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 ?

Permutations

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

σ

$1 \longmapsto 4$

$2 \longmapsto 2$

$3 \longmapsto 1$

$4 \longmapsto 6$

$5 \longmapsto 7$

$6 \longmapsto 3$

$7 \longmapsto 5$

Exemple – représentations diverses

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

Exemple – représentations diverses

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

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

Exemple – représentations diverses

σ

$1 \longmapsto 4$

$2 \longmapsto 2$

$3 \longmapsto 1$

$4 \longmapsto 6$

$5 \longmapsto 7$

$6 \longmapsto 3$

$7 \longmapsto 5$

$\Leftrightarrow \quad \sigma =$

$1 \longmapsto 4$

Exemple – représentations diverses

σ

$1 \longmapsto 4$

$2 \longmapsto 2$

$3 \longmapsto 1$

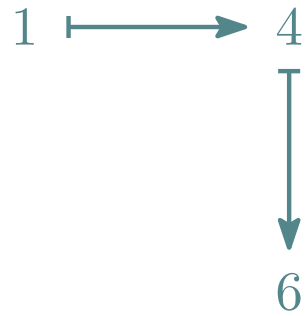
$4 \longmapsto 6$

$5 \longmapsto 7$

$6 \longmapsto 3$

$7 \longmapsto 5$

$\Leftrightarrow \quad \sigma =$



Exemple – représentations diverses

σ

$1 \longmapsto 4$

$2 \longmapsto 2$

$3 \longmapsto 1$

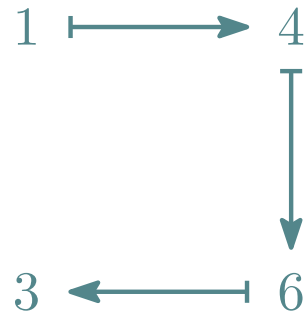
$4 \longmapsto 6$

$5 \longmapsto 7$

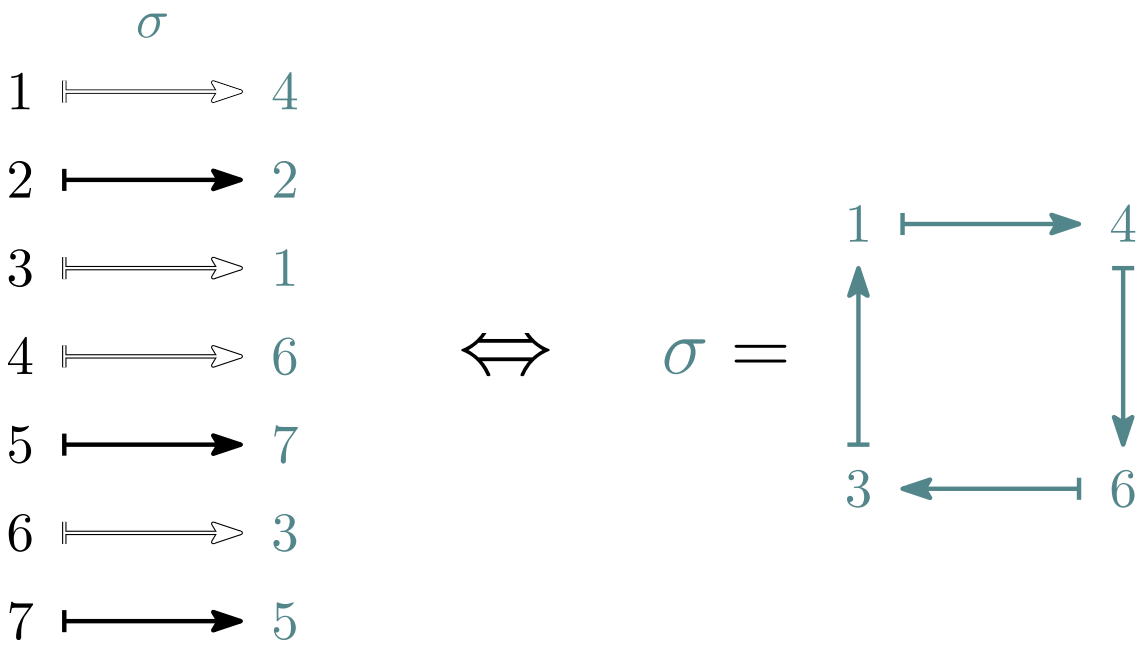
$6 \longmapsto 3$

$7 \longmapsto 5$

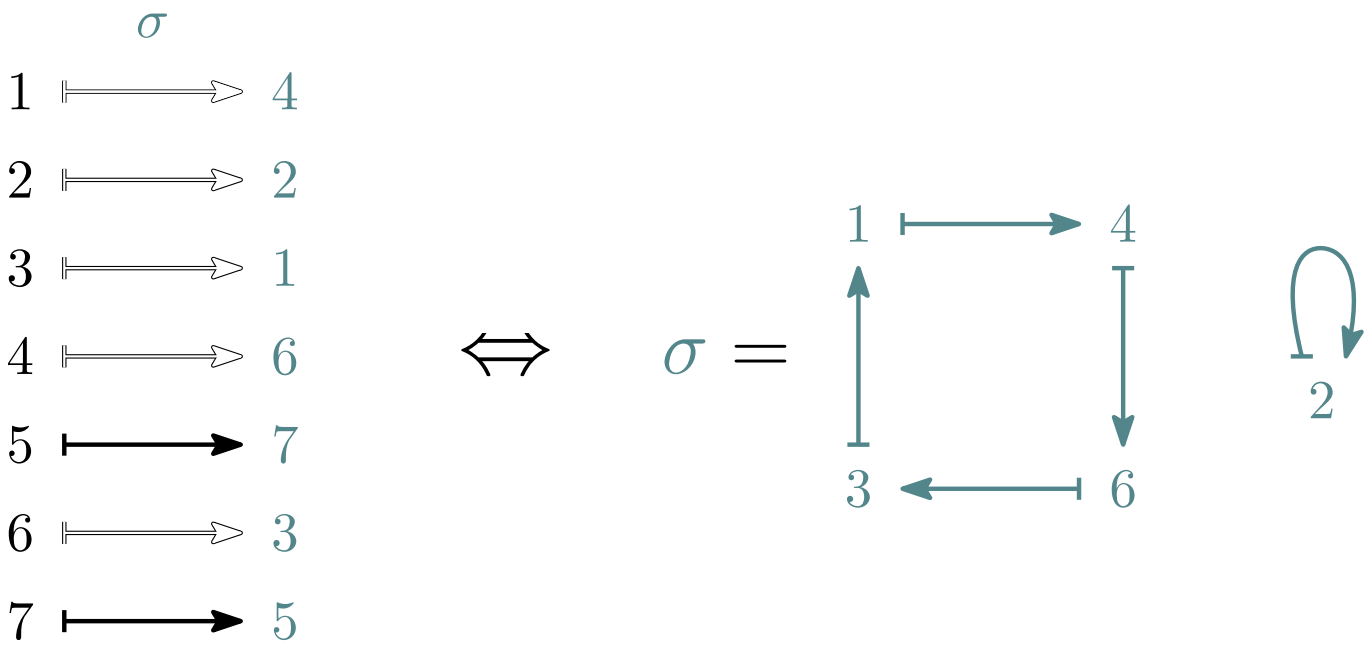
$\Leftrightarrow \quad \sigma =$



Exemple – représentations diverses



Exemple – représentations diverses

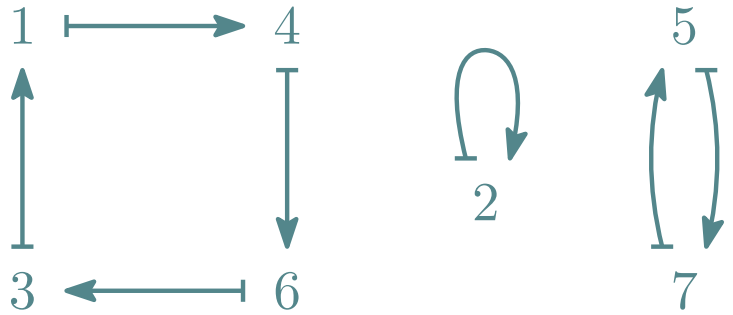


Exemple – représentations diverses

σ

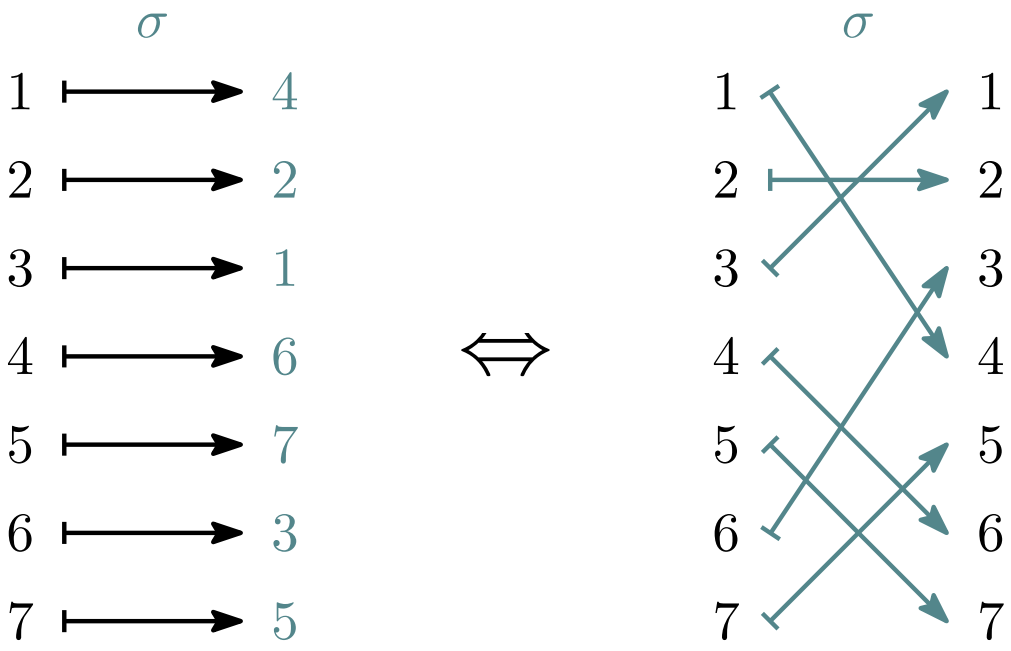
$1 \longmapsto 4$
$2 \longmapsto 2$
$3 \longmapsto 1$
$4 \longmapsto 6$
$5 \longmapsto 7$
$6 \longmapsto 3$
$7 \longmapsto 5$

