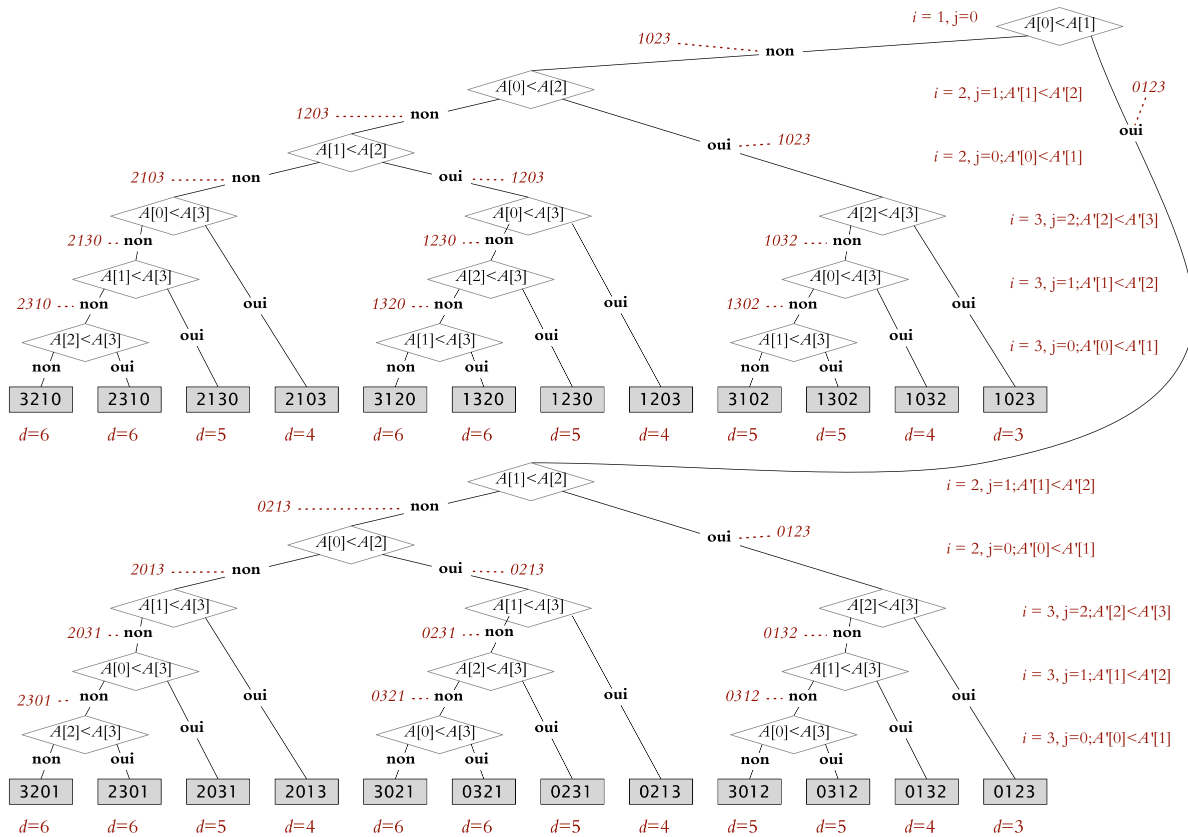


FIG. 1 – Arbre de décision du tri par insertion (§10.3, version initiale qui échange les éléments à chaque comparaison) appliqué à 4 éléments. Les nœuds internes montrent les indices originaux des éléments. A' est le tableau réarrangé pendant l'exécution, le résultat des échanges sur l'ordre initial est indiqué à chaque branche (p.e., à l'échec de la première comparaison $A[0] < A[1]$, on échange les deux qui donne l'ordre 1023 sur la branche "non"). C'est un arbre de hauteur 6 — l'algorithme utilise 6 comparaisons au pire. La profondeur des nœuds externes ($d =$) est entre 3 (meilleur cas) et 6 (pire cas).