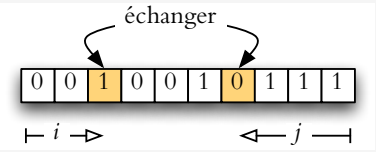
12 Tri rapide

12.1 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.




```
    TRI01(A[0..n − 1])                                          // tri binaire
B1  i ← 0 ; j ← n − 1
B2  boucler
B3      tandis que A[i] = 0 et i < j faire i ← i + 1
B4      tandis que A[j] = 1 et i < j faire j ← j − 1
B5      si i < j alors échanger A[i] ↔ A[j]
B6      sinon sortir de la boucle
```

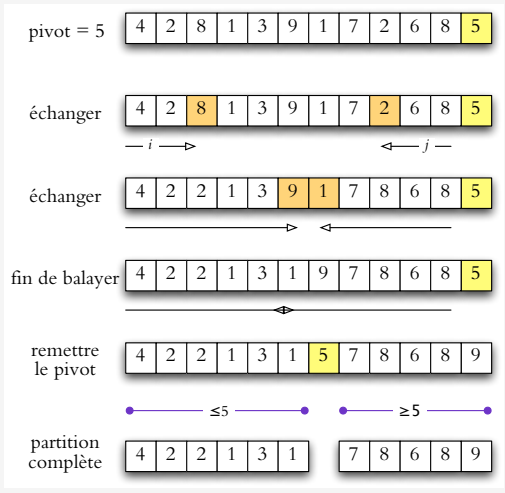
On peut généraliser l'idée pour trier des clés de b bits. D'abord on doit trier selon le bit le plus significatif (BPS), en déterminant deux blocs : éléments dont la clé a un BPS 0 et ceux avec BPS 1. On continue avec le tri selon le deuxième bit le plus significatif à l'intérieur des blocs, etc. Le tri prend $O(nb)$ temps.

```
    TRIMSD(A[], g, d, k)
    // tri de A[g..d] selon bits k, k + 1, . . . ; k = b − 1 est le BPS, k = 0 est le moins significatif
B1  si g ≥ d ou k < 0 alors retourner
B2  i ← g; j ← d
B3  boucler                                          // tri binaire selon le k-ème bit
B4      tandis que bit(A[i], k) = 0 et i < j faire i ← i + 1
B5      tandis que bit(A[j], k) = 1 et i < j faire j ← j − 1
B6      si i < j alors échanger A[i] ↔ A[j] sinon sortir de la boucle
B7  si bit(A[i], k) = 0 alors j ← j + 1                  // au cas où A[g..d] ≡ 0
B8  TRIMSD(A, g, j − 1, k − 1)
B9  TRIMSD(A, j, d, k − 1)
```

12.2 Tri rapide

W(fr)

La récursion du **tri rapide** (*quicksort*) représente une logique complémentaire à celle du tri par fusion. De même façon, le tableau à l'entrée est divisé en deux et les deux moitiés sont triés par des appels récursifs. En tri par fusion, la division ne dépend pas du contenu du tableau — les deux moitiés sont combinés après les appels récursifs. En tri rapide, la division se fait en groupant les éléments dans les deux moitiés à l'aide d'un élément appellé le **pivot** p. On place les éléments inférieurs à p à la gauche, et ceux supérieurs à p à la droite du tableau. Avec p au milieu, on devra pas combiner les deux sous-tableaux après les avoir triés. Donc on fait la majeure partie du travail avant les appels récursifs, en une structure algorithmique de genre «combiner pour régner». La partition même suit la logique du tri binaire.



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

Algo QUICKSORT$(A[0..n-1], g, d)$ *// tri de $A[g..d]$*
Q1 **si** $g \geq d$ **alors retourner** *// cas de base*
Q2 $i \leftarrow$ PARTITION(A, g, d)
Q3 QUICKSORT$(A, g, i-1)$
Q4 QUICKSORT$(A, i+1, d)$

Algo PARTITION(A, g, d) *// partition de $A[g..d]$*
P1 chosir le pivot $p \leftarrow A[d]$
P2 $i \leftarrow g-1$; $j \leftarrow d$
P3 **boucler**
P4 **faire** $i \leftarrow i+1$ **tandis que** $A[i] < p$
P5 **faire** $j \leftarrow j-1$ **tandis que** $j \geq i$ et $A[j] > p$
P6 **si** $i \geq j$ **alors** sortir de la boucle
P7 échanger $A[i] \leftrightarrow A[j]$
P8 échanger $A[i] \leftrightarrow A[d]$
P9 **retourner** i

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

12.3 Performances

La partition (Lignes P3–P7) se fait en un temps $\Theta(n)$. Le temps de calcul est donc

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

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 12.1. *On a aussi $O(n \log n)$ quand la partition regroupe au moins, α fraction (p.e., $\alpha = 10\%$) des éléments à la fois. En général, supposons qu'il existe un $\alpha \in (0, 1/2]$ tel que $\min\{i, n-1-i\} \geq \alpha n$ lors de la partition de sous-tableaux de taille n. On a donc $T(n) \leq O(n) + T(n\alpha) + T\big(n(1-\alpha)\big)$ (pour tout n à partir de $n = n_0$ quelconque). Démontrez que $T(n) = O(n \log n / \log(1-\alpha)^{-1})$. Remarque : comme $\ln(1-\alpha) = -\alpha(1-o(1))$ (par expansion de la série Taylor), on a $T(n) = O(n\alpha^{-1} \log n)$.*

12.4 Améliorations

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

et on se sert des **sentinelles** qui sont maintenant en place $A[g]$, $A[d-1]$:

```
P2'  i ← g; j ← d − 1
P5'      faire j ← j − 1 tandis que A[j] > p
```

Aléatoire

```
P1.1  k ← RANDOM(g, d)
P1.2  p ← A[k]
P1.3  si k ≠ d alors
P1.4      A[k] ← A[d]
P1.5      A[d] ← 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]
QI1  tandis que g < d faire
QI2      i ← PARTITION(A, g, d)
QI3      QUICKSORT_ITER(A, g, i − 1)
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 $A[i+1..d]$ qui assure que la profondeur maximale est $\Theta(\log n)$.

12.5 Moyen cas

Théorème 12.1. *Soit $D(n)$ le nombre moyen de comparaisons avec un pivot aléatoire, où n est le nombre d'éléments dans un tableau $A[0..n-1]$. Alors,*

$$\frac{D(n)}{n} = O(\log n).$$

Lemme 12.2. *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 12.1. Par Lemme 12.2,

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

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 peut écrire

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

Donc,

$$\begin{aligned} E(n) &= E(0) + \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n+1} \\ &= \frac{D(0)-2}{1} + 2(H_{n+1} - 1) = 2H_{n+1} - 4 \end{aligned}$$

où $H_n = \sum_{i=1}^{n} 1/i = \ln n + \gamma + o(1)$ est le n-ième 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 $\frac{D(n)}{n} < 2H_{n+1} = O(\log n)$. ■

En fait la preuve montre que $D(n)/n = \big(2 + o(1)\big)H_n \sim 2\ln n \approx 1.39 \lg n$. C'est seulement 39% pire que le meilleur cas !

12.6 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. On peut se servir de la partition autour d'un pivot et achever un temps de calcul linéaire *en moyen cas* (voir Théorème 12.3 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 $i \leftarrow$ PARTITION$(A, 0, n-1)$, 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 si d ≤ g + 1 alors                                   // cas de base : 1 ou 2 éléments
S2     si d = g + 1 et A[d] < A[g] alors échanger A[g] ↔ A[d]   // 2 éléments
S3     retourner A[k]
S4 i ← PARTITION(A, g, d)
S5 si k = i alors retourner A[k]                        // on l'a trouvé
S6 si k < i alors retourner SELECTION(A, g, i − 1, k)   // continuer à la gauche
S7 si k > i alors retourner 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 12.3. *Avec un pivot aléatoire, algorithme* SELECTION *fait* $\big(2 + o(1)\big)n$ *comparaisons en moyenne.*

Exercice 12.2. *Démontrer Théorème 12.3.*