$\Leftrightarrow \quad \sigma =$



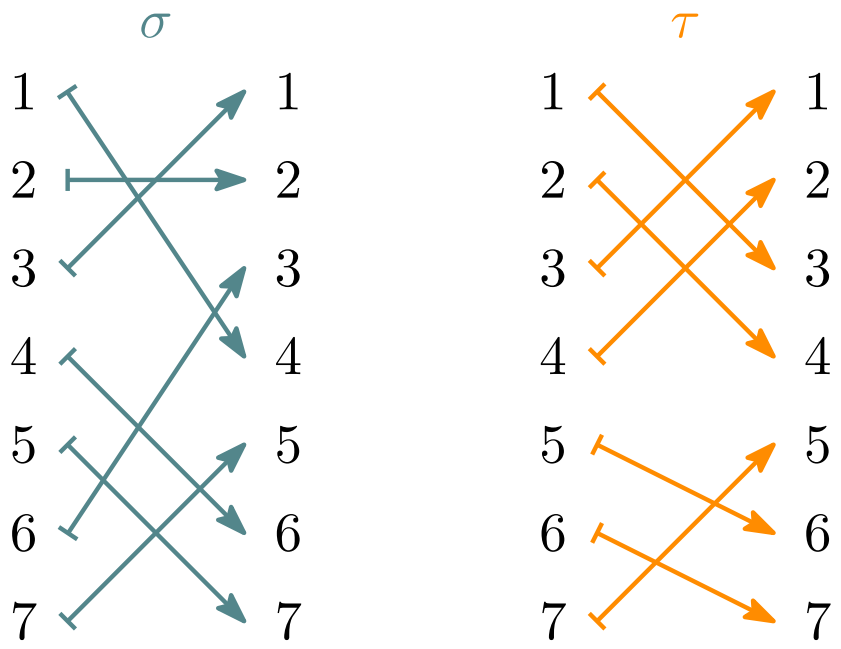
notation cyclique : sur cet exemple, $\sigma = (1\ 4\ 6\ 3)\ (2)\ (5\ 7)$,
ou plus simplement : $\sigma = (1\ 4\ 6\ 3)\ (5\ 7)$

Exemple – représentations diverses



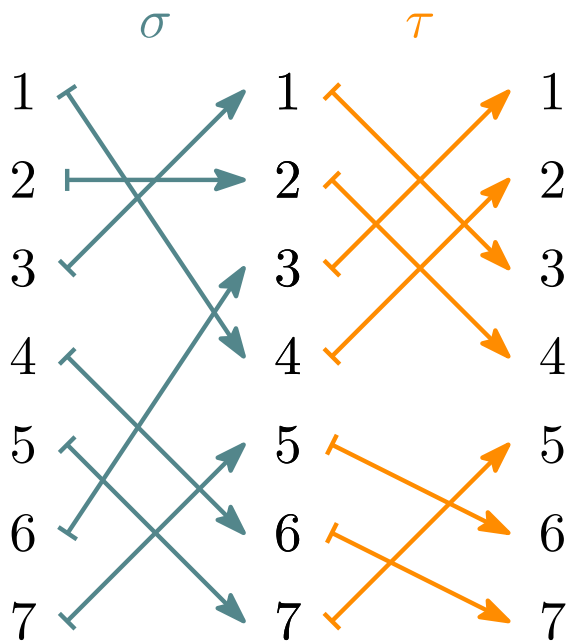
Produit de permutations

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



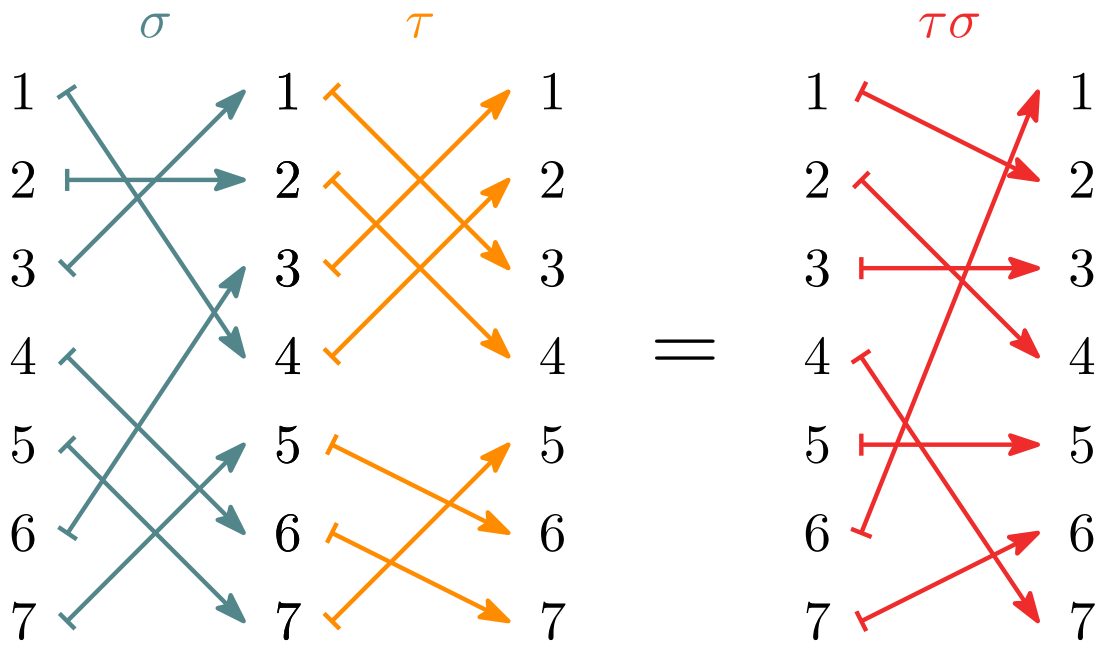
Produit de permutations

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



Produit de permutations

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

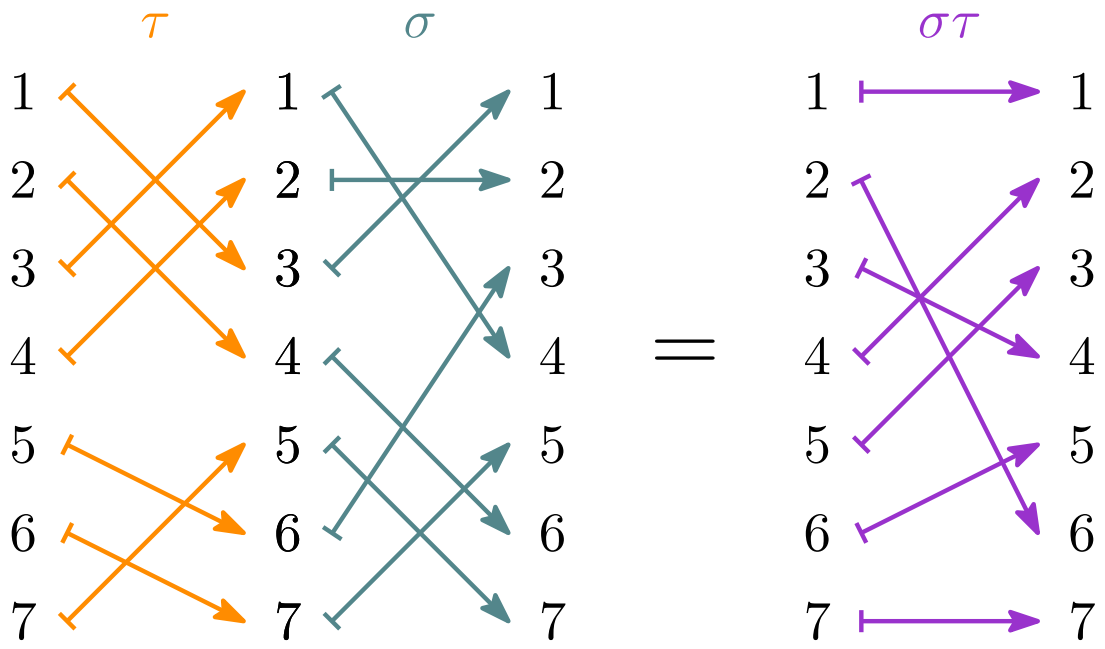


Lemme

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

Produit de permutations

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



Lemme

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

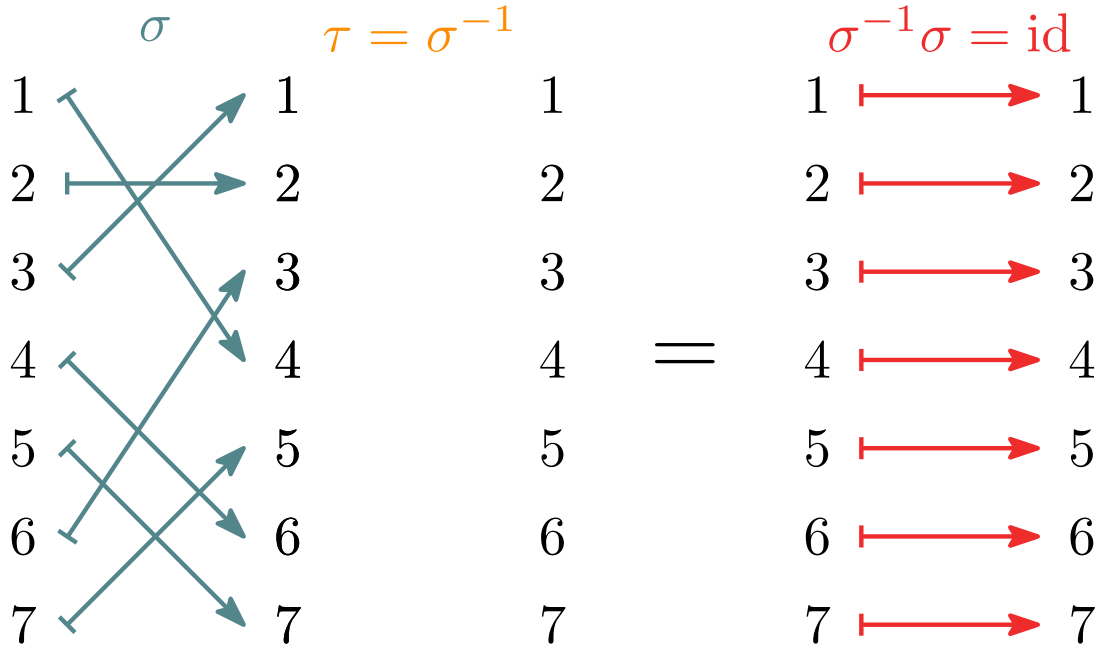
attention, le produit n'est pas commutatif !

