Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
`dominique.poulalhon@irif.fr`

Université Paris Diderot

L2 Informatique & Math-Info

Année universitaire 2019-2020

La semaine dernière...

apport de l'hypothèse « L *est un tableau trié* » sur quelques problèmes manipulant des listes

La semaine dernière...

apport de l'hypothèse « L *est un tableau trié* » sur quelques problèmes manipulant des listes

deux exemples d'algorithmes de tri par comparaisons :

La semaine dernière...

apport de l'hypothèse « L *est un tableau trié* » sur quelques problèmes manipulant des listes

deux exemples d'algorithmes de tri par comparaisons :

- le tri par sélection

La semaine dernière...

apport de l'hypothèse « L *est un tableau trié* » sur quelques problèmes manipulant des listes

deux exemples d'algorithmes de tri par comparaisons :

- le tri par sélection
- le tri par insertion

La semaine dernière...

apport de l'hypothèse « L *est un tableau trié* » sur quelques problèmes manipulant des listes

deux exemples d'algorithmes de tri par comparaisons :

- le tri par sélection
- le tri par insertion

tri par comparaisons : algorithme n'utilisant pas d'autre propriété sur les éléments que l'existence d'un ordre total
$\Longrightarrow$ les éléments ne peuvent être utilisés que pour des comparaisons deux à deux

Tri par sélection $\Theta(n^2)$ comparaisons dans tous les cas

Tri par insertion $\Theta(n^2)$ comparaisons au pire

Questions

- peut-on être plus précis pour le tri par insertion ?
- peut-on faire mieux que $\Theta(n^2)$ dans le pire cas ?

permutation de taille n = bijection de $[\![1, n]\!]$ dans lui-même

Permutations

permutation de taille n = bijection de $[\![1, n]\!]$ dans lui-même

$\mathfrak{S}_n$ = ensemble des permutations de taille n

permutation de taille n = bijection de $[\![1, n]\!]$ dans lui-même

$\mathfrak{S}_n$ = ensemble des permutations de taille n

notation bilinéaire : $\sigma = \begin{pmatrix} 1 & 2 & \dots & n \\ \sigma(1) & \sigma(2) & \dots & \sigma(n) \end{pmatrix}$

Permutations

permutation de taille n $=$ bijection de $[\![1,n]\!]$ dans lui-même

$\mathfrak{S}_n$ $=$ ensemble des permutations de taille n

notation bilinéaire : $\sigma = \begin{pmatrix} 1 & 2 & \dots & n \\ \sigma(1) & \sigma(2) & \dots & \sigma(n) \end{pmatrix}$

notation linéaire : $\sigma = \sigma(1)\ \sigma(2)\ \dots\ \sigma(n)$

Exemple

$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array}$$

Exemple

$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \quad \Leftrightarrow \quad \sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 2 & 1 & 6 & 7 & 3 & 5 \end{pmatrix}$$

Exemple

$$\begin{array}{ccc} 1 & \xmapsto{\sigma} & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array}$$

$$\Leftrightarrow \quad \sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 2 & 1 & 6 & 7 & 3 & 5 \end{pmatrix}$$

$$\Leftrightarrow \quad \sigma = \boxed{4 \quad 2 \quad 1 \quad 6 \quad 7 \quad 3 \quad 5}$$

Exemple

$$\sigma$$

$$\begin{array}{lcl} 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \qquad \Leftrightarrow \qquad \sigma = \quad 1 \longmapsto 4$$

$$
\begin{array}{ccc}
 & \sigma & \\
1 & \longmapsto & 4 \\
2 & \longmapsto & 2 \\
3 & \longmapsto & 1 \\
4 & \longmapsto & 6 \\
5 & \longmapsto & 7 \\
6 & \longmapsto & 3 \\
7 & \longmapsto & 5
\end{array}
\quad \Longleftrightarrow \quad
\sigma =
\begin{tikzcd}
1 \arrow[r, mapsto] & 4 \arrow[d, mapsto] \\
 & 6
\end{tikzcd}
$$

Exemple

$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \qquad \iff \qquad \sigma = \begin{tikzcd} 1 \arrow[r, mapsto] & 4 \arrow[d, mapsto] \\ 3 & 6 \arrow[l, mapsto] \end{tikzcd}$$

Exemple

$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \quad \Leftrightarrow \quad \sigma = \begin{tikzcd} 1 \arrow[r, mapsto] & 4 \arrow[d, mapsto] \\ 3 \arrow[u, mapsto] & 6 \arrow[l, mapsto] \end{tikzcd}$$

Exemple

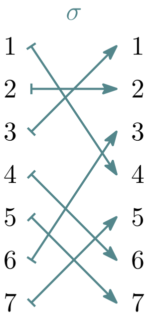
$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \quad \iff \quad \sigma = \begin{tikzcd} 1 \arrow[r, mapsto] & 4 \arrow[d, mapsto] \\ 3 \arrow[u, mapsto] & 6 \arrow[l, mapsto] \end{tikzcd} \qquad \begin{tikzcd} 2 \arrow[loop above] \end{tikzcd}$$

Exemple

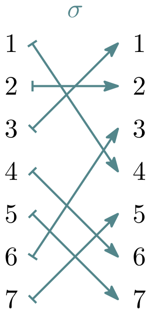
$$
\begin{array}{ccc}
 & \sigma & \\
1 & \longmapsto & 4 \\
2 & \longmapsto & 2 \\
3 & \longmapsto & 1 \\
4 & \longmapsto & 6 \\
5 & \longmapsto & 7 \\
6 & \longmapsto & 3 \\
7 & \longmapsto & 5
\end{array}
\quad \Leftrightarrow \quad
\sigma =
\begin{tikzcd}
1 \arrow[r, mapsto] & 4 \arrow[d, mapsto] \\
3 \arrow[u, mapsto] & 6 \arrow[l, mapsto]
\end{tikzcd}
\qquad
\begin{tikzcd}
2 \arrow[loop above, mapsto]
\end{tikzcd}
\qquad
\begin{tikzcd}
5 \arrow[d, mapsto, bend right] \\
7 \arrow[u, mapsto, bend right]
\end{tikzcd}
$$

Exemple

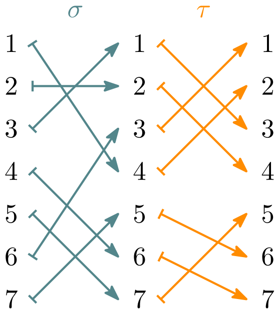
$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \qquad \Longleftrightarrow$$



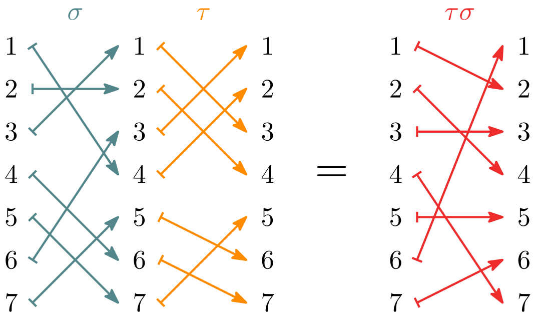
Exemple



Exemple



Exemple



Produit, inverse

produit : $\sigma\tau = \sigma \circ \tau \; : \; i \xmapsto{\tau} \tau(i) \xmapsto{\sigma} \sigma(\tau(i))$

produit : $\sigma\tau = \sigma \circ \tau \ : \ i \xmapsto{\tau} \tau(i) \xmapsto{\sigma} \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

produit : $\sigma\tau = \sigma \circ \tau \ : \ i \xmapsto{\tau} \tau(i) \xmapsto{\sigma} \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

inverse de σ : application τ telle que $\tau\sigma = \mathrm{id}_n = 1\ 2\ \ldots\ n$

notation : σ^{-1}

$$i \xmapsto{\sigma} \sigma(i) \xmapsto{\tau = \sigma^{-1}} i$$

produit : $\sigma\tau = \sigma \circ \tau \ : \ i \xmapsto{\ \tau\ } \tau(i) \xmapsto{\ \sigma\ } \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

inverse de σ : application τ telle que $\tau\sigma = \mathrm{id}_n = 1\ 2\ \ldots\ n$

notation : σ^{-1}

$$i \xmapsto{\ \sigma\ } \sigma(i) \xmapsto{\ \tau = \sigma^{-1}\ } i$$

Lemme

produit : $\sigma\tau = \sigma \circ \tau \;:\; i \xmapsto{\tau} \tau(i) \xmapsto{\sigma} \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

inverse de σ : application τ telle que $\tau\sigma = \mathrm{id}_n = 1\ 2\ \ldots\ n$

notation : σ^{-1}

$$i \xmapsto{\sigma} \sigma(i) \xmapsto{\tau = \sigma^{-1}} i$$

Lemme

- $\sigma \in \mathfrak{S}_n \implies \sigma^{-1} \in \mathfrak{S}_n$

produit : $\sigma\tau = \sigma \circ \tau \ : \ i \xmapsto{\ \tau\ } \tau(i) \xmapsto{\ \sigma\ } \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

inverse de σ : application τ telle que $\tau\sigma = \mathrm{id}_n = 1\ 2\ \ldots\ n$

notation : σ^{-1}

$$i \xmapsto{\ \sigma\ } \sigma(i) \xmapsto{\ \tau = \sigma^{-1}\ } i$$

Lemme

- $\sigma \in \mathfrak{S}_n \implies \sigma^{-1} \in \mathfrak{S}_n$
- $\sigma\sigma^{-1} = \sigma^{-1}\sigma = \mathrm{id}_n \ : \ i = \sigma(j) \xmapsto{\ \sigma^{-1}\ } \sigma^{-1}(i) = j \xmapsto{\ \sigma\ } i$

produit : $\sigma\tau = \sigma \circ \tau \ : \ i \xmapsto{\ \tau\ } \tau(i) \xmapsto{\ \sigma\ } \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

inverse de σ : application τ telle que $\tau\sigma = id_n = 1\ 2\ \dots\ n$

notation : σ^{-1}

$$i \xmapsto{\ \sigma\ } \sigma(i) \xmapsto{\ \tau = \sigma^{-1}\ } i$$

Lemme

- $\sigma \in \mathfrak{S}_n \implies \sigma^{-1} \in \mathfrak{S}_n$
- $\sigma\sigma^{-1} = \sigma^{-1}\sigma = id_n \ : \ i = \sigma(j) \xmapsto{\ \sigma^{-1}\ } \sigma^{-1}(i) = j \xmapsto{\ \sigma\ } i$
- $(\sigma^{-1})^{-1} = \sigma$

