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 2023-2024

Rappel des épisodes précédents

Borne inférieure de complexité

tout algorithme de tri par comparaisons nécessite $\Omega(n \log n)$ comparaisons dans le pire cas (et en moyenne)

Rappel des épisodes précédents

Borne inférieure de complexité

tout algorithme de tri par comparaisons nécessite $\Omega(n \log n)$ comparaisons dans le pire cas (et en moyenne)

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 $\in \Theta(n)$

Rappel des épisodes précédents

Borne inférieure de complexité

tout algorithme de tri par comparaisons nécessite $\Omega(n \log n)$ comparaisons dans le pire cas (et en moyenne)

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*

Rappel des épisodes précédents

Borne inférieure de complexité

tout algorithme de tri par comparaisons nécessite $\Omega(n \log n)$ comparaisons dans le pire cas (et en moyenne)

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 plus efficace en moyenne que le tri fusion ?
- ... et qui trie en place ?

Des permutations particulières : les 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 $Supp(\sigma)$)

Des permutations particulières : les 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 $Supp(\sigma)$)

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

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

Des permutations particulières : les 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 $Supp(\sigma)$)

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

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

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

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

Des permutations particulières : les 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

Des permutations particulières : les 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

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

Des permutations particulières : les 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

Action des transpositions sur les permutations

- produit *à gauche* par $(i\ j) \Leftrightarrow$ échange des *valeurs* i et j
- produit *à droite* par $(i\ j) \Leftrightarrow$ échange des (éléments en) *positions* i et j

Produits de transpositions

Action des transpositions sur les permutations

- produit *à gauche* par $(i\ j) \Leftrightarrow$ échange des *valeurs* i et j
- produit *à droite* par $(i\ j) \Leftrightarrow$ échange des (éléments en) *positions* i et j

Lemme

toute permutation peut être décomposée en produit de transpositions

Action des transpositions sur les permutations

- produit *à gauche* par $(i\ j) \Leftrightarrow$ échange des *valeurs* i et j
- produit *à droite* par $(i\ j) \Leftrightarrow$ échange des (éléments en) *positions* i et j

Lemme

toute permutation peut être décomposée en produit de transpositions

c'est ce que calcule tout algorithme de tri par comparaisons/échanges □

Produits de transpositions

Action des transpositions sur les permutations

- produit *à gauche* par $(i\ j)$ $\Leftrightarrow$ échange des *valeurs* i et j
- produit *à droite* par $(i\ j)$ $\Leftrightarrow$ échange des (éléments en) *positions* i et j

Lemme

toute permutation peut être décomposée en produit de transpositions

c'est ce que calcule tout algorithme de tri par comparaisons/échanges □

Par exemple, en exécutant le tri par sélection on obtient :

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 :

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

Produits de transpositions

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 :*

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

Produits de transpositions

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 $n!$

Produits de transpositions

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 $n!$

Dit autrement : tout tableau peut être trié en échangeant

- les éléments en position 1 et b_1, pour un $b_1 \geqslant 1$ bien choisi,
- puis les éléments en position 2 et b_2, pour un $b_2 \geqslant 2$ bien choisi,
- ...

Produits de transpositions

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 $n!$

Dit autrement : tout tableau peut être trié en échangeant

- les éléments en position 1 et b_1, pour un $b_1 \geqslant 1$ bien choisi,
- puis les éléments en position 2 et b_2, pour un $b_2 \geqslant 2$ bien choisi,
- ...

Démonstration.

C'est très exactement ce que fait le tri par sélection (version en place) ! □

Aparté : génération aléatoire de permutations

RandomPermutation(n)

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

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)

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

Aparté : génération aléatoire de permutations

RandomPermutation(n)

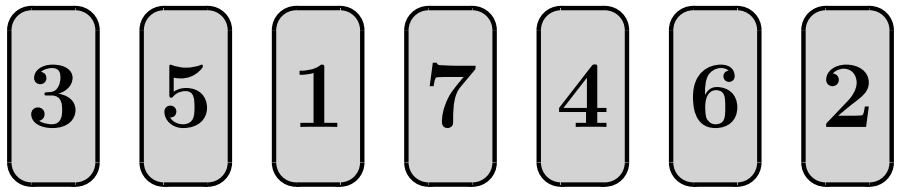
construire une des $n!$ permutations de taille n selon la loi de probabilité uniforme