Permutation inverse

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

notation : σ^{-1}

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

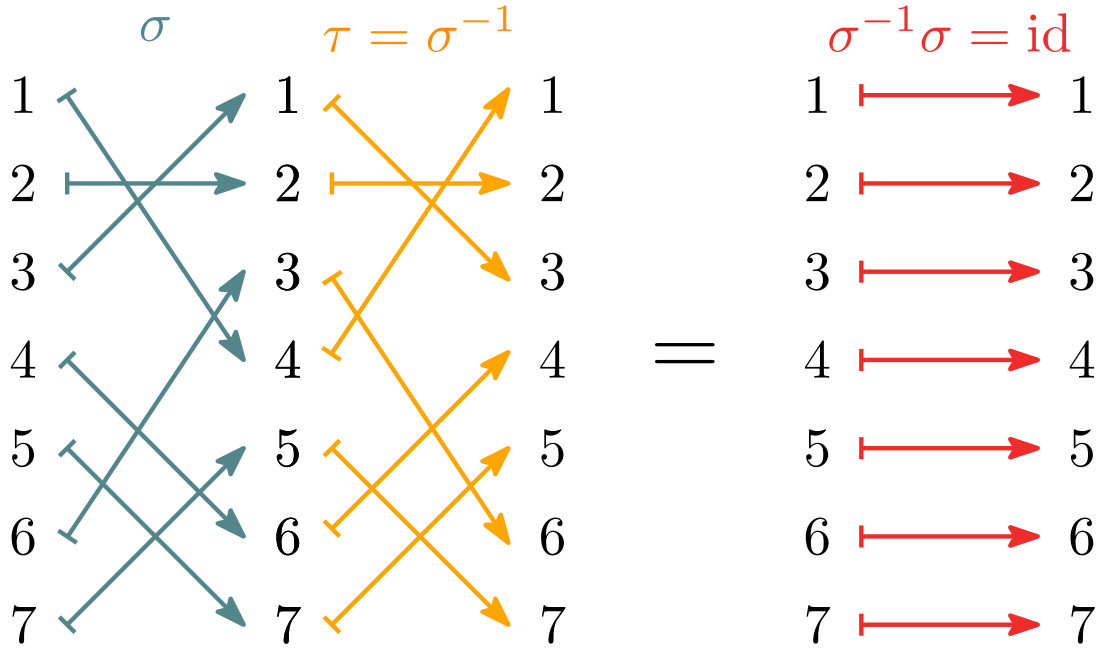


Permutation inverse

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

notation : σ^{-1}

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

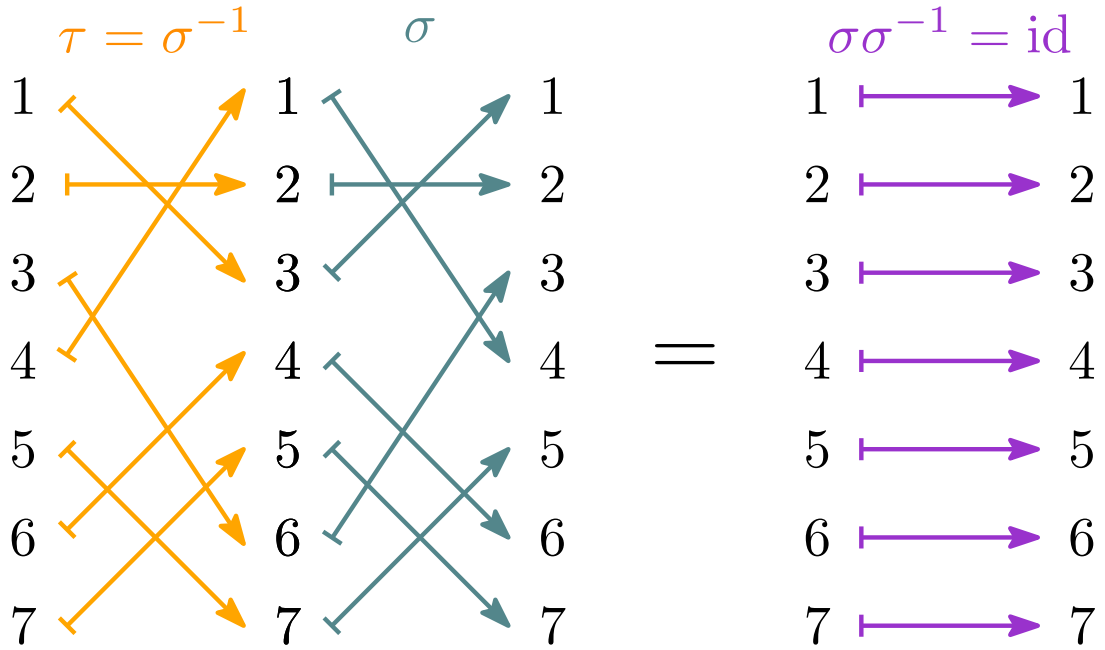


Permutation inverse

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

notation : σ^{-1}

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

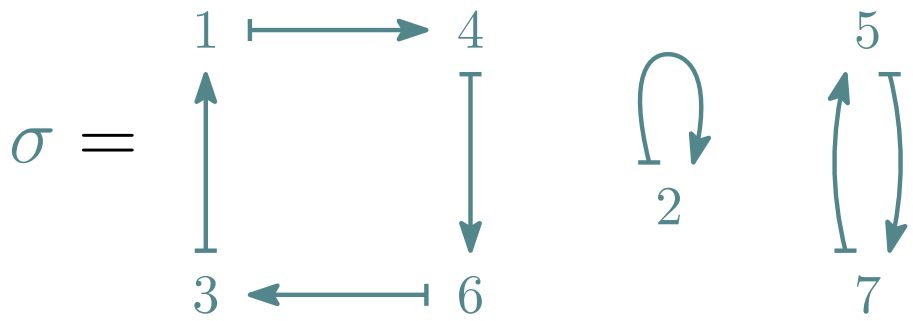


Permutation inverse

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

notation : σ^{-1}

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

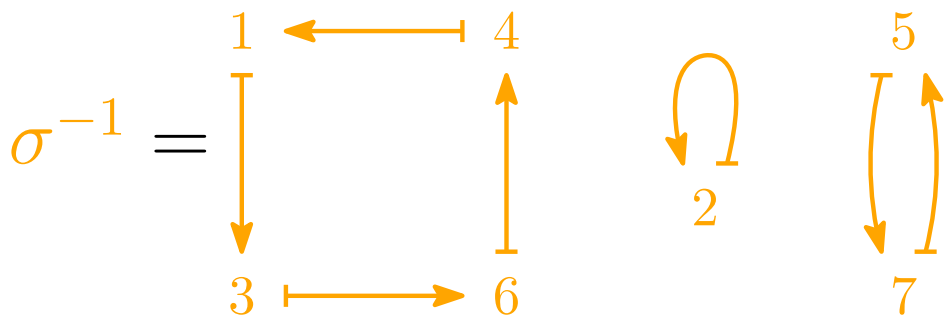


Permutation inverse

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

notation : σ^{-1}

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



Lemme

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

(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

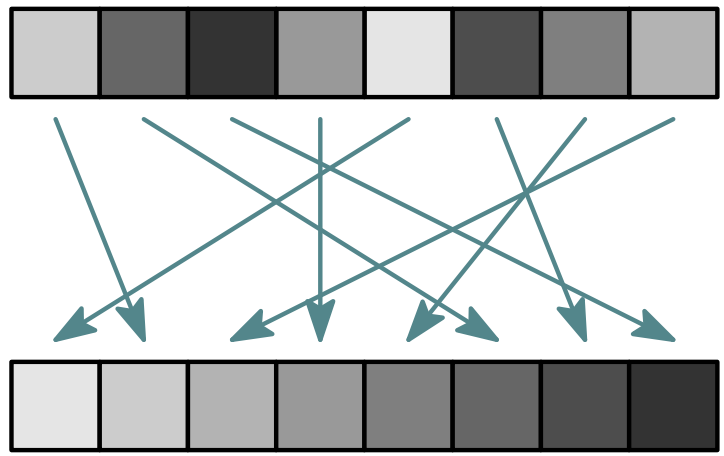


tableau trié

Tris *vs.* permutations

tableau à trier

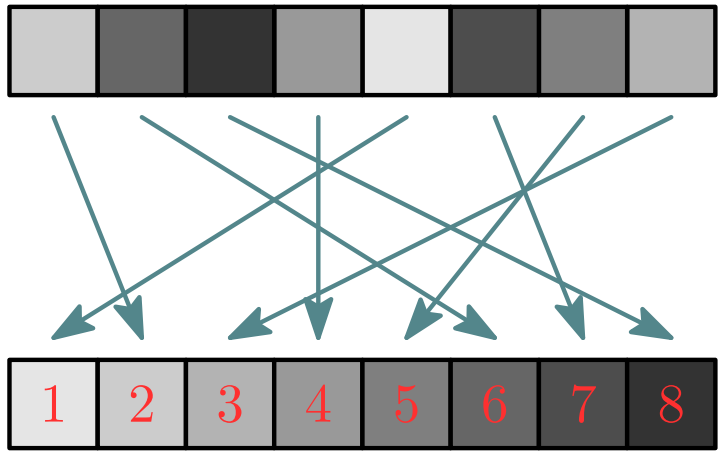


tableau trié

Tris *vs.* permutations

tableau à trier

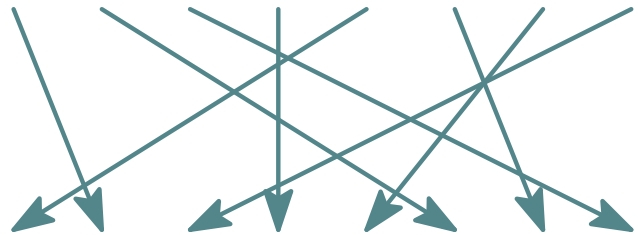
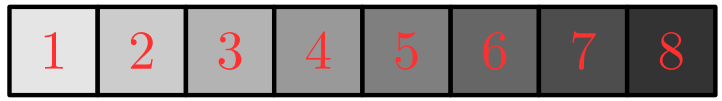
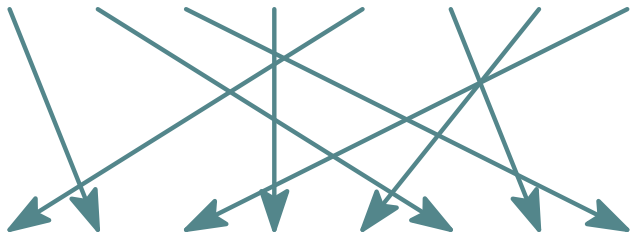


tableau trié



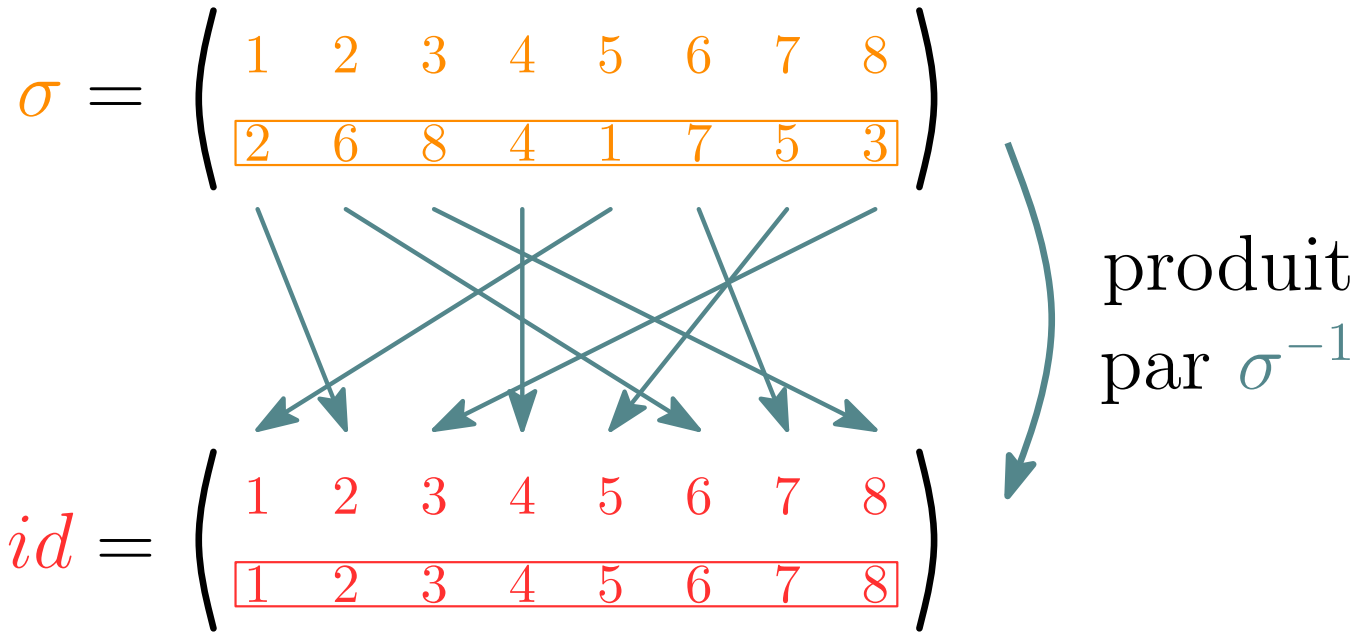
Tris *vs.* permutations

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

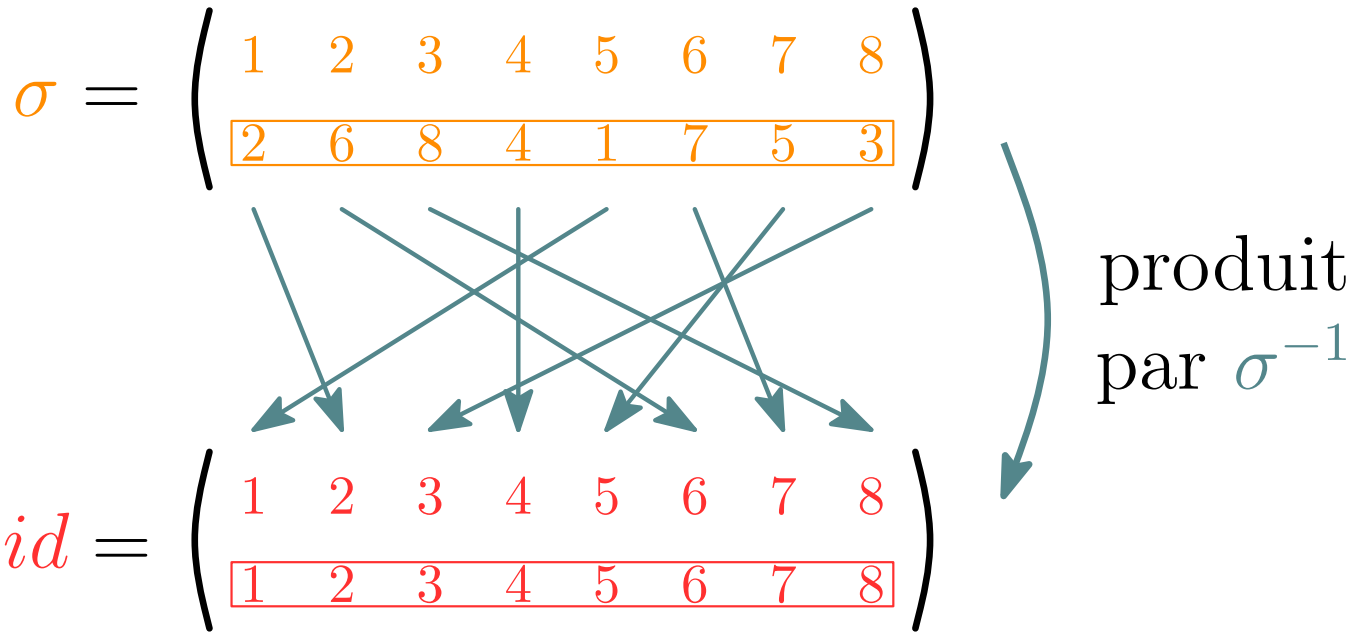


$$id = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}$$

Tris *vs.* permutations



Tris *vs.* permutations



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

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* »

- 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

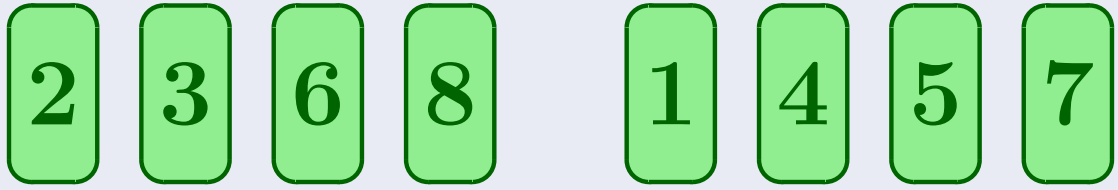
Tri par fusion

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

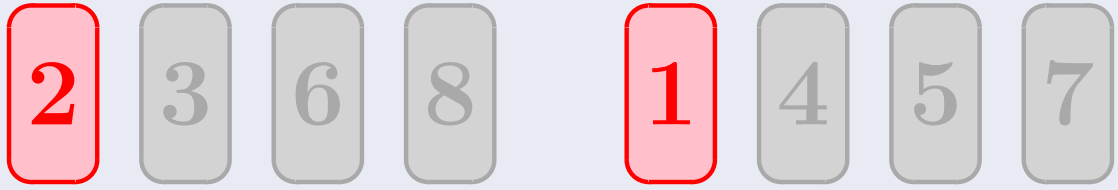
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



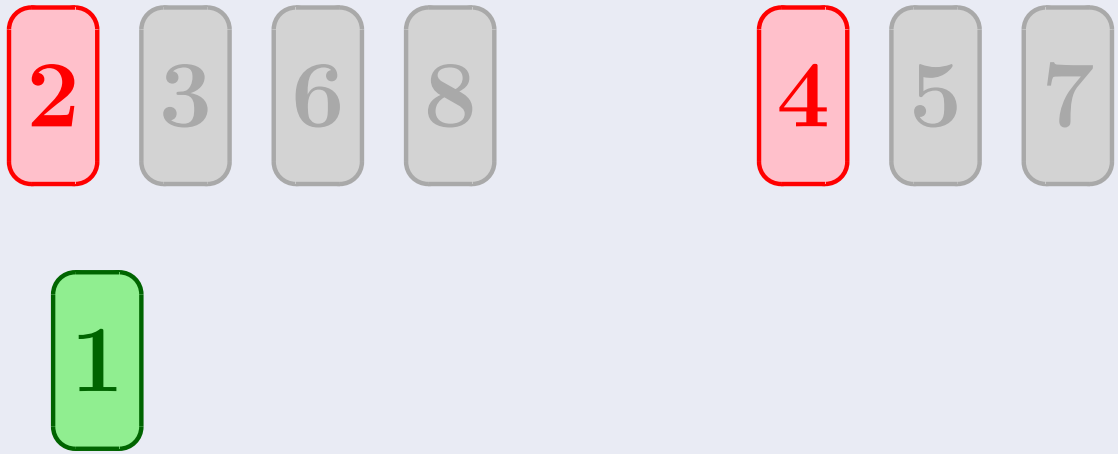
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



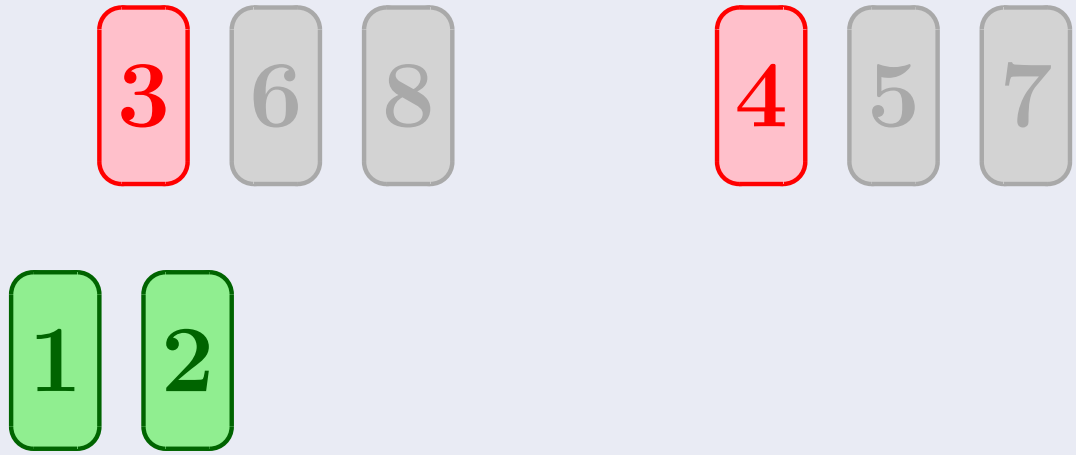
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



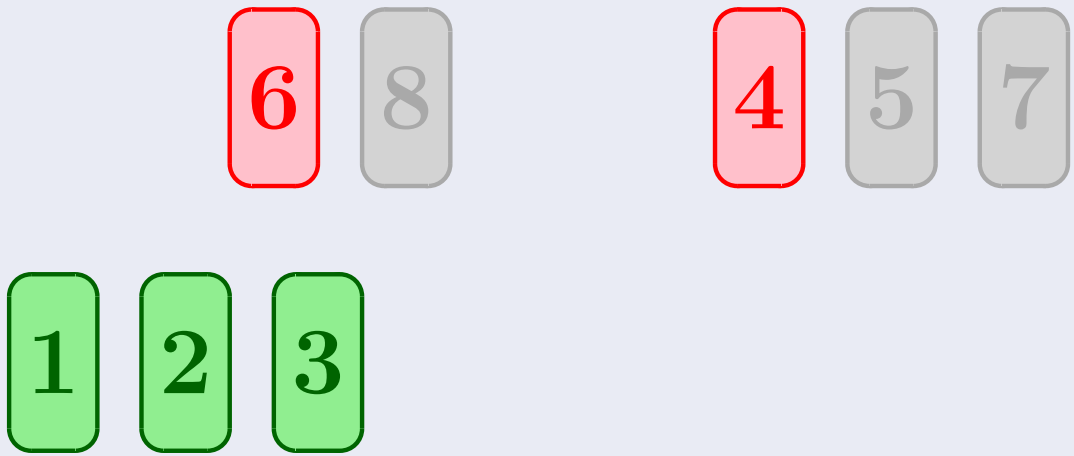
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



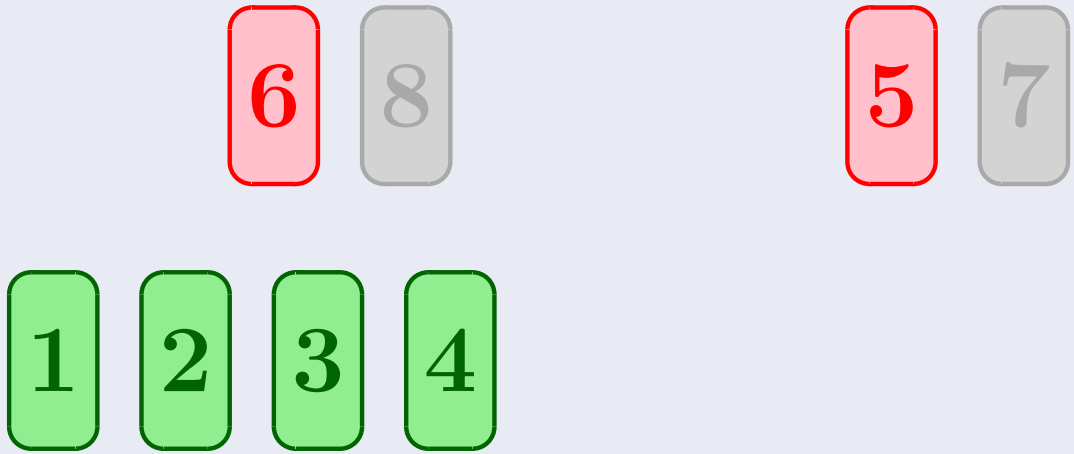
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



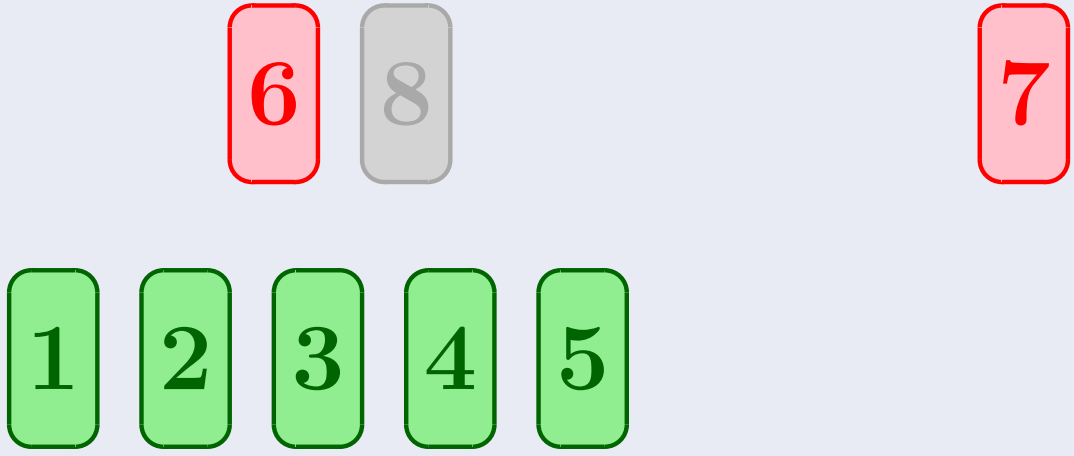
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



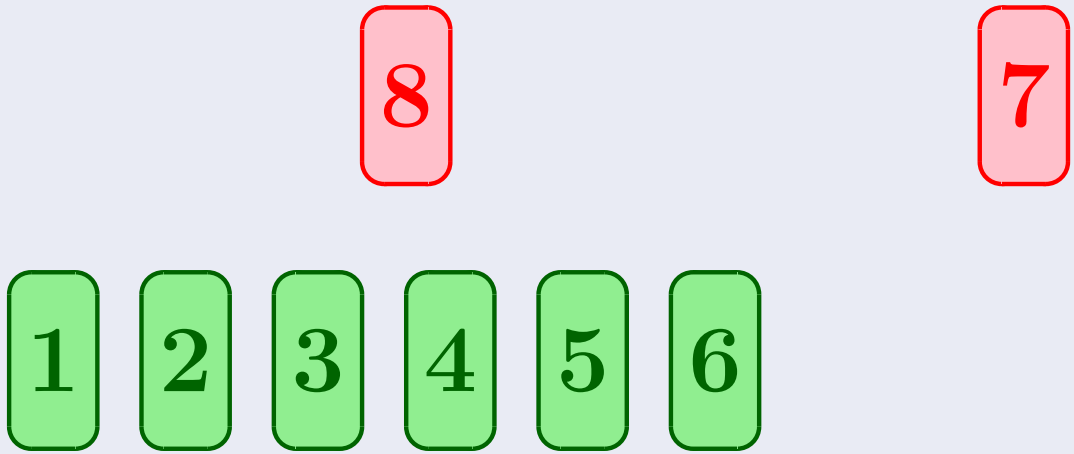
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



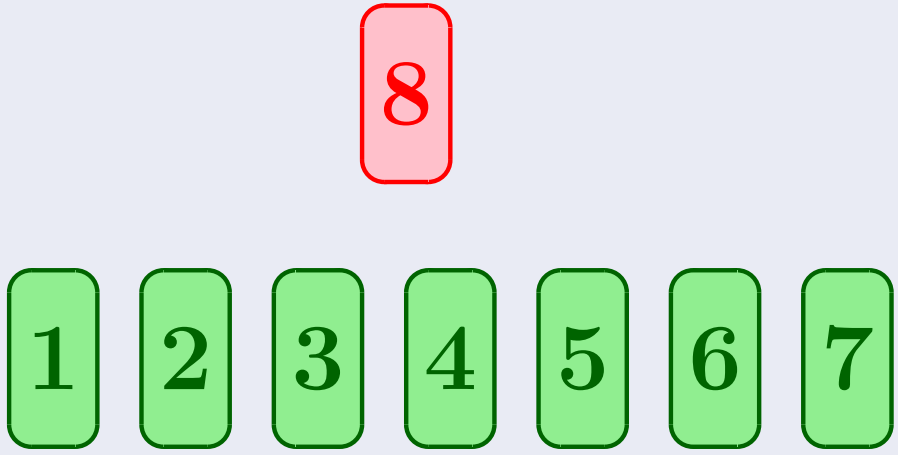
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



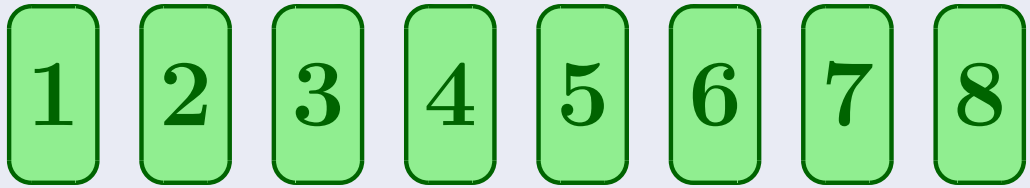
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



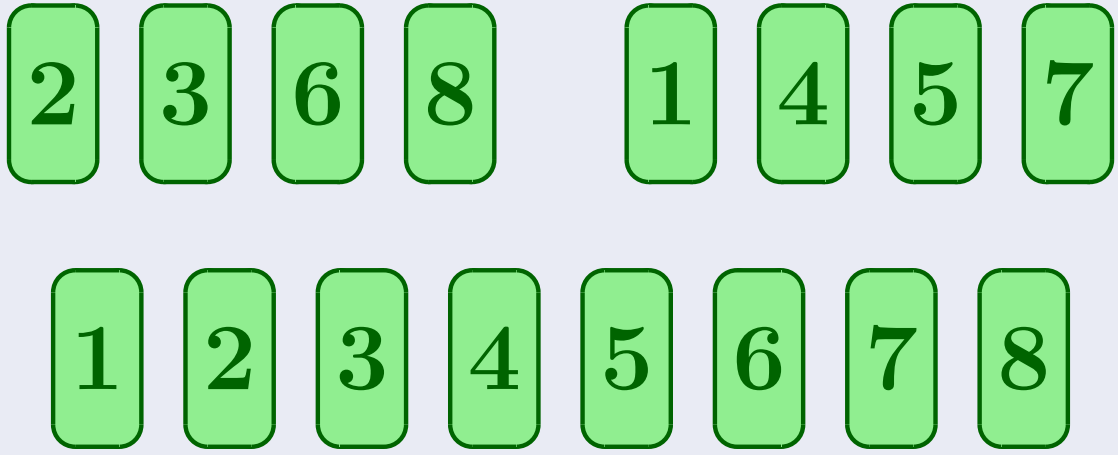
tri utilisant la stratégie « *diviser-pour-régner* »