produit : $\sigma\tau = \sigma \circ \tau \ : \ i \xmapsto{\tau} \tau(i) \xmapsto{\sigma} \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

inverse de σ : application τ telle que $\tau\sigma = \mathrm{id}_n = 1\ 2\ \dots\ n$

notation : σ^{-1}

$$i \xmapsto{\sigma} \sigma(i) \xmapsto{\tau = \sigma^{-1}} i$$

Lemme

- $\sigma \in \mathfrak{S}_n \implies \sigma^{-1} \in \mathfrak{S}_n$
- $\sigma\sigma^{-1} = \sigma^{-1}\sigma = \mathrm{id}_n \ : \ i = \sigma(j) \xmapsto{\sigma^{-1}} \sigma^{-1}(i) = j \xmapsto{\sigma} i$
- $(\sigma^{-1})^{-1} = \sigma$

produit : $\sigma\tau = \sigma \circ \tau \ : \ i \xmapsto{\tau} \tau(i) \xmapsto{\sigma} \sigma(\tau(i))$

Lemme

$\sigma, \tau \in \mathfrak{S}_n \implies \sigma\tau \in \mathfrak{S}_n$ *(loi de composition interne)*

inverse de σ : application τ telle que $\tau\sigma = \mathrm{id}_n = 1\ 2\ \dots\ n$

notation : σ^{-1}

$$i \xmapsto{\sigma} \sigma(i) \xmapsto{\tau = \sigma^{-1}} i$$

Lemme

- $\sigma \in \mathfrak{S}_n \implies \sigma^{-1} \in \mathfrak{S}_n$
- $\sigma\sigma^{-1} = \sigma^{-1}\sigma = \mathrm{id}_n \ : \ i = \sigma(j) \xmapsto{\sigma^{-1}} \sigma^{-1}(i) = j \xmapsto{\sigma} i$
- $(\sigma^{-1})^{-1} = \sigma$

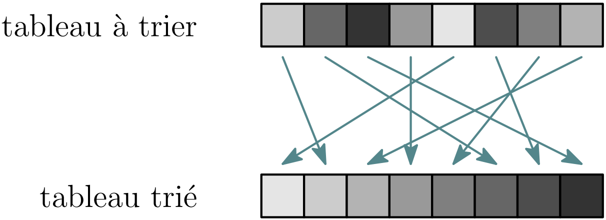
(on dit que $\mathfrak{S}_n$ a une structure de groupe)

Tris *vs.* permutations

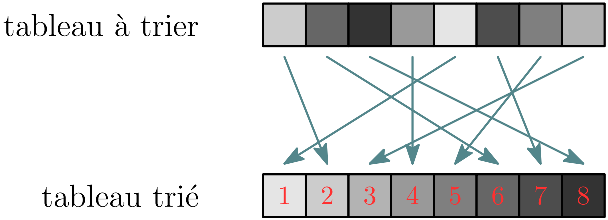
tableau à trier

tableau trié

Tris *vs.* permutations



Tris *vs.* permutations



Tris *vs.* permutations

tableau à trier

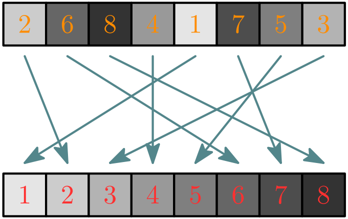


tableau trié

Tris *vs.* permutations

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 6 & 8 & 4 & 1 & 7 & 5 & 3 \end{pmatrix}$$

$$id = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}$$

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 6 & 8 & 4 & 1 & 7 & 5 & 3 \end{pmatrix}$$

produit
par σ^{-1}

$$id = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}$$

Tris *vs.* permutations

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 6 & 8 & 4 & 1 & 7 & 5 & 3 \end{pmatrix}$$

produit par σ^{-1}

$$id = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}$$

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

point fixe = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) = i$
point mobile = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) \neq i$
support = ensemble des points mobiles de σ (noté $\mathrm{Supp}(\sigma)$)

point fixe = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) = i$
point mobile = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) \neq i$
support = ensemble des points mobiles de σ (noté $\mathrm{Supp}(\sigma)$)

transposition = permutation ayant exactement 2 points mobiles
(et donc exactement $n-2$ points fixes)

si $\mathrm{Supp}(\tau) = \{i, j\}$, on note $\tau = (i\ j)$

point fixe = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) = i$
point mobile = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) \neq i$
support = ensemble des points mobiles de σ (noté $\mathrm{Supp}(\sigma)$)

transposition = permutation ayant exactement 2 points mobiles
(et donc exactement $n - 2$ points fixes)

si $\mathrm{Supp}(\tau) = \{i, j\}$, on note $\tau = (i\ j)$

action par produit à gauche : si $\sigma \in \mathfrak{S}_n$, alors

$$(i\ j)\ \sigma = (i\ j) \circ \sigma \ : \ k \longmapsto \begin{cases} i & \text{si } k = \sigma^{-1}(j) \\ j & \text{si } k = \sigma^{-1}(i) \\ \sigma(k) & \text{sinon} \end{cases}$$