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

```
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
```

Complexité : $\Theta(n)$ tirages aléatoires d'entiers

Rappel : tri par insertion dans un tableau




```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

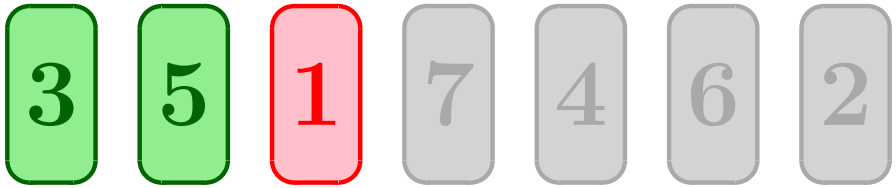
Rappel : tri par insertion dans un tableau


```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau


```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau




```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau


```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

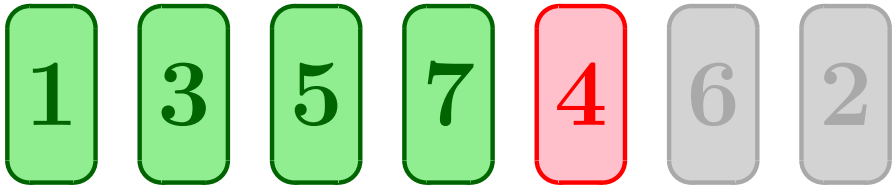
Rappel : tri par insertion dans un tableau


```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau

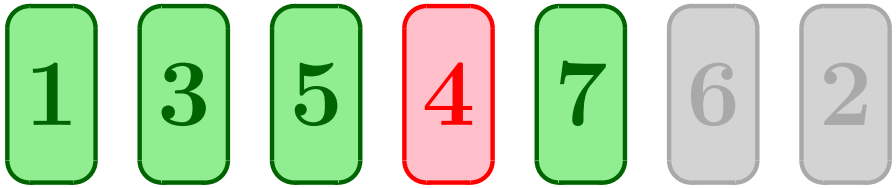

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau



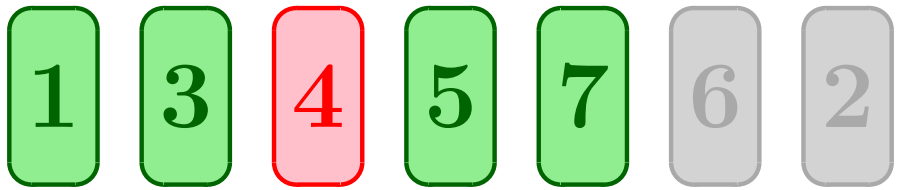

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau



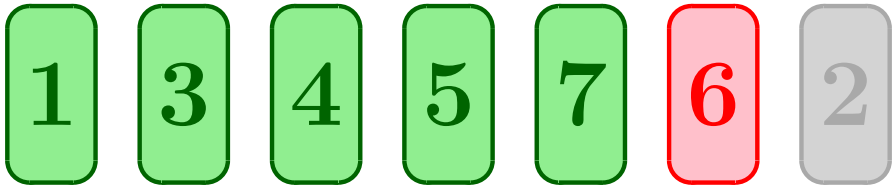

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau



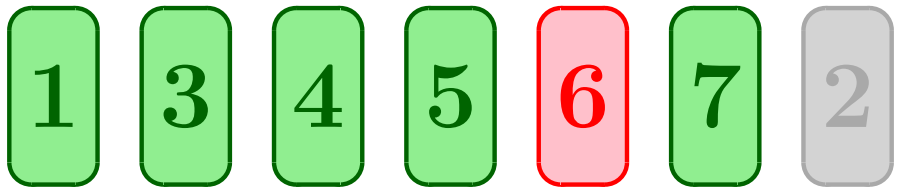

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau



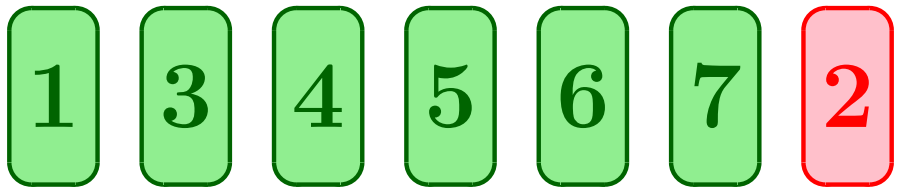

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau



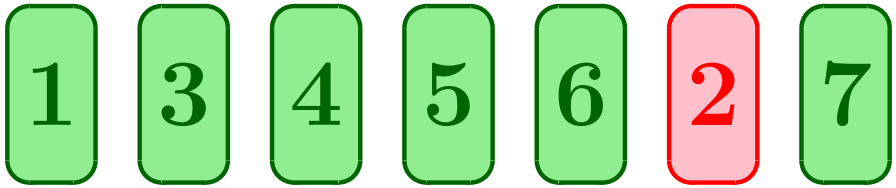

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau




```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau

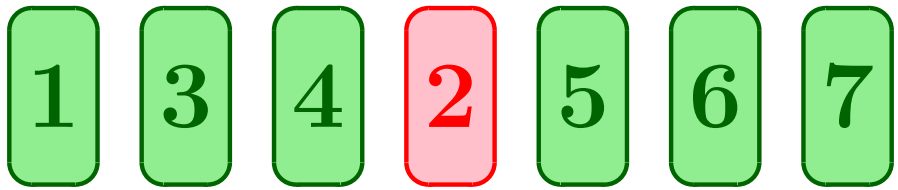



```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau

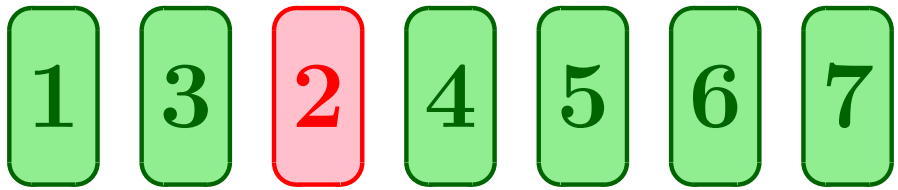

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau



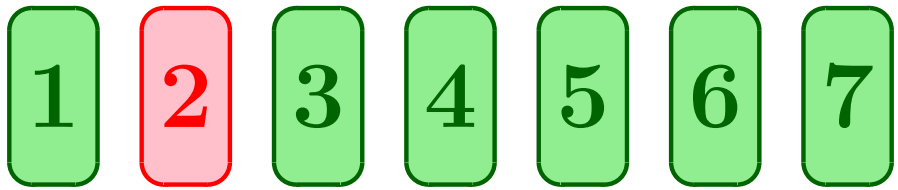

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau



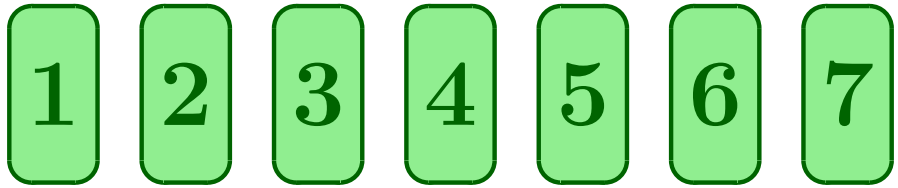

```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau




```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Rappel : tri par insertion dans un tableau




```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Remarque : il est important d'effectuer le parcours de droite à gauche – sinon la complexité serait $\Theta(n^2)$ dans tous les cas

Inversions

inversion de σ : couple (i, j) d'éléments de $[\![1, n]\!]$ tel que

$$i < j \quad \text{et} \quad \sigma^{-1}(j) < \sigma^{-1}(i)$$

(autrement dit : les positions ne respectent pas l'ordre des valeurs)

notations : $\mathcal{I}(\sigma) = \{(i, j) \text{ inversion de } \sigma\}$, $\mathrm{Inv}(\sigma)$ son cardinal

Inversions

inversion de σ : couple (i, j) d'éléments de $[\![1, n]\!]$ tel que

$$i < j \quad \text{et} \quad \sigma^{-1}(j) < \sigma^{-1}(i)$$

(autrement dit : les positions ne respectent pas l'ordre des valeurs)

notations : $\mathcal{I}(\sigma) = \{(i, j) \text{ inversion de } \sigma\}$, $\mathrm{Inv}(\sigma)$ son cardinal

Exemple $\sigma = 2\,4\,6\,1\,5\,3$ a 7 inversions :

- $\boxed{2}\,4\,6\,\boxed{1}\,5\,3$
- $2\,\boxed{4}\,6\,\boxed{1}\,5\,3$
- $2\,4\,\boxed{6}\,\boxed{1}\,5\,3$
- $2\,\boxed{4}\,6\,1\,5\,\boxed{3}$
- $2\,4\,6\,1\,\boxed{5}\,\boxed{3}$
- $2\,4\,\boxed{6}\,1\,5\,\boxed{3}$
- $2\,4\,\boxed{6}\,1\,\boxed{5}\,3$

Inversions

inversion de σ : couple (i,j) d'éléments de $[\![1,n]\!]$ tel que

$$i<j \quad \text{et} \quad \sigma^{-1}(j)<\sigma^{-1}(i)$$

(autrement dit : les positions ne respectent pas l'ordre des valeurs)

notations : $\mathcal{I}(\sigma)=\{(i,j) \text{ inversion de } \sigma\}$, $\mathrm{Inv}(\sigma)$ son cardinal

Proposition

pour tout $\sigma \in \mathfrak{S}_n$, $0 \leqslant \mathrm{Inv}(\sigma) \leqslant \dfrac{n(n-1)}{2}$

inversion de σ : couple (i,j) d'éléments de $[\![1,n]\!]$ tel que

$$i<j \quad \text{et} \quad \sigma^{-1}(j)<\sigma^{-1}(i)$$

(autrement dit : les positions ne respectent pas l'ordre des valeurs)

notations : $\mathcal{I}(\sigma)=\{(i,j) \text{ inversion de } \sigma\}$, $\mathrm{Inv}(\sigma)$ son cardinal

Proposition

pour tout $\sigma \in \mathfrak{S}_n$, $0 \leqslant \mathrm{Inv}(\sigma) \leqslant \dfrac{n(n-1)}{2}$

Proposition

la valeur moyenne de $\mathrm{Inv}(\sigma)$ *pour* $\sigma \in \mathfrak{S}_n$ *est* $\dfrac{n(n-1)}{4}$

Inversions et tri par insertion

échange de deux valeurs à des positions ***contiguës***

$\Longleftrightarrow$

multiplication (à droite) par une transposition de type $(i \;\; i+1)$

$\Longleftrightarrow$

ajout ou suppression d'exactement une ***inversion***

Inversions et tri par insertion

échange de deux valeurs à des positions ***contiguës***

$\Longleftrightarrow$

multiplication (à droite) par une transposition de type $(i \;\; i+1)$

$\Longleftrightarrow$

ajout ou suppression d'exactement une ***inversion***

Proposition

le tri par insertion supprime exactement une inversion à chaque échange

(c'est aussi le cas du tri à bulles, mais le tri par insertion fait beaucoup moins de comparaisons)

Inversions et tri par insertion

échange de deux valeurs à des positions ***contiguës***

$\Longleftrightarrow$

multiplication (à droite) par une transposition de type $(i \;\; i+1)$

$\Longleftrightarrow$

ajout ou suppression d'exactement une ***inversion***

Proposition

le tri par insertion supprime exactement une inversion à chaque échange

(c'est aussi le cas du tri à bulles, mais le tri par insertion fait beaucoup moins de comparaisons)

Théorème

*la complexité **moyenne** du tri par insertion est en* $\Theta(n^2)$

Théorème

la complexité ***moyenne*** *du tri par insertion est en* $\Theta(n^2)$

Inversions et tri par insertion

Théorème

*la complexité **moyenne** du tri par insertion est en* $\Theta(n^2)$

plus généralement, la complexité du tri par insertion d'une permutation de taille n ayant ℓ inversions est en $\Theta(\ell + n)$:
ℓ comparaisons-échanges et $\Theta(n)$ comparaisons supplémentaires

Inversions et tri par insertion

Théorème

*la complexité **moyenne** du tri par insertion est en* $\Theta(n^2)$

plus généralement, la complexité du tri par insertion d'une permutation de taille n ayant ℓ inversions est en $\Theta(\ell + n)$:
ℓ comparaisons-échanges et $\Theta(n)$ comparaisons supplémentaires

le tri par insertion est donc un tri de complexité ***linéaire*** lorsqu'il est appliqué sur des (familles de) permutations ayant un ***nombre sous-linