11. Algorithmes de tri interne

LE TRI est une tâche fondamentale en informatique. On a un fichier d'éléments avec **clés comparables** — on veut les ranger selon l'ordre les clés. Clés comparables en Java :

```
public interface Comparable<T>
{
      int compareTo(T autre_objet);
}
```

`x.compareTo(y)` retourne une valeur négative si `x` précède `y`, ou positive si `x` suit `y`, selon l'ordre «naturel» des éléments.

Tri externe et interne. Un algorithme de **tri externe** sert à ordonner les éléments d'un fichier stocké partiellement ou entièrement en mémoire externe (disque). Il existe des algorithmes spécialisés pour trier des fichiers trop grand pour la mémoire vive : en une telle application, on veut minimiser l'accès à mémoire externe. Le **tri interne** se fait sur un fichier stocké entièrement en mémoire vive. Le plus souvent, on considère le tri d'un tableau $A[0..n-1]$; parfois, on peut adapter un tel algorithme aux listes chaînées aussi.

Temps de calcul. On caractérise le temps de calcul d'un algorithme de tri par le nombre d'opérations de **comparaison** entre éléments et d'**échange** d'éléments. En une caractérisation plus générale d'un tri de tableau, on compte le nombre d'**accès** aux cellules.

Espace de travail. Un aspect important d'efficacité est l'usage de mémoire pendant le tri. Noter que cela inclut toutes les variables locales sur la pile d'appel dans récurrences. Un algorithme *en place* utilise $O(\log n)$ mémoire à part du fichier d'entrée. (En particulier, un tel algorithme ne crée pas de copies du tableau à trier.)

Méthode de tri interne	**liste chaînée**	**comparaisons**	**espace de travail**
tri par sélection (*selection sort*)	oui	$\sim n^2/2$ (toujours)	$O(1)$ — en place
tri par insertion (*insertion sort*)	oui	$\sim n^2/2$ (pire), $\Theta(n^2)$ (moyenne), $\sim n$ (meilleur)	$O(1)$ — en place
tri par fusion (*Mergesort*)	oui	$\sim n \lg n$ (toujours)	$\Theta(n)$
tri par tas (*Heapsort*)	non	$\sim 2n \lg n$ (pire), $\Theta(n \log n)$ (toujours)	$O(1)$ — en place
tri rapide (*Quicksort*)	non	$\sim 2n \ln n$ (moyenne), $O(n^2)$ (pire)	$O(\log n)$ — en place

11.1 Tri par fusion

Fusion de deux tableaux triés

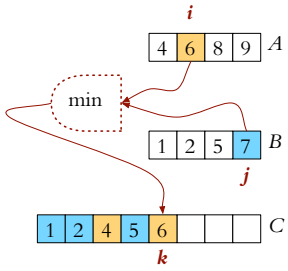
On peut fusionner deux listes (implantées comme tableaux ou listes chaînées), par la récurrence

$$\mathsf{fusion}(A,B) = \begin{cases} B & \text{si } A \text{ est vide} \\ A & \text{si } B \text{ est vide} \\ A[0] \odot \mathsf{fusion}(A[1..],B) & \text{si } A[0] \le B[0] \\ B[0] \odot \mathsf{fusion}(A,B[1..]) & \text{si } B[0] < A[0] \end{cases}$$

où $\odot$ dénote la concaténation.

Dans le cas de deux tableaux, une solution itérative utilise deux indices parcourant les deux listes.

```
    FUSION(A[0..n−1], B[0..m−1]) (de type Comparable[], triés)
F1  initialiser C[0..n+m−1]                          // on place le résultat dans C
F2  i ← 0; j ← 0; k ← 0                  // i est l'indice dans A ; j est l'indice dans B
F3  while i < n et j < m do
F4      if A[i] ≤ B[j] then C[k] ← A[i]; i ← i+1
F5      else C[k] ← B[j]; j ← j+1
F6      k ← k+1
F7  while i < n do C[k] ← A[i]; i ← i+1; k ← k+1
F8  while j < m do C[k] ← B[j]; j ← j+1; k ← k+1
```



Fusion de tableaux triés.

Théorème 11.1. *L'algorithme* FUSION *calcule la fusion de deux tableaux triés en un temps* $\Theta(n+m)$*, avec* $n+m-1$ *comparaisons d'éléments au pire.*

Tri par fusion

Tri par fusion[1] (*mergesort*) utilise le principe de **diviser pour régner** dans une procédure récursive.

[1] W(fr):tri par fusion

$$\mathsf{tri}(A[0..n-1]) = \begin{cases} A & \{n<2\} \\ \mathsf{fusion}\Big(\mathsf{tri}(A[0..\lfloor n/2\rfloor]), \mathsf{tri}(A[\lfloor n/2\rfloor+1..n-1])\Big) & \{n\ge 2\} \end{cases}$$

```
    MERGESORT(A[0..n−1], g, d)                          // appel initiel avec g = 0, d = n
    // récursion pour trier le sous-tableau A[g..d−1]
M1  if d − g < 2 then return                  // cas de base : tableau vide ou un seul élément
M2  m ← ⌊(d+g)/2⌋                                                    // m est au milieu
M3  MERGESORT(A, g, m)                                               // trier partie gauche
M4  MERGESORT(A, m, d)                                               // trier partie droite
M5  FUSION(A, g, m, d)                                               // fusion des résultats
```

Tri hybride. En pratique, le tri par fusion performe le mieux dans une **approche hybride** : la récursion est trop couteuse pour les petits sous-tableaux,

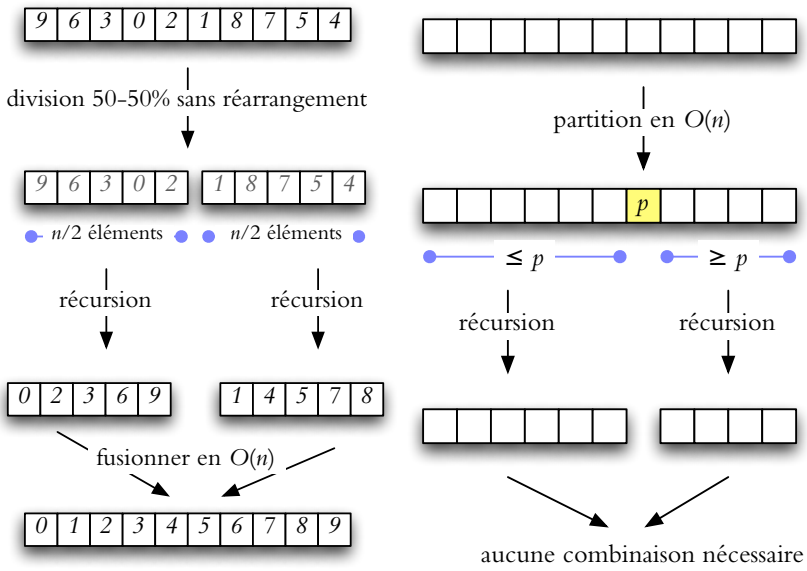


FIG. 1: Démarches du tri par fusion et du tri rapide. Le **tri par fusion** utilise la logique de «diviser pour régner» : le tableau est divisé en deux sous-tableaux (en temps $O(1)$) qui sont triés ensuite dans des appels récursifs, et on combine les résultats (fusion) en un temps linéaire. En **tri 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 gaches et droits résultants peuvent avoir des tailles très différentes.

et le tri par insertion est plus rapide. Pour cette raison, il vaut implanter le tri par fusion avec un seuil $\ell > 1$ sur les petits sous-tableaux. On passe les sous-tableaux de taille $d - g < \ell$ en Ligne M1 directement à un tri par insertion.

Gestion d'espace auxiliaire. Typiquement, on utilise un seul tableau auxiliaire pour faire la fusion. Avec un arrangement **bitonique**, on peut simplifier les conditions dans la boucle de la fusion.

```
    FUSION(A[],g,m,d)                          // fusion pour A[g..m−1] et A[m..d−1]
F1  for i ← g, g+1, ..., m−1 do aux[i] ← A[i]
F2  for j ← m, m+1, ..., d−1 do aux[m+d−1−j] ← A[j]
F3  i ← g ; j ← d−1 ; k ← g
F4  while k < d do                             // meilleure condition : i < m
F5      if aux[i] ≤ aux[j] then A[k] ← aux[i] ; i ← i+1; k ← k+1
F6      else A[k] ← aux[j] ; j ← j−1; k ← k+1
```

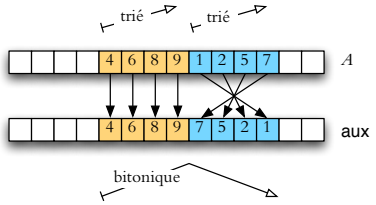


FIG. 2: Tri par fusion avec arrangement bitonique

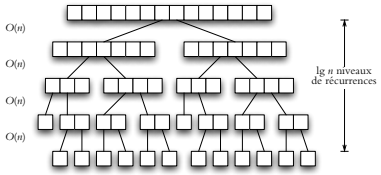


FIG. 3: Pourquoi le temps de calcul est $\Theta(n \log n)$.

Temps de calcul du tri par fusion

Dans la fusion, on combine les deux tableaux triés en un troisième en un temps linéaire (Théorème 11.1). Le temps de calcul s'écrit[2] donc comme

$$T(n) = T\big(\lfloor n/2 \rfloor\big) + T\big(\lceil n/2 \rceil\big) + T_{\text{fusion}}\big(\lfloor n/2 \rfloor, \lceil n/2 \rceil\big) + O(1) \quad (11.1)$$

où $T_{\text{fusion}}(k,m) = \Theta(k+m)$ est le temps pour fusionner deux tableaux de tailles k et m. On a donc $T_{\text{fusion}}(k,m) = \Theta(k+m)$ en (11.1), qui mène à la récurrence classique

$$T(n) = T\big(\lfloor n/2 \rfloor\big) + T\big(\lceil n/2 \rceil\big) + \Theta(n).$$

La solution de la récurrence est $T(n) = \Theta(n \log n)$. On peut voir cette solution directement en dessinant l'**arbre de récursions** : on a $\lceil \lg n \rceil$ niveaux et $O(n)$ travail (fusions) à chaque niveau.

[2] On a vu antérieurement la récurrence

$$C(n) = (n-1) + C\big(\lfloor n/2 \rfloor\big) + C\big(\lceil n/2 \rceil\big)$$

pour le maximum de comparaisons dans mergesort. Avec $C(0) = C(1) = 0$, la solution exacte est

$$C(n) = n \lg n - a_n \cdot n + 1$$

avec $0.9139 \leq a_n \leq 1$, une facteur périodique. Précisement

$$a_n = A\big(\lg n - \lfloor \lg n \rfloor\big) \qquad \text{où}$$
$$A(u) = u + 2^{1-u} - 1 \qquad \{0 \leq u < 1\}$$

11.2 Tri rapide

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 deux côtés vers le milieu.

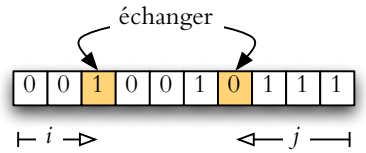


FIG. 4: Tri binaire.

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

Tri rapide

W(fr):tri rapide

L'idée principale est la **partition** autour d'un pivot. Les éléments plus petits ou plus grands sont triés antre eux par récursion. La partition même suit la logique du tri binaire. Afin de trier un tableau $A[0..n-1]$ en ordre croissant, on exécute QUICKSORT$(A, 0, n-1)$. C'est un tri en place.

```
    Algo QUICKSORT(A[0..n−1], g, d)                    // tri de A[g..d − 1]
Q1  if d − g ≤ 1 then return                           // cas de base
Q2  i ← PARTITION(A, g, d)
Q3  QUICKSORT(A, g, i)
Q4  QUICKSORT(A, i + 1, d)

    Algo PARTITION(A, g, d)                            // partition de A[g..d − 1]
P1  chosir le pivot p ← A[d − 1]
P2  i ← g − 1 ; j ← d − 1
P3  loop                                               // condition terminale à P6
P4      do i ← i + 1 while A[i] < p
P5      do j ← j − 1 while j > i et A[j] > p
P6      if i ≥ j then sortir de la boucle (à P8)
P7      échanger A[i] ↔ A[j]
P8  échanger A[i] ↔ A[d]
P9  return i
```

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 !

Moyen cas

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

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

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

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

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 11.2. 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 $\frac{D(n)}{n}<2H_{n+1}=O(\log n)$. ■

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 $\Theta(\log n)$.

QuickSelect

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 11.4 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 11.4. *Avec un pivot aléatoire, algorithme* SELECTION *fait* $\big(2 + o(1)\big)n$ *comparaisons en moyenne.*

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 élémént du suffixe A[i..n − 1] ; i = j est possible
```

11.3 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 5 montre l'exemple du tri par insertion de $A[0..3]$.

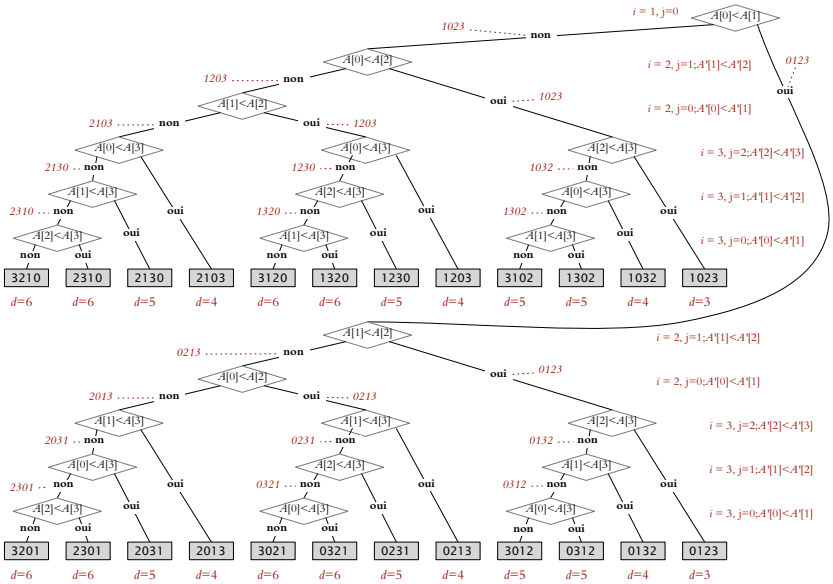


FIG. 5: Arbre de décision du tri par insertion (§10.3, version initiale qui échange les éléments à chaque comparaison) appliqué à 4 éléments. Les nœuds internes montrent les indices originaux des éléments. A' est le tableau réarrangé pendant l'exécution, le résultat des échanges sur l'ordre initial est 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).

Théorème 11.5. *Pour tout algorithme déterministe de tri par comparaison, il existe un arrangement initiel de n éléments distincts qui prend au moins $\lg(n!) \sim n \lg n$ 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. ■

Théorème 11.5 montre que le pire temps de calcul doit être $\Omega(n \log n)$ pour tout algorithme de tri par comparaison — c'est une barrière informatique insurmontable. Le tri par fusion utilise ce nombre minimal de comparaisons ; le tri rapide en fait 39% plus en moyenne selon Théorème 11.2. 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.