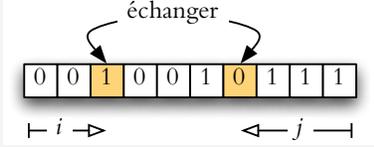


11 Tri rapide

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 extrémités vers le milieu.



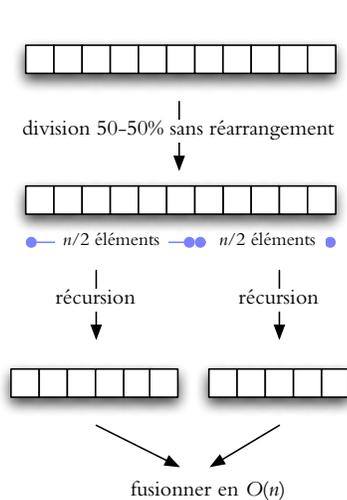
```

TRI01(A[0..n-1]) // tri binaire
B1 i ← 0; j ← n - 1
B2 loop
B3   while A[i] = 0 et i < j do i ← i + 1
B4   while A[j] = 1 et i < j do j ← j - 1
B5   if i < j then échanger A[i] ↔ A[j]
B6   else return

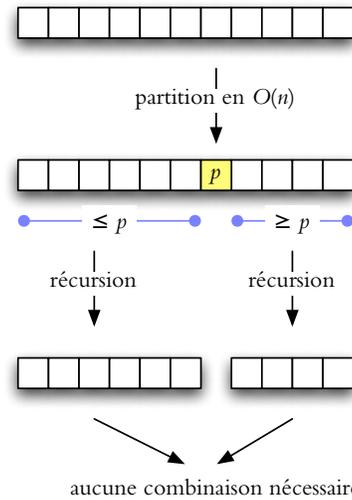
```

Exercice 11.1. On peut rendre l'exécution plus efficace si on a des sentinelles aux deux extrémités : $A[0] = 0$ et/ou $A[n-1] = 1$. Montrez une version améliorée de TRI01 qui place des sentinelles dans un premier parcours, et utilise des conditions plus simples dans les boucles intérieures.

11.2 Tri rapide



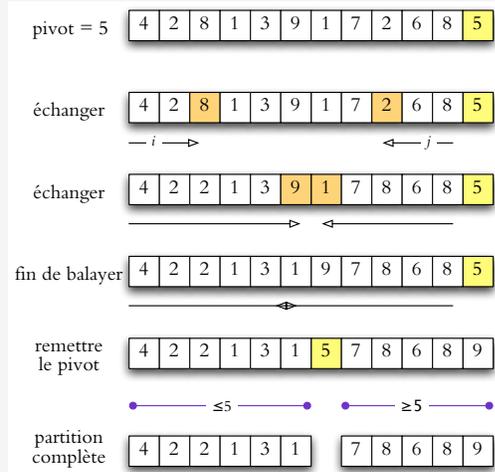
Le tri par fusion utilise la logique de «diviser pour régner» : le tableau est divisé en deux sous-tableaux (en temps $O(1)$) qui sont triés ensuite dans des appels récursifs, et on combine les résultats (fusion) en un temps linéaire.



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 gauches et droits résultants peuvent avoir des tailles très différentes. La partition même suit la logique du tri binaire.

W^(fr)

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



```

Algo QUICKSORT( $A[0..n-1], g, d$ ) // tri de  $A[g..d]$ 
Q1 if  $g \geq d$  then return // 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 choisir le pivot  $p \leftarrow A[d]$ 
P2  $i \leftarrow g-1; j \leftarrow d$ 
P3 loop
P4 do  $i \leftarrow i+1$  while  $A[i] < p$ 
P5 do  $j \leftarrow j-1$  while  $j \geq i$  et  $A[j] > p$ 
P6 if  $i \geq j$  then sortir de la boucle
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 $\text{QUICKSORT}(A, 0, n-1)$. C'est un tri en place.

11.3 Performances

Soit $m = d - g + 1$, 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	$(n-1)/2$	$2 \cdot T((n-1)/2) + \Theta(n)$	$\Theta(n \log n)$
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 fixée (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 .

11.4 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 performant très bien en pratique : médiane ou aléatoire.

Médiane de trois

```

P1.1 si  $d \geq g + 2$  alors
P1.2   if  $A[g] > A[d - 1]$  then échanger  $A[g] \leftrightarrow A[d - 1]$ 
P1.3   if  $A[d] > A[d - 1]$  then échanger  $A[d] \leftrightarrow A[d - 1]$ 
P1.4   if  $A[g] > A[d]$  then échanger  $A[g] \leftrightarrow A[d]$ 
P1.5  $p \leftarrow A[d]$  //  $A[g] \leq A[d] \leq A[d - 1]$ 

```

Aléatoire

```

P1.1  $k \leftarrow \text{RANDOM}(g, d)$ 
P1.2  $p \leftarrow A[k]$ 
P1.3 if  $k \neq d$  then
P1.4    $A[k] \leftarrow A[d]$ 
P1.5    $A[d] \leftarrow p$ 

```

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

```

P2'  $i \leftarrow g; j \leftarrow d - 1$ 
P5'   do  $j \leftarrow j - 1$  while  $A[j] > p$ 

```

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 while  $g < d$  do
QI2    $i \leftarrow \text{PARTITION}(A, g, d)$ 
QI3   QUICKSORT_ITER( $A, g, i - 1$ )
QI4    $g \leftarrow i + 1$  // 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 $\Theta(\log n)$.

11.5 Moyen cas

Théorème 11.1. 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 11.2. On a $D(0) = D(1) = 0$, et

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

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 $0, 1, \dots, n - 1$ avec la même probabilité. ■

Preuve de Théorème 11.1. Par Lemme 11.2,

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

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,

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

où $H_n = \sum_{i=1}^n 1/i = \ln n + \gamma + o(1)$ est le n -ème nombre harmonique ($\gamma = 0.5772\dots$ 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 $D(n)/n = (2 + o(1))H_n \sim 2 \ln n \approx 1.39 \lg n$. C'est seulement 39% pire que le meilleur cas !

11.6 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. Ici, on se sert plutôt de la partition autour d'un pivot pour achever un temps de calcul linéaire *en moyen cas* (voir Théorème 11.3 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 garanti.

Idée de clé : après avoir appelé $i \leftarrow \text{PARTITION}(A, 0, n-1)$, 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 if  $d \leq g+1$  then                                     // cas de base : 1 ou 2 éléments
S2   if  $d = g+1$  et  $A[d] < A[g]$  then échanger  $A[g] \leftrightarrow A[d]$            // 2 éléments
S3   return  $A[k]$ 
S4  $i \leftarrow \text{PARTITION}(A, g, d)$ 
S5 if  $k = i$  then return  $A[k]$                                      // on l'a trouvé
S6 if  $k < i$  then return SELECTION( $A, g, i-1, k$ )                 // continuer à la gauche
S7 if  $k > i$  then return SELECTION( $A, i+1, d, k$ )                 // continuer à la droite

```

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

Théorème 11.3. Avec un pivot aléatoire, algorithme SELECTION fait $(2 + o(1))n$ comparaisons en moyenne.

Exercice 11.3. Démontrer Théorème 11.3.