Étape élémentaire : la fusion de listes triées



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

Étape élémentaire : la fusion de listes triées



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

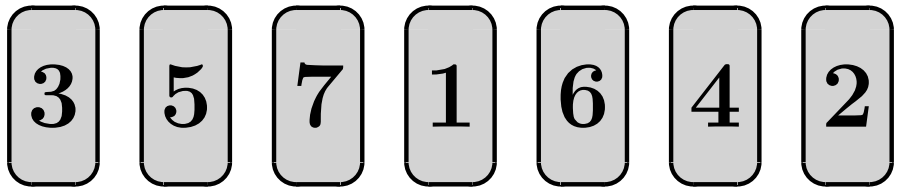
Fusion de deux listes triées

```
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

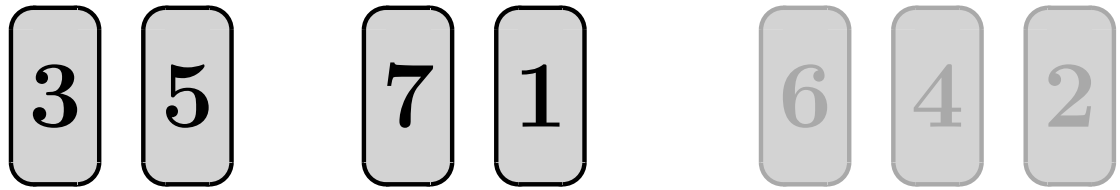
(ce n'est pas une bonne mesure de la complexité de la fonction écrite ci-dessus : chaque appel récursif travaille sur une ***copie*** *de l'une des deux listes... mais c'est facile à résoudre, soit en dérécursivant la fonction, soit en passant les indices de début et fin en paramètre, et la complexité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

Exemple d'exécution complète :



Exemple d'exécution complète :

Exemple d'exécution complète :

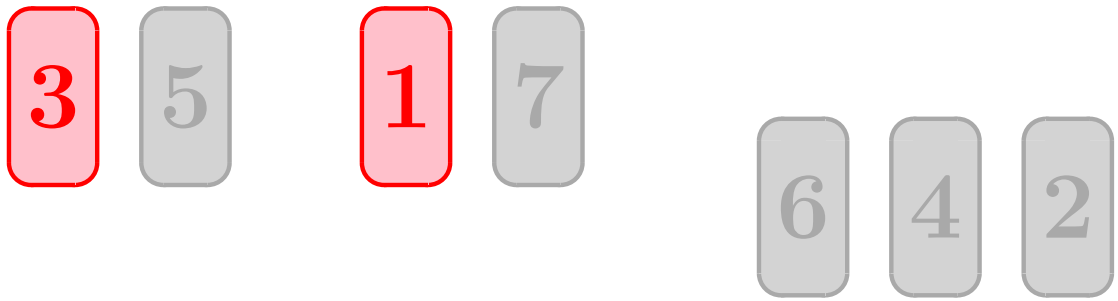


Exemple d'exécution complète :

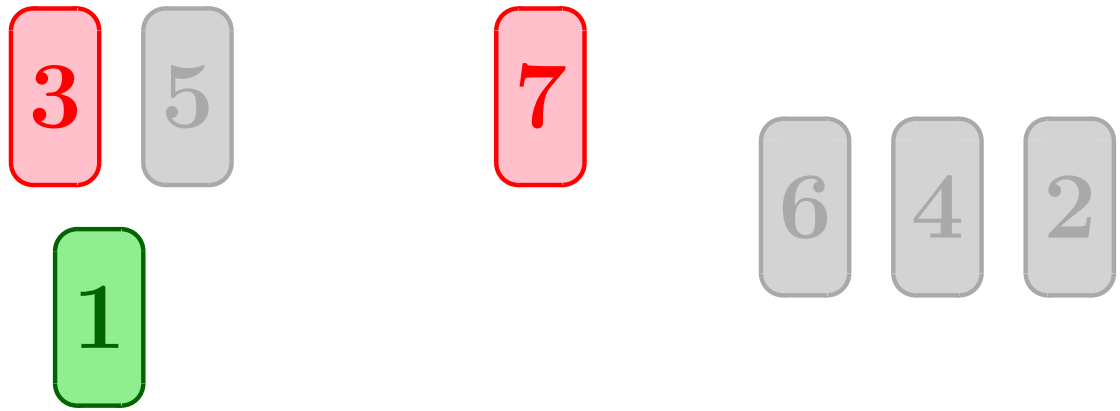
Exemple d'exécution complète :

Exemple d'exécution complète :

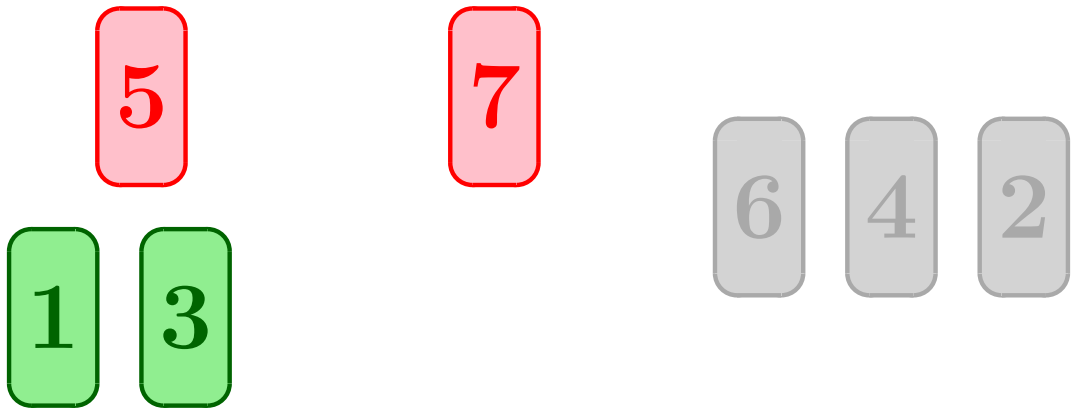
Exemple d'exécution complète :



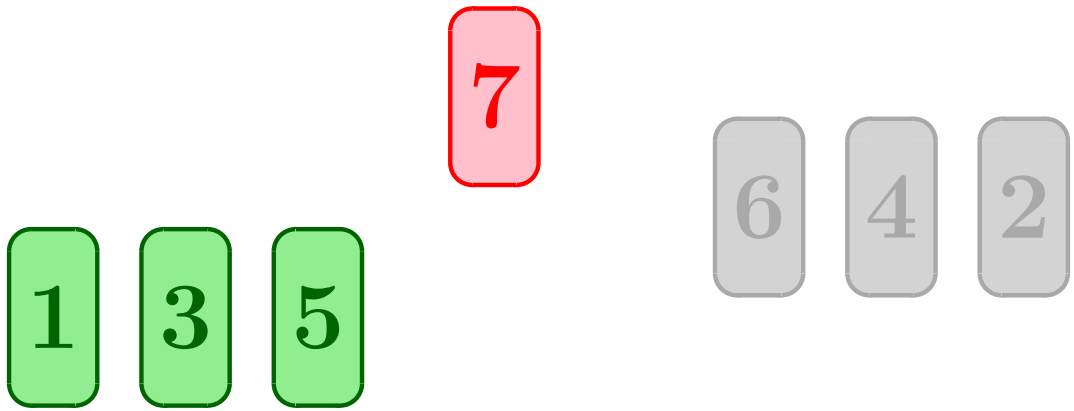
Exemple d'exécution complète :



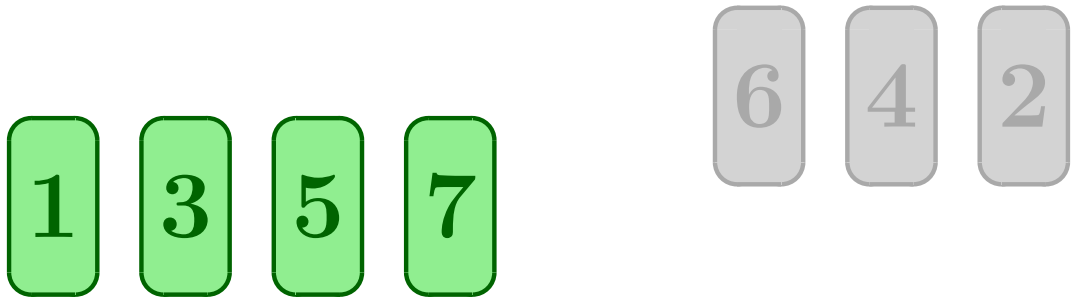
Exemple d'exécution complète :



Exemple d'exécution complète :



Exemple d'exécution complète :



Exemple d'exécution complète :

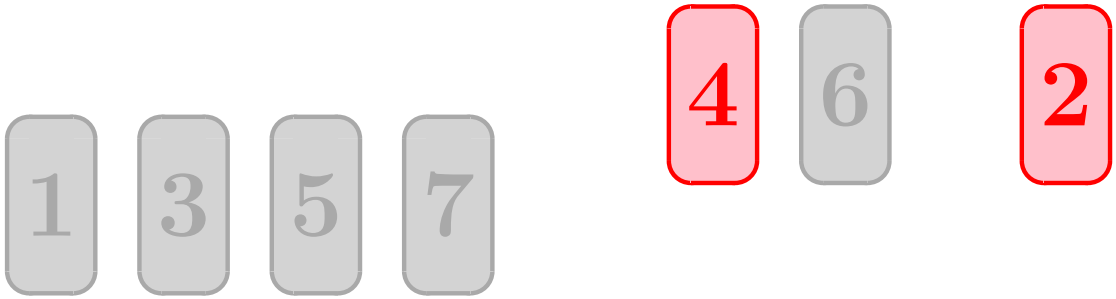
Exemple d'exécution complète :

Exemple d'exécution complète :

Exemple d'exécution complète :

Exemple d'exécution complète :

Exemple d'exécution complète :



Exemple d'exécution complète :

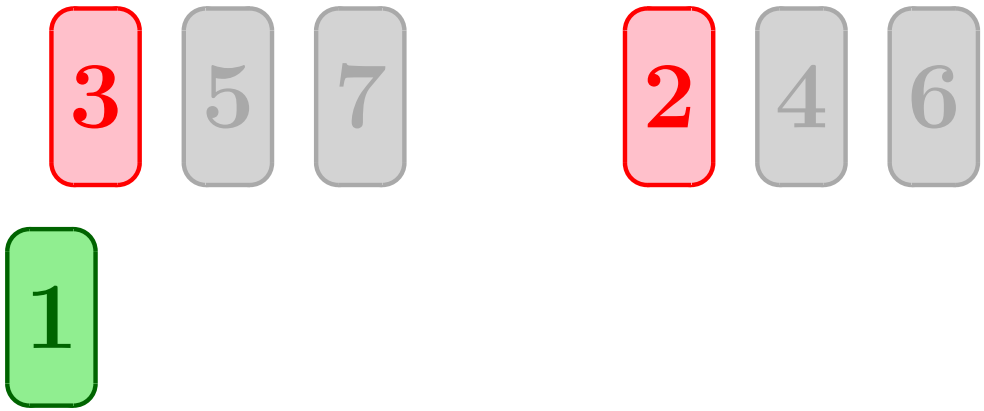
Exemple d'exécution complète :

Exemple d'exécution complète :

Exemple d'exécution complète :

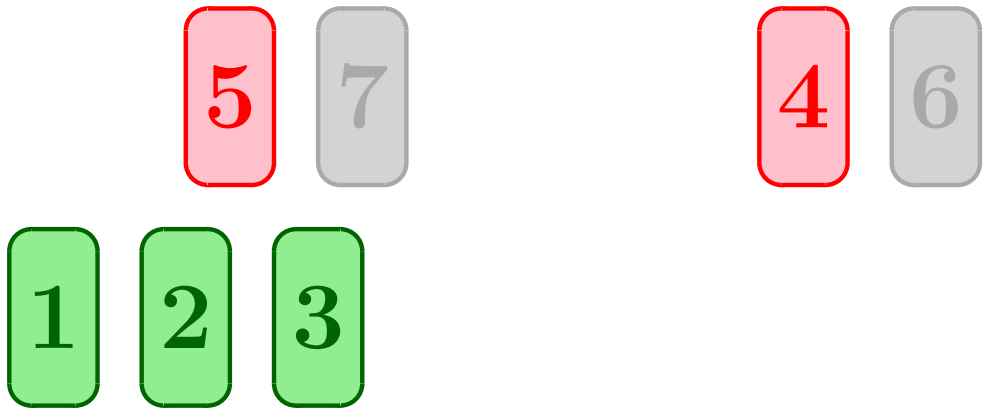
Exemple d'exécution complète :

Exemple d'exécution complète :

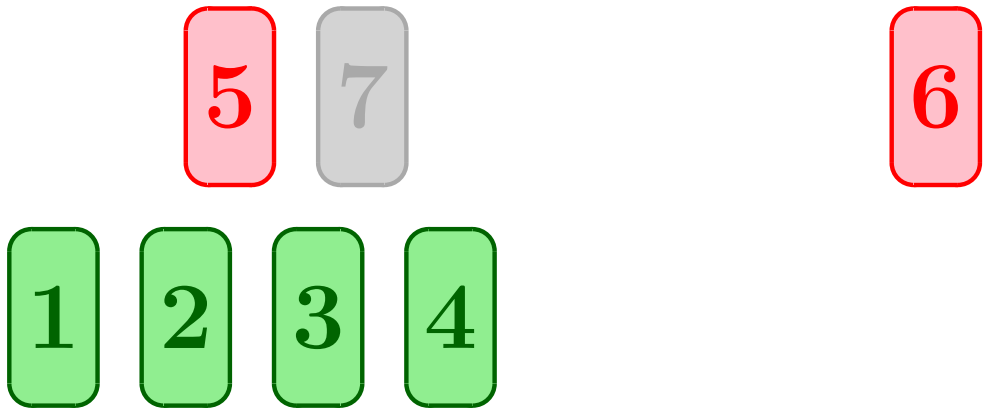


Exemple d'exécution complète :

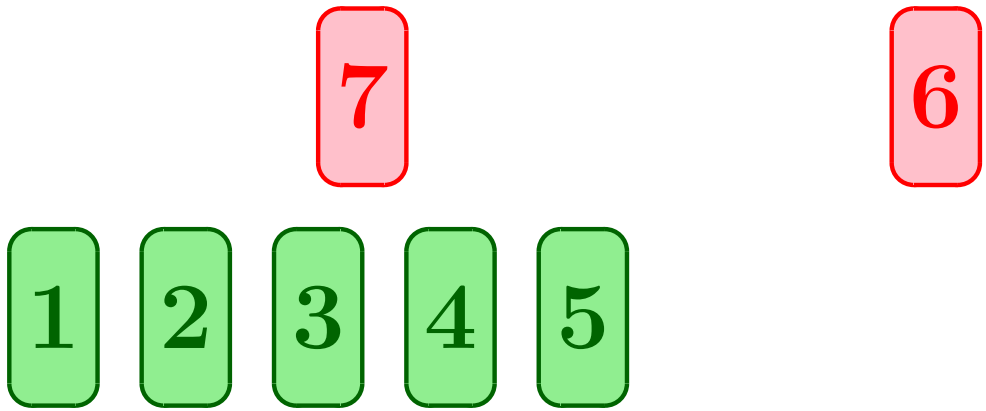
Exemple d'exécution complète :



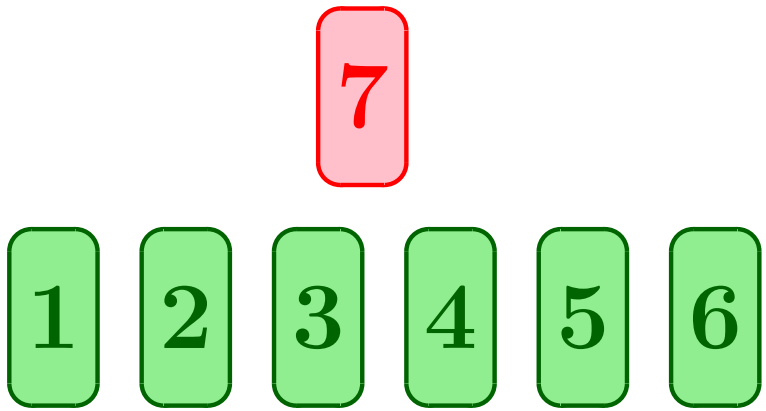
Exemple d'exécution complète :



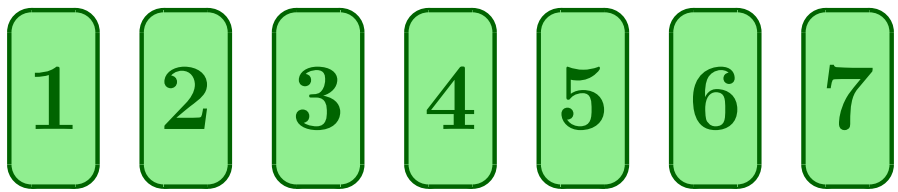
Exemple d'exécution complète :



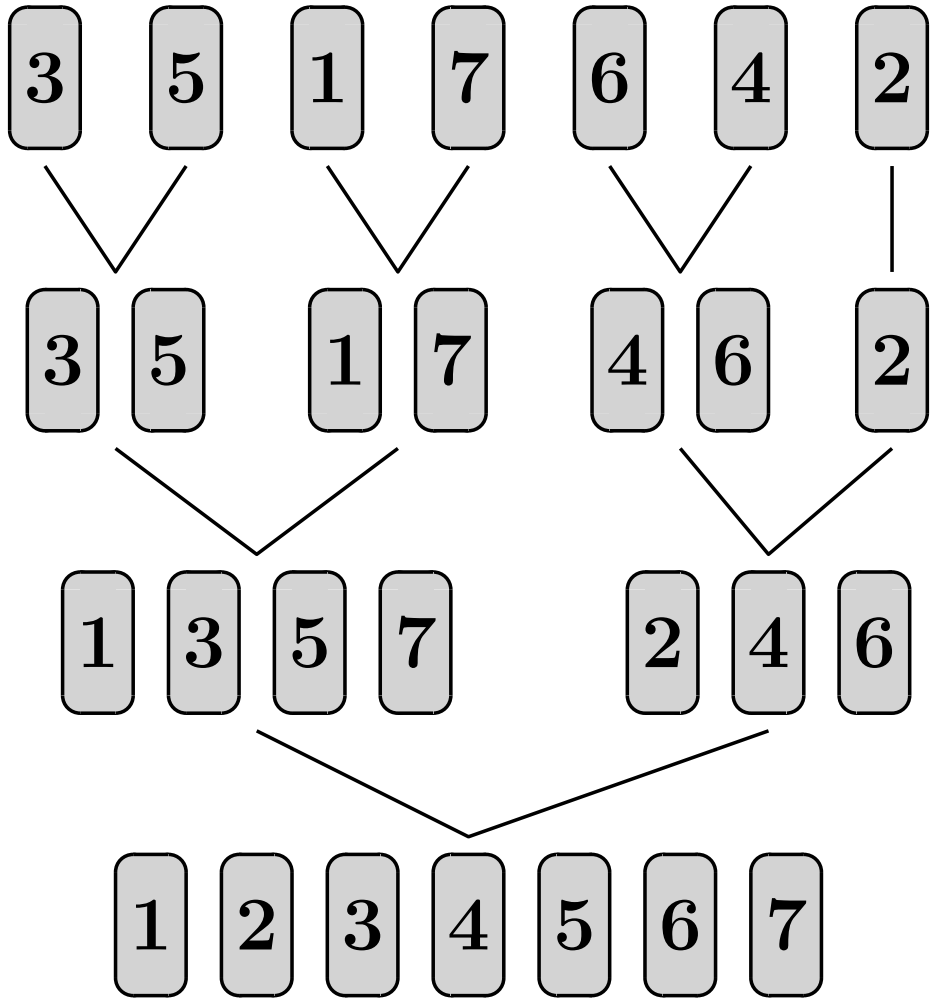
Exemple d'exécution complète :



Exemple d'exécution complète :



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)
```

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)
```

(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)
```

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

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

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

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)
```

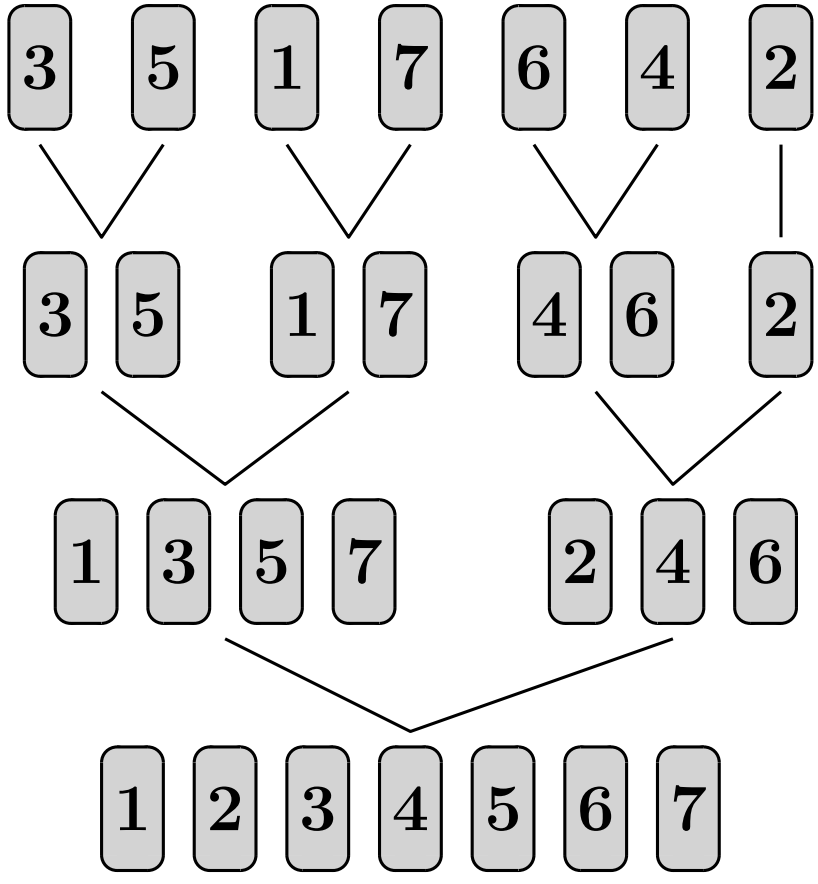
Complexité

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

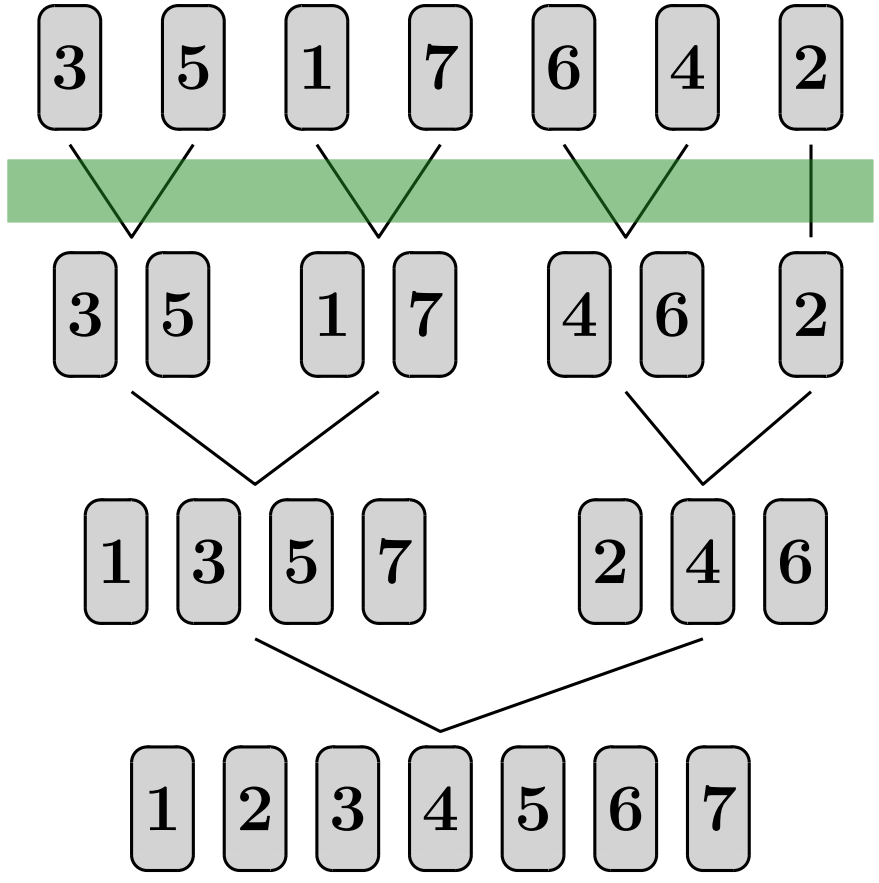
$$C(n) = 2 \times C(n//2) + \Theta(n)$$

Et donc ? ? ?

Complexité du tri par fusion

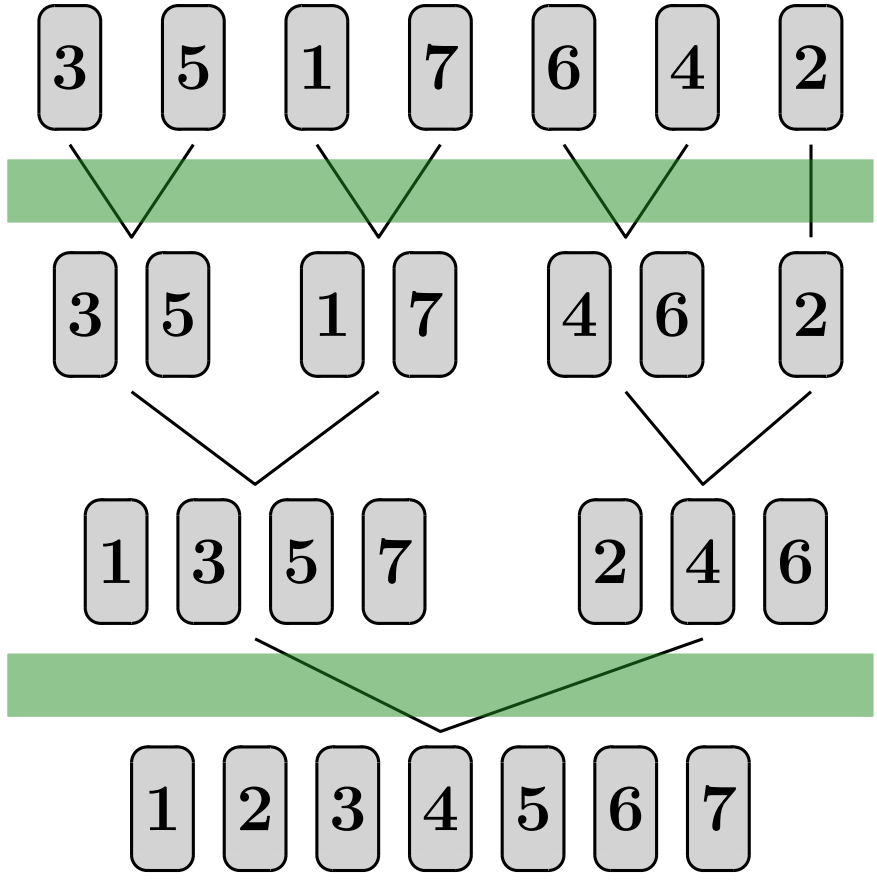


Complexité du tri par fusion



$\frac{n}{2}$ comparaisons

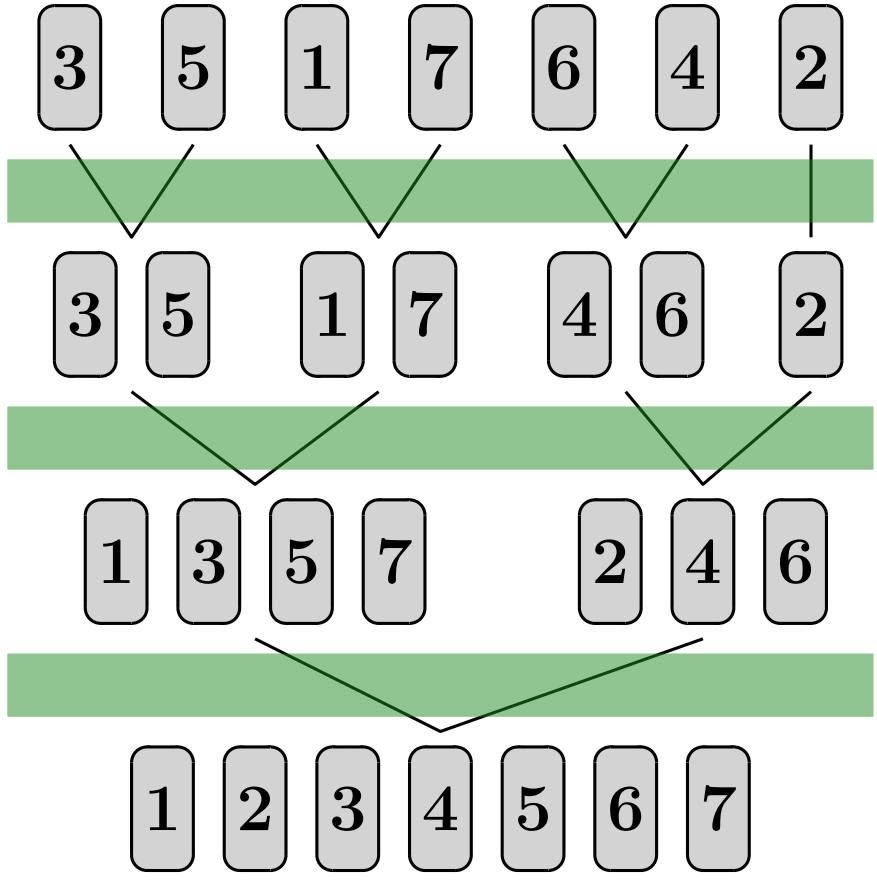
Complexité du tri par fusion



$\frac{n}{2}$ comparaisons

$n - 1$ comparaisons

Complexité du tri par fusion

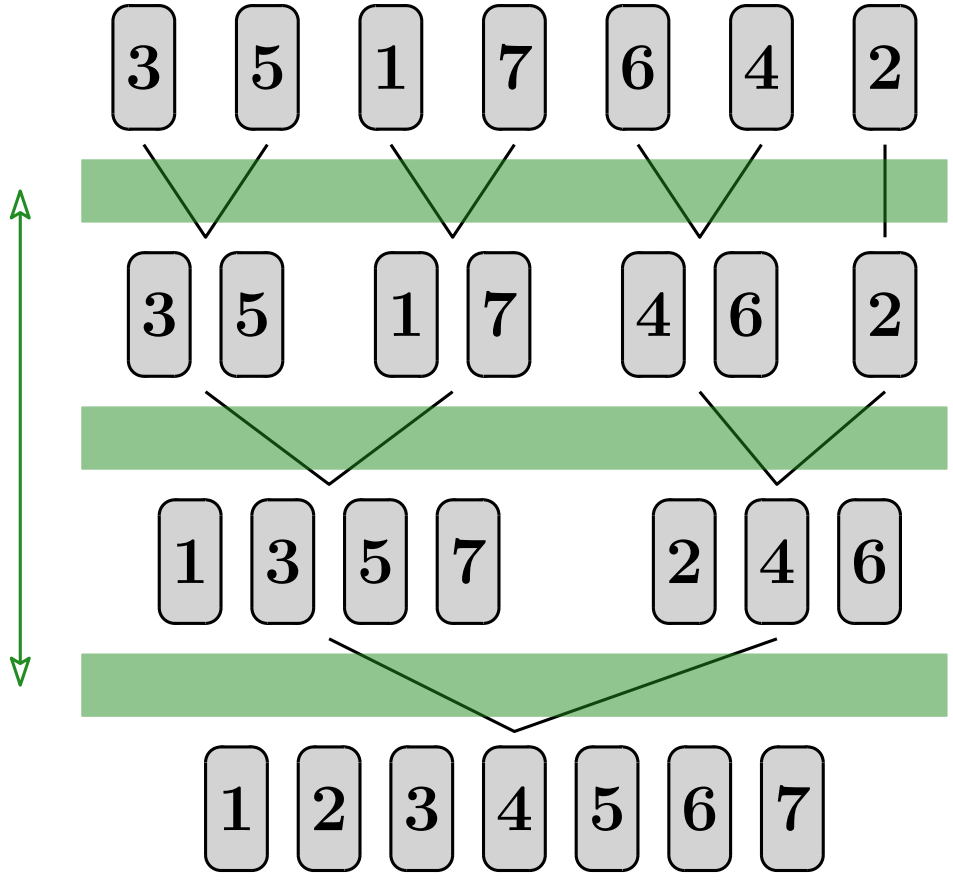


$\frac{n}{2}$ comparaisons

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

$n-1$ comparaisons

Complexité du tri par fusion



$\log n$ étapes

$\frac{n}{2}$ comparaisons

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

$n-1$ comparaisons

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 $\in \Theta(n)$