Transpositions

point fixe = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) = i$
point mobile = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) \neq i$
support = ensemble des points mobiles de σ (noté $\mathrm{Supp}(\sigma)$)

transposition = permutation ayant exactement 2 points mobiles
(et donc exactement $n-2$ points fixes)

si $\mathrm{Supp}(\tau) = \{i, j\}$, on note $\tau = (i\ j)$

action par produit à gauche : si $\sigma \in \mathfrak{S}_n$, alors

$$(i\ j)\ \sigma = (i\ j) \circ \sigma\ :\ k \longmapsto \begin{cases} i & \text{si } k = \sigma^{-1}(j) \\ j & \text{si } k = \sigma^{-1}(i) \\ \sigma(k) & \text{sinon} \end{cases}$$

= échange des valeurs i et j

point fixe = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) = i$
point mobile = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) \neq i$
support = ensemble des points mobiles de σ (noté $\mathrm{Supp}(\sigma)$)

transposition = permutation ayant exactement 2 points mobiles
(et donc exactement $n - 2$ points fixes)

si $\mathrm{Supp}(\tau) = \{i, j\}$, on note $\tau = (i\ j)$

action par produit à droite : si $\sigma \in \mathfrak{S}_n$, alors

$$\sigma\ (i\ j) = \sigma \circ (i\ j)\ :\ k \longmapsto \begin{cases} \sigma(j) & \text{si } k = i \\ \sigma(i) & \text{si } k = j \\ \sigma(k) & \text{sinon} \end{cases}$$

Transpositions

point fixe = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) = i$
point mobile = élément $i \in [\![1, n]\!]$ t.q. $\sigma(i) \neq i$
support = ensemble des points mobiles de σ (noté $\mathrm{Supp}(\sigma)$)

transposition = permutation ayant exactement 2 points mobiles
(et donc exactement $n - 2$ points fixes)

si $\mathrm{Supp}(\tau) = \{i, j\}$, on note $\tau = (i\ j)$

action par produit à droite : si $\sigma \in \mathfrak{S}_n$, alors

$$\sigma\ (i\ j) = \sigma \circ (i\ j)\ :\ k \longmapsto \begin{cases} \sigma(j) & \text{si } k = i \\ \sigma(i) & \text{si } k = j \\ \sigma(k) & \text{sinon} \end{cases}$$

= échange des éléments en positions i et j

Lemme

toute permutation σ *possède une unique décomposition en produit de transpositions* $(a_1\ b_1)(a_2\ b_2)\dots(a_\ell\ b_\ell)$ *avec la contrainte :*

$$\forall i \leqslant \ell,\quad a_i < b_i \quad \textit{et} \quad a_1 < a_2 < \cdots < a_\ell$$

Lemme

toute permutation σ *possède une unique décomposition en produit de transpositions* $(a_1\ b_1)(a_2\ b_2)\ldots(a_\ell\ b_\ell)$ *avec la contrainte :*

$$\forall i \leqslant \ell,\quad a_i < b_i \quad \textit{et} \quad a_1 < a_2 < \cdots < a_\ell$$

De manière équivalente, $\sigma = \tau_1 \ldots \tau_n$ avec pour chaque i :

$$\tau_i = \mathrm{id} \quad \text{ou} \quad \tau_i = (i\ b_i) \text{ avec } b_i > i$$

$\Longrightarrow$ le nombre de tels produits est donc exactement $n!$

Lemme

toute permutation σ *possède une unique décomposition en produit de transpositions* $(a_1\ b_1)(a_2\ b_2)\dots(a_\ell\ b_\ell)$ *avec la contrainte :*

$$\forall i \leqslant \ell,\quad a_i < b_i \quad \textit{et} \quad a_1 < a_2 < \cdots < a_\ell$$

De manière équivalente, $\sigma = \tau_1 \dots \tau_n$ avec pour chaque i :

$$\tau_i = \mathrm{id} \quad \text{ou} \quad \tau_i = (i\ b_i) \text{ avec } b_i > i$$

$\Longrightarrow$ le nombre de tels produits est donc exactement $n!$

Ou encore : tout tableau peut être trié en échangeant l'élément en position 1 avec l'élément en position b_1, puis l'élément en position 2 avec l'élément en position b_2, ...

Lemme

toute permutation σ *possède une unique décomposition en produit de transpositions* $(a_1\ b_1)(a_2\ b_2)\dots(a_\ell\ b_\ell)$ *avec la contrainte :*

$$\forall i \leqslant \ell,\quad a_i < b_i \quad \textit{et} \quad a_1 < a_2 < \cdots < a_\ell$$

De manière équivalente, $\sigma = \tau_1 \dots \tau_n$ avec pour chaque i :

$$\tau_i = \mathrm{id} \quad \text{ou} \quad \tau_i = (i\ b_i) \text{ avec } b_i > i$$

$\Longrightarrow$ le nombre de tels produits est donc exactement $n!$

Ou encore : tout tableau peut être trié en échangeant l'élément en position 1 avec l'élément en position b_1, puis l'élément en position 2 avec l'élément en position b_2, ...

Démonstration.

C'est exactement ce que fait le tri par sélection (version en place)... $\square$

`RandomPermutation(n)`

construire une des $n!$ permutations de taille n selon la loi de probabilité uniforme

RandomPermutation(n)

construire une des $n!$ permutations de taille n selon la loi de probabilité uniforme

(*i.e.* : si on exécute tous les comportements (aléatoires) possibles, chaque permutation doit être obtenue le même nombre de fois)

Aparté : génération aléatoire de permutations

`RandomPermutation(n)`

construire une des $n!$ permutations de taille n selon la loi de probabilité uniforme

(*i.e.* : si on exécute tous les comportements (aléatoires) possibles, chaque permutation doit être obtenue le même nombre de fois)

Principe : mimer un tri par sélection, en remplaçant la recherche de l'indice du minimum par le tirage aléatoire d'un indice dans le bon intervalle

RandomPermutation(n)

construire une des $n!$ permutations de taille n selon la loi de probabilité uniforme

(*i.e.* : si on exécute tous les comportements (aléatoires) possibles, chaque permutation doit être obtenue le même nombre de fois)

```
from random import randint # générateur uniforme d'entiers
def randomPerm(n) :
  T = [ i+1 for i in range(n) ] # T = [ 1, 2, ..., n ]
  for i in range(n-1) :
    r = randint(i, n-1) # entier aléatoire dans [i, n-1]
    if i != r : T[i], T[r] = T[r], T[i]
  return T
```

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Lemme

le nombre de permutations de taille n est $n!$

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Lemme

le nombre de permutations de taille n *est* $n!$

Corollaire

un algorithme de tri doit avoir $n!$ *comportements différents sur les entrées de taille* n

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Lemme

le nombre de permutations de taille n *est* $n!$

Corollaire

un algorithme de tri doit avoir $n!$ *comportements différents sur les entrées de taille* n

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Question : c'est gros comment, $\log_2 n!$?

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Question : c'est gros comment, $\log_2 n!$?

Théorème

$$\log_2 n! \in \Theta(n \log n)$$

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Question : c'est gros comment, $\log_2 n!$?

Théorème

$$\log_2 n! \in \Theta(n \log n)$$

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en

$$\Omega(n \log n)$$

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Questions :

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Questions :

- existe-t-il des algorithmes de tri de complexité $\Theta(n \log n)$ en moyenne ? dans le pire cas ?

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Questions :

- existe-t-il des algorithmes de tri de complexité $\Theta(n \log n)$ en moyenne ? dans le pire cas ?
- quid de la complexité en moyenne du tri par insertion ?

Tri par fusion

tri utilisant la stratégie « diviser-pour-régner »

Tri par fusion

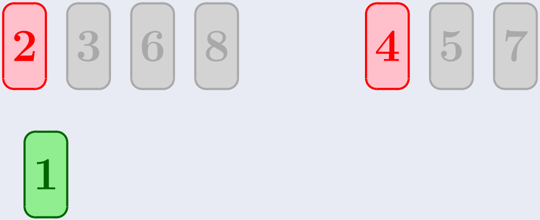
tri utilisant la stratégie « diviser-pour-régner »

Étape élémentaire : la fusion de listes triées

2 3 6 8 **1** 4 5 7

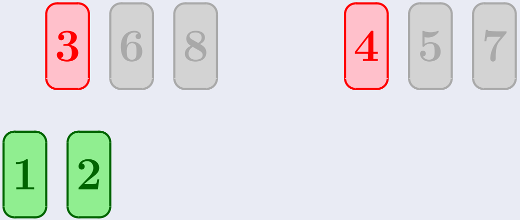
tri utilisant la stratégie « diviser-pour-régner »

Étape élémentaire : la fusion de listes triées



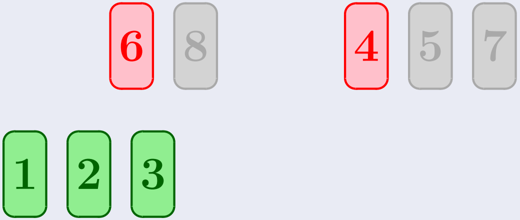
tri utilisant la stratégie « diviser-pour-régner »

Étape élémentaire : la fusion de listes triées



tri utilisant la stratégie « diviser-pour-régner »

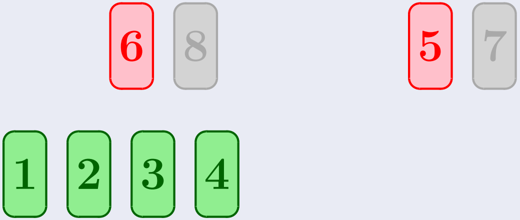
Étape élémentaire : la fusion de listes triées



Tri par fusion

tri utilisant la stratégie « diviser-pour-régner »

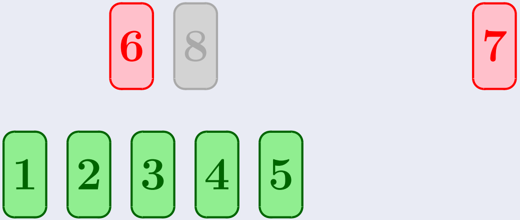
Étape élémentaire : la fusion de listes triées



Tri par fusion

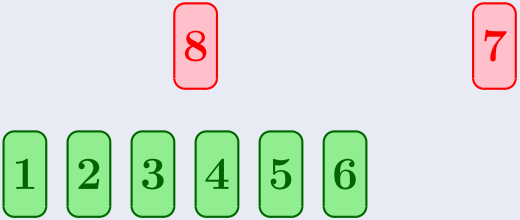
tri utilisant la stratégie « diviser-pour-régner »

Étape élémentaire : la fusion de listes triées



tri utilisant la stratégie « diviser-pour-régner »

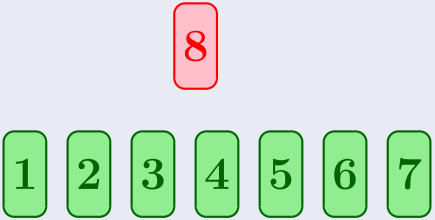
Étape élémentaire : la fusion de listes triées



Tri par fusion

tri utilisant la stratégie « diviser-pour-régner »

Étape élémentaire : la fusion de listes triées



tri utilisant la stratégie « diviser-pour-régner »

Étape élémentaire : la fusion de listes triées

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Tri par fusion

tri utilisant la stratégie « diviser-pour-régner »

Étape élémentaire : la fusion de listes triées

2 3 6 8 1 4 5 7

1 2 3 4 5 6 7 8

Étape élémentaire : la fusion de listes triées

```
def fusion(L1, L2) :        # version récursive (mal écrite)
  if len(L1) == 0 : return L2
  elif len(L2) == 0 : return L1
  elif L1[0] < L2[0] :
    return [L1[0]] + fusion(L1[1:], L2)
  else :
    return [L2[0]] + fusion(L1, L2[1:])
```

Étape élémentaire : la fusion de listes triées

```
def fusion(L1, L2) :      # version récursive (mal écrite)
  if len(L1) == 0 : return L2
  elif len(L2) == 0 : return L1
  elif L1[0] < L2[0] :
    return [L1[0]] + fusion(L1[1:], L2)
  else :
    return [L2[0]] + fusion(L1, L2[1:])
```

$\Longrightarrow$ complexité $\Theta(n)$, où n est la taille de la liste fusionnée

Étape élémentaire : la fusion de listes triées

```
def fusion(L1, L2) :      # version récursive (mal écrite)
  if len(L1) == 0 : return L2
  elif len(L2) == 0 : return L1
  elif L1[0] < L2[0] :
    return [L1[0]] + fusion(L1[1:], L2)
  else :
    return [L2[0]] + fusion(L1, L2[1:])
```

$\Longrightarrow$ complexité $\Theta(n)$, où n est la taille de la liste fusionnée

(enfin, pas telle que la fonction est écrite ci-dessus : chaque appel récursif travaille sur une *copie* de l'une des deux listes... mais c'est facile à résoudre en dérécursivant la fonction ou en passant les indices de début et fin en paramètre)

Tri par fusion

Exemple d'exécution complète :

3	**5**	**7**	**1**	**6**	**4**	**2**

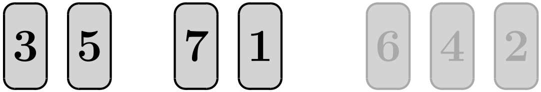
Tri par fusion

Exemple d'exécution complète :

3	**5**	**7**	**1**		**6**	**4**	**2**

Tri par fusion

Exemple d'exécution complète :



Exemple d'exécution complète :

3 **5**   7 1   6 4 2

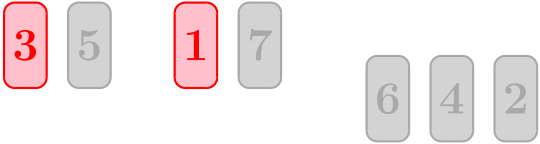
Exemple d'exécution complète :

Tri par fusion

Exemple d'exécution complète :

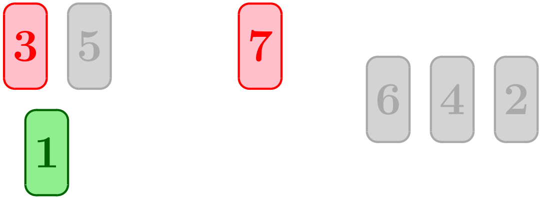
3 **5** **1** **7** **6** **4** **2**

Exemple d'exécution complète :



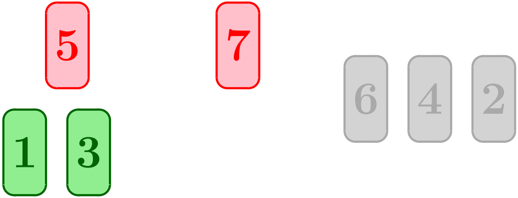
Tri par fusion

Exemple d'exécution complète :

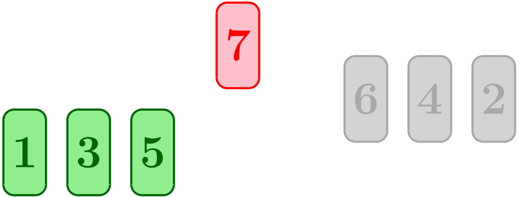


Tri par fusion

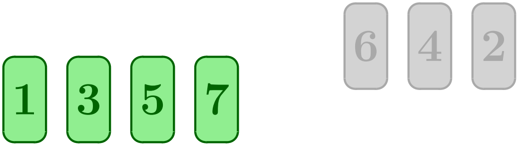
Exemple d'exécution complète :



Exemple d'exécution complète :



Exemple d'exécution complète :



Tri par fusion

Exemple d'exécution complète :

1	3	5	7		6	4	2

Tri par fusion

Exemple d'exécution complète :

Exemple d'exécution complète :

Tri par fusion

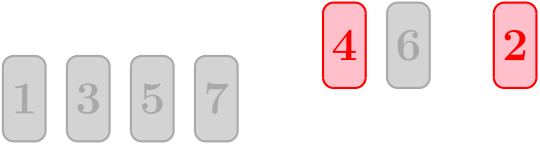
Exemple d'exécution complète :

Exemple d'exécution complète :

1 3 5 7   **4** **6**   **2**

Tri par fusion

Exemple d'exécution complète :



Exemple d’exécution complète :

Tri par fusion

Exemple d'exécution complète :

Tri par fusion

Exemple d'exécution complète :

Tri par fusion

Exemple d'exécution complète :

1 **3** **5** **7**

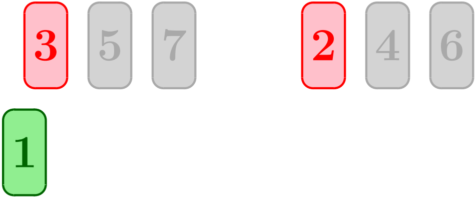
2 **4** **6**

Exemple d'exécution complète :

1 3 5 7   **2** 4 6

Tri par fusion

Exemple d'exécution complète :

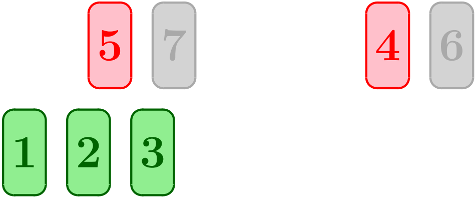


Tri par fusion

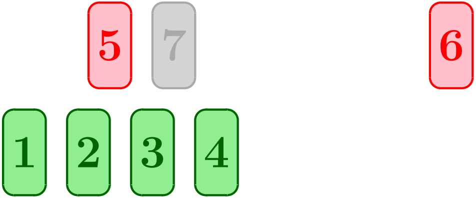
Exemple d'exécution complète :

Tri par fusion

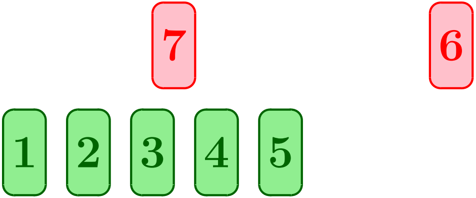
Exemple d'exécution complète :



Exemple d'exécution complète :

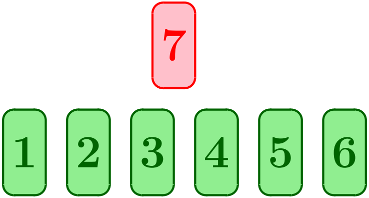


Exemple d'exécution complète :



Tri par fusion

Exemple d'exécution complète :



Tri par fusion

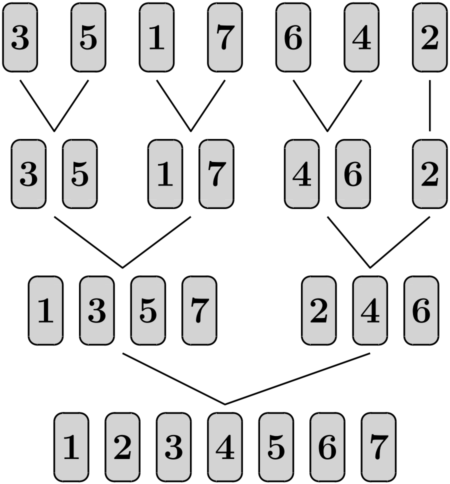
Exemple d'exécution complète :

$\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}$

Exemple d'exécution complète :

1	**2**	**3**	**4**	**5**	**6**	**7**

Récapitulatif des étapes de fusion :



Tri par fusion

```
def tri_fusion(T) :   # version trop naïve
  if len(T) < 2 : return T
  else :
    milieu = len(T)//2
    gauche = tri_fusion(T[:milieu])
    droite = tri_fusion(T[milieu:])
    return fusion(gauche, droite)
```

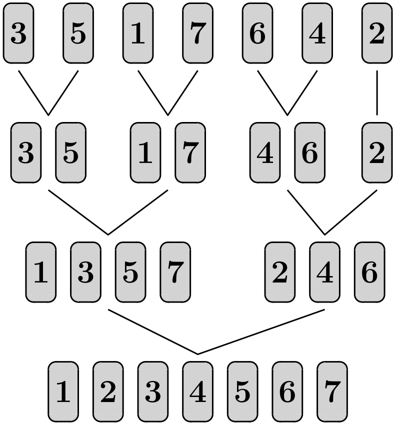
```
def tri_fusion(T) :   # version trop naïve
  if len(T) < 2 : return T
  else :
    milieu = len(T)//2
    gauche = tri_fusion(T[:milieu])
    droite = tri_fusion(T[milieu:])
    return fusion(gauche, droite)
```

(encore beaucoup de recopies de tableaux inutiles...)

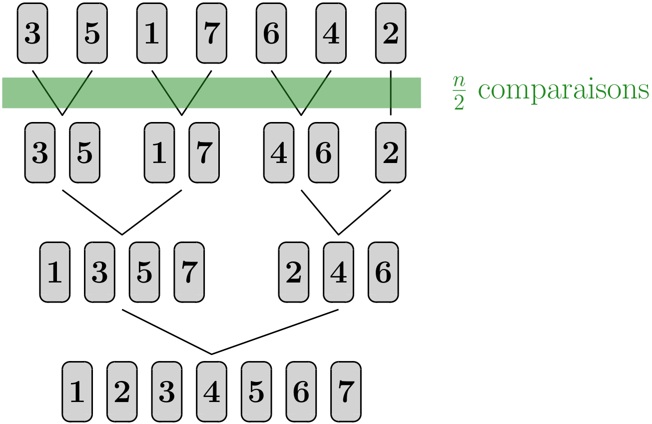
Tri par fusion

```
def tri_fusion(T, debut, fin) :
  ''' trie T entre les indices debut (inclus) et fin (exclue) '''
  if fin - debut < 2 : return T[debut:fin]
  else :
    milieu = (debut + fin)//2
    gauche = tri_fusion(T, debut, milieu)
    droite = tri_fusion(T, milieu, fin)
    return fusion(gauche, droite)
```

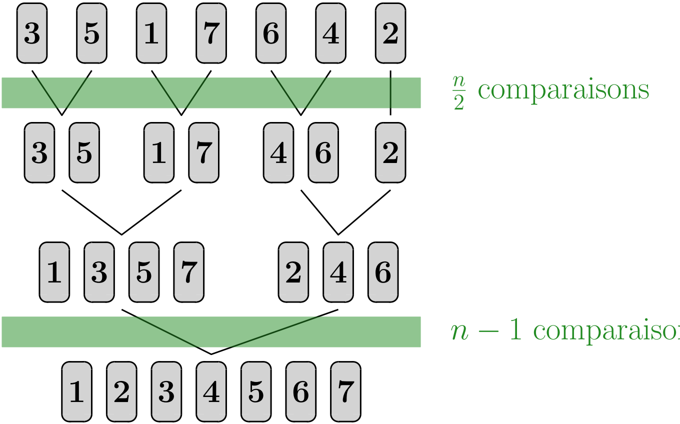
Complexité du tri par fusion



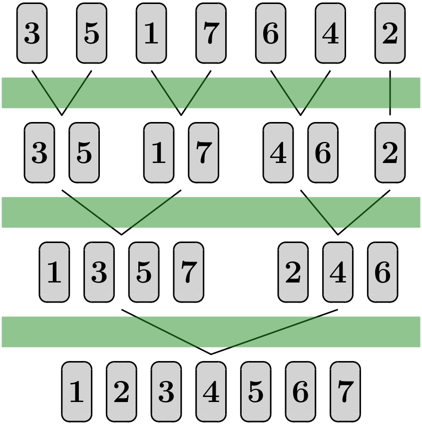
Complexité du tri par fusion



Complexité du tri par fusion



Complexité du tri par fusion

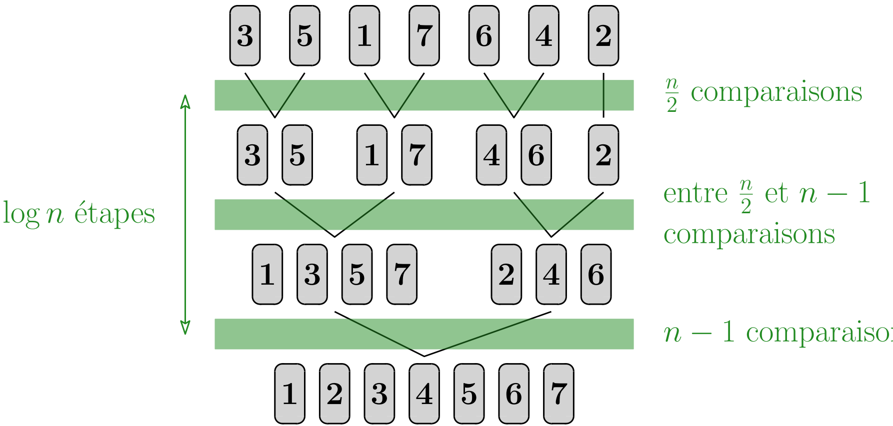


$\frac{n}{2}$ comparaisons

entre $\frac{n}{2}$ et $n-1$ comparaisons

$n-1$ comparaiso

Complexité du tri par fusion



Complexité du tri par fusion

Théorème

Le tri fusion d'un tableau de taille n *s'effectue en* $\Theta(n \log n)$ *comparaisons*

Complexité du tri par fusion

Théorème

Le tri fusion d'un tableau de taille n *s'effectue en* $\Theta(n \log n)$ *comparaisons*

Corollaire

Le tri fusion est un tri par comparaison asymptotiquement optimal

Complexité du tri par fusion

Théorème

Le tri fusion d'un tableau de taille n s'effectue en $\Theta(n \log n)$ comparaisons

Corollaire

Le tri fusion est un tri par comparaison asymptotiquement optimal

Points négatifs

- $\Theta(n \log n)$ comparaisons dans tous les cas (et jamais moins)
- la constante cachée dans le Θ est importante
- ne trie pas en place : complexité en espace $\in \Theta(n)$