Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon

dominique.poulalhon@irif.fr

Université de Paris

L2 Informatique & DL Bio-Info, Jap-Info, Math-Info

Année universitaire 2021-2022

Points d'organisation

Évaluation n° 1 : sur moodle, mardi 8 mars, 18h-19h
(plus précisément, limite d'une heure dans la plage 18h-19h30)

Contrôle n° 1 : sur table, vendredi 18 mars, 16h30-18h

programme : cours 1 à 6

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

Permutations

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

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

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

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 – représentations diverses

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

Exemple – représentations diverses

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

Exemple – représentations diverses

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

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

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

Exemple – représentations diverses

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

$$1 \longmapsto 4$$

Exemple – représentations diverses

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

Exemple – représentations diverses

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

$$
\begin{tikzcd}
1 \arrow[r, mapsto] & 4 \arrow[d, mapsto] \\
3 & 6 \arrow[l, mapsto]
\end{tikzcd}
$$

Exemple – représentations diverses

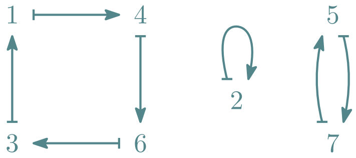
$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \quad \Leftrightarrow \quad \sigma = \begin{tikzcd} 1 \arrow[r, mapsto] & 4 \arrow[d, mapsto] \\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

Exemple – représentations diverses

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

Exemple – représentations diverses

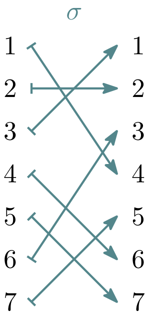
$$\begin{array}{ccc} & \sigma & \\ 1 & \longmapsto & 4 \\ 2 & \longmapsto & 2 \\ 3 & \longmapsto & 1 \\ 4 & \longmapsto & 6 \\ 5 & \longmapsto & 7 \\ 6 & \longmapsto & 3 \\ 7 & \longmapsto & 5 \end{array} \quad \Leftrightarrow \quad \sigma =$$



notation cyclique : sur cet exemple, $\sigma = (1\ 4\ 6\ 3)\ (2)\ (5\ 7)$,
ou plus simplement : $\sigma = (1\ 4\ 6\ 3)\ (5\ 7)$

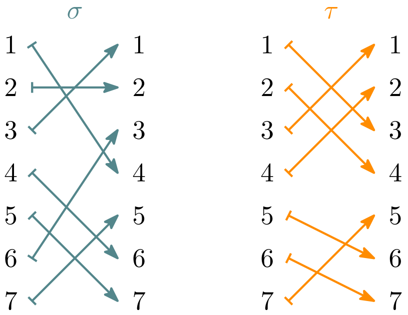
Exemple – représentations diverses

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



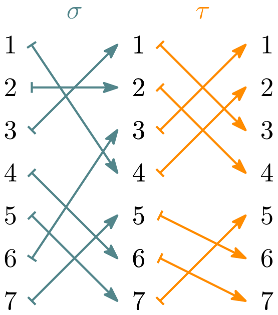
Produit de permutations

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



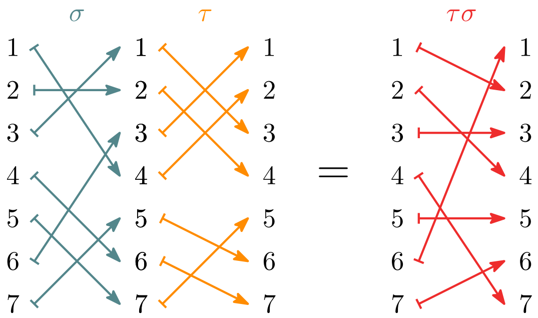
Produit de permutations

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



Produit de permutations

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

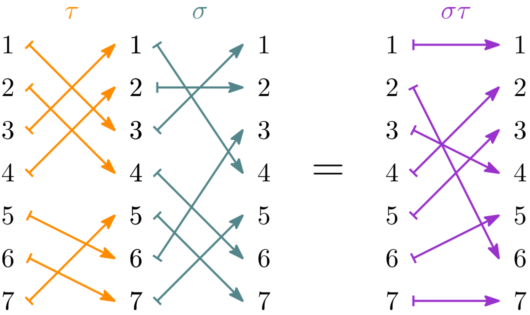


Lemme

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

Produit de permutations

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*)

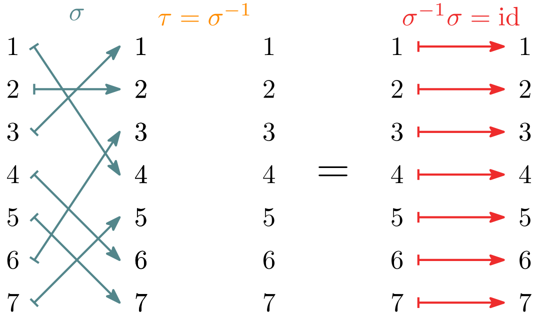
attention, le produit n'est pas commutatif !

Permutation inverse

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

notation : σ^{-1}

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

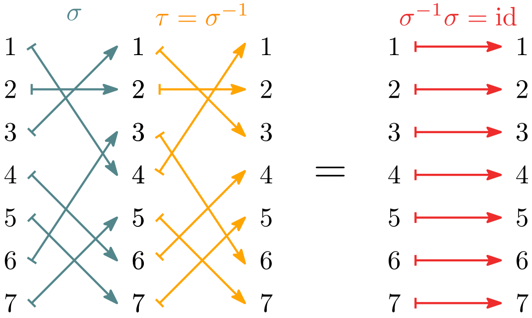


Permutation inverse

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

notation : σ^{-1}

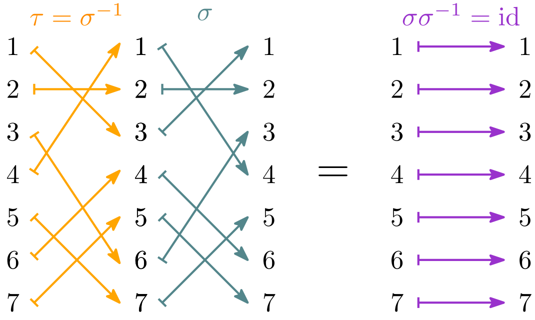
$$i \xmapsto{\ \sigma\ } \sigma(i) \xmapsto{\ \tau = \sigma^{-1}\ } i$$



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

notation : σ^{-1}

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



Permutation inverse

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

notation : σ^{-1}

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

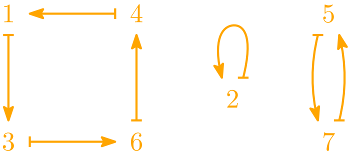
$$\begin{tikzcd}
 & 1 \arrow[r, mapsto] & 4 \arrow[dd, mapsto] & & 5 \arrow[dd, mapsto, bend left] \\
\sigma = & & & 2 \arrow[loop above, mapsto] & \\
 & 3 \arrow[uu, mapsto] & 6 \arrow[l, mapsto] & & 7 \arrow[uu, mapsto, bend left]
\end{tikzcd}$$

Permutation inverse

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

notation : σ^{-1}

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

$\sigma^{-1} =$ 

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

tableau à trier

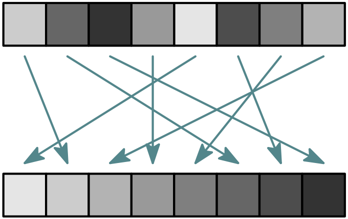
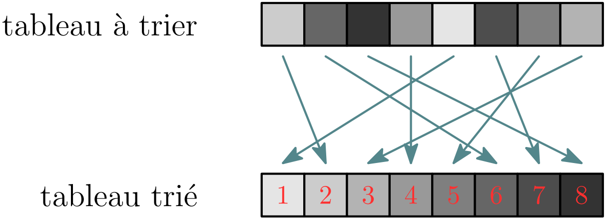


tableau trié

Tris *vs.* permutations



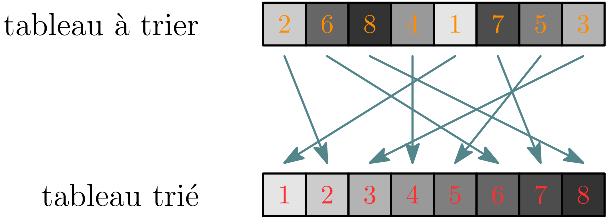


tableau à trier
2 6 8 4 1 7 5 3
tableau trié
1 2 3 4 5 6 7 8

Tris *vs.* permutations

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

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

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

produit par σ^{-1}

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

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

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

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* »

- découper le problème en sous-problèmes de taille inférieure
- résoudre *récursivement* le ou les sous-problèmes
- résoudre le problème initial à l'aide des résultats des sous-problèmes

Tri par fusion

tri utilisant la stratégie « *diviser-pour-régner* »

- scinder la liste à trier en deux, `gauche` et `droite`

- résoudre *récursivement* le ou les sous-problèmes

- résoudre le problème initial à l'aide des résultats des sous-problèmes

Tri par fusion

tri utilisant la stratégie « *diviser-pour-régner* »

- scinder la liste à trier en deux, `gauche` et `droite`

- trier `gauche` et `droite`

- résoudre le problème initial à l'aide des résultats des sous-problèmes

Tri par fusion

tri utilisant la stratégie « *diviser-pour-régner* »

- scinder la liste à trier en deux, `gauche` et `droite`

- trier `gauche` et `droite`

- *fusionner* `gauche` et `droite` en une unique liste triée

tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées

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

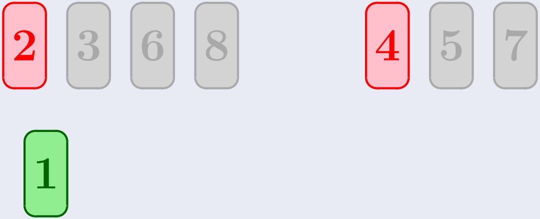
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées

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

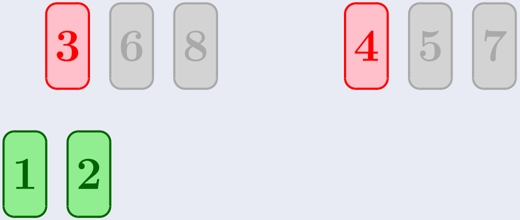
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



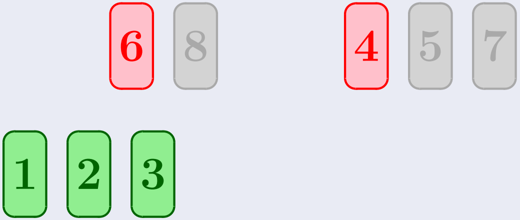
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



tri utilisant la stratégie « *diviser-pour-régner* »

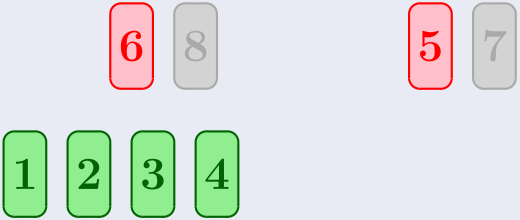
Étape élémentaire : la fusion de listes triées



Tri par fusion

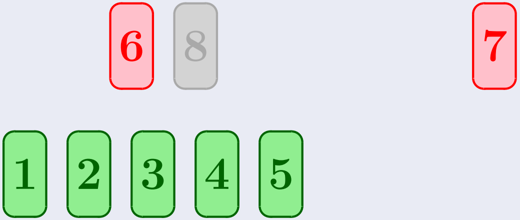
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



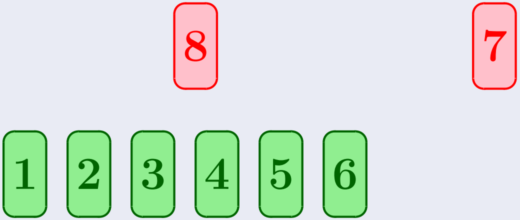
tri utilisant la stratégie « *diviser-pour-régner* »

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

8

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

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

Fusion de deux 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:])
```

Fusion de deux 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:])
```

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

Fusion de deux 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:])
```

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

(ce n'est pas une bonne mesure de la complexité de la fonction écrite ci-dessus : chaque appel récursif travaille sur une ***copie*** *de l'une des deux listes… mais c'est facile à résoudre, soit en dérécursivant la fonction, soit en passant les indices de début et fin en paramètre, et la complexité est alors bien en $\Theta(n)$)*

Tri par fusion

Exemple d'exécution complète :

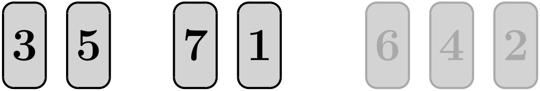
3	**5**	**7**	**1**	**6**	**4**	**2**

Tri par fusion

Exemple d'exécution complète :

3 5 7 1   6 4 2

Exemple d'exécution complète :



Exemple d'exécution complète :

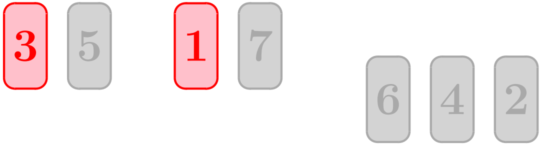
3 **5** 7 1 6 4 2

Exemple d'exécution complète :

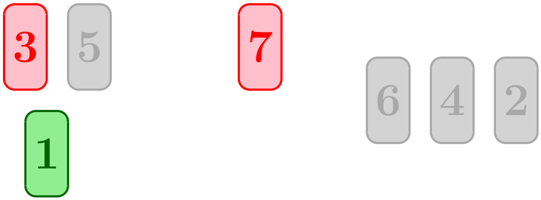
3 5 | 7 1 | 6 4 2

Exemple d'exécution complète :

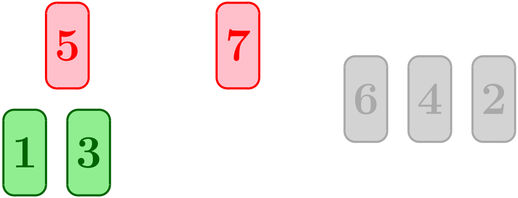
Exemple d'exécution complète :



Exemple d'exécution complète :

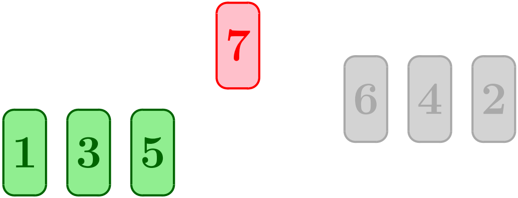


Exemple d'exécution complète :

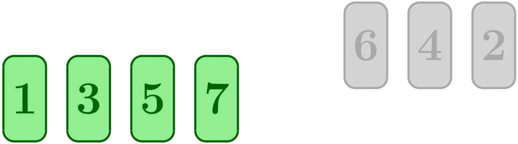


Tri par fusion

Exemple d'exécution complète :



Exemple d'exécution complète :



Tri par fusion

Exemple d'exécution complète :

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

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 :

1 3 5 7 4 6 2

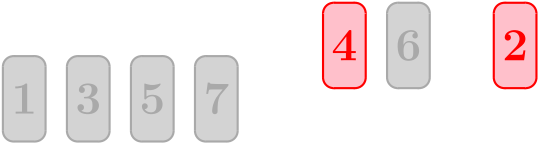
Tri par fusion

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 :

Exemple d'exécution complète :

Exemple d'exécution complète :

1 3 5 7   2 4 6

Exemple d'exécution complète :

1 3 5 7

2 4 6

Exemple d'exécution complète :

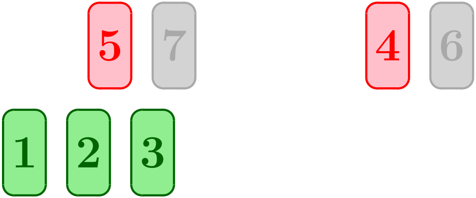
3 5 7   2 4 6

1

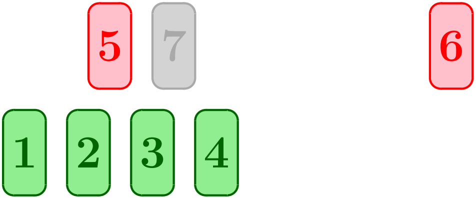
Tri par fusion

Exemple d'exécution complète :

Exemple d'exécution complète :

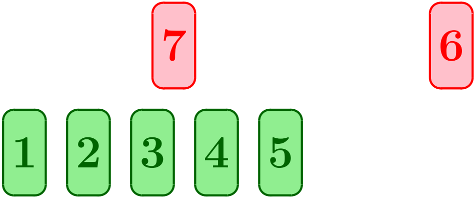


Exemple d'exécution complète :

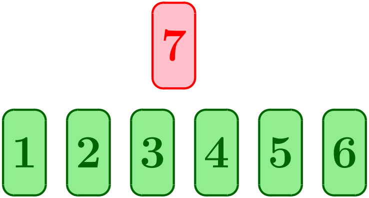


Tri par fusion

Exemple d’exécution complète :



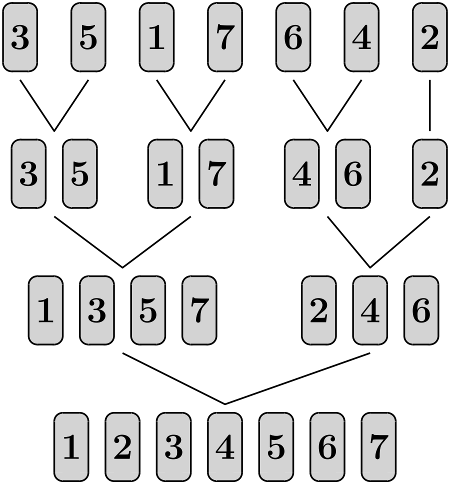
Exemple d'exécution complète :



Exemple d'exécution complète :

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

Récapitulatif des étapes de fusion :



Tri par fusion

```
def tri_fusion(T) :   # attention, 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) :   # attention, 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...)

```
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)
```

```
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

$C(n)$: nombre de comparaisons nécessaires pour trier T de taille n

$$C(n) = 2 \times C(n//2) + \Theta(n)$$

```
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)
```

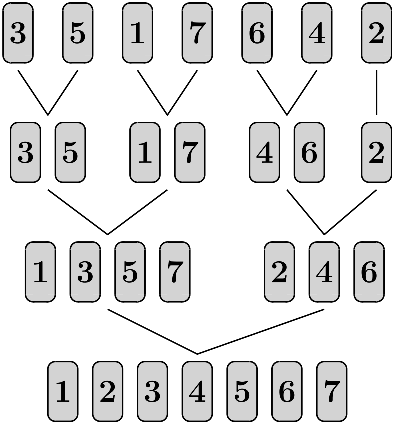
Complexité

$C(n)$: nombre de comparaisons nécessaires pour trier T de taille n

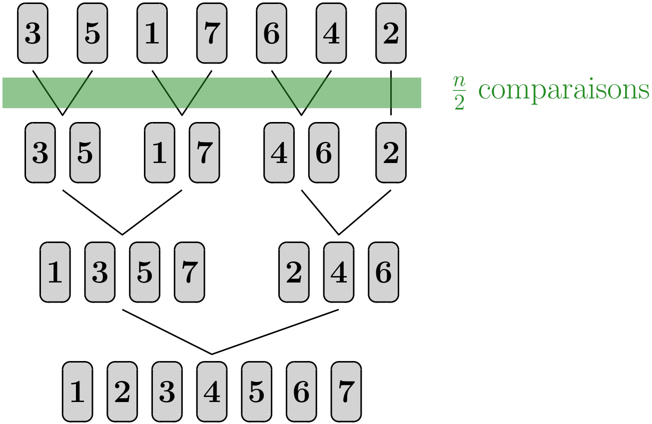
$$C(n) = 2 \times C(n//2) + \Theta(n)$$

Et donc ? ? ?

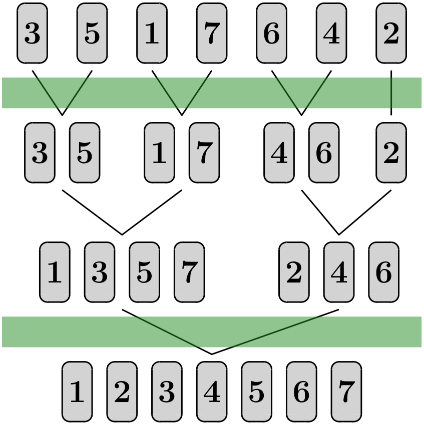
Complexité du tri par fusion



Complexité du tri par fusion



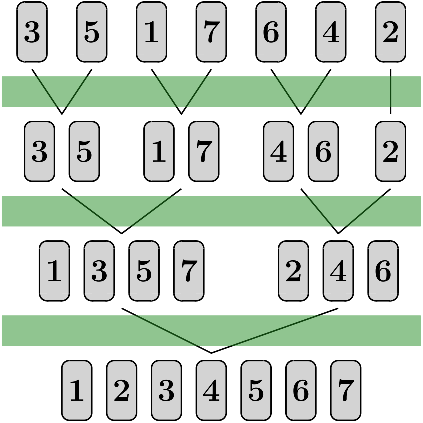
Complexité du tri par fusion



$\frac{n}{2}$ comparaisons

$n-1$ comparaisons (au plus)

Complexité du tri par fusion

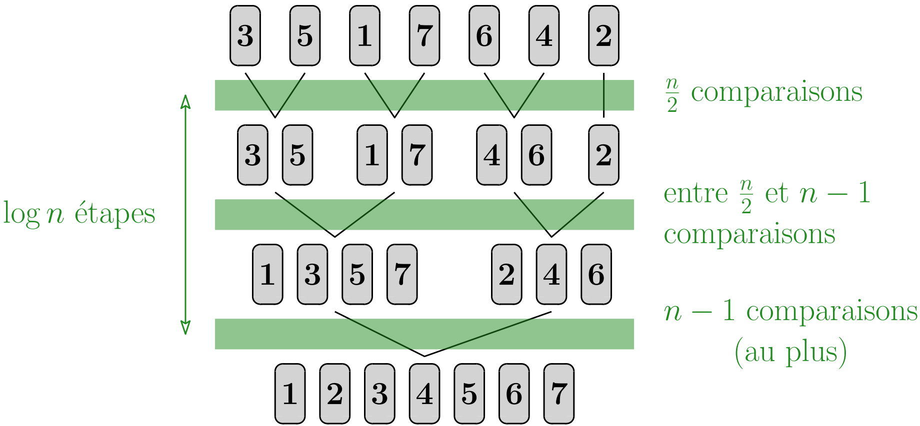


$\frac{n}{2}$ comparaisons

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

$n-1$ comparaisons (au plus)

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 comparaisons ***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 comparaisons ***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 $\in \Theta(n)$

Tri par fusion

- $\Theta(n \log n)$ comparaisons au pire (mais ***dans tous les cas***),
- la ***constante cachée*** dans le Θ est importante,
- ne trie ***pas en place*** : complexité en espace $\in \Theta(n)$

Tri par insertion

- $\Theta(n^2)$ comparaisons ***au pire***, $\Theta(n)$ au mieux,
- trie en place

Fusion *vs* insertion

Tri par fusion

- $\Theta(n \log n)$ comparaisons au pire (mais ***dans tous les cas***),
- la ***constante cachée*** dans le Θ est importante,
- ne trie ***pas en place*** : complexité en espace $\in \Theta(n)$

Tri par insertion

- $\Theta(n^2)$ comparaisons ***au pire***, $\Theta(n)$ au mieux,
- trie en place

- quid de la complexité en moyenne du tri par insertion ?
- dans quels cas trie-t-il en $\Theta(n)$?
- existe-t-il un algorithme de meilleure complexité en moyenne que le tri fusion ?
- ... et qui trie en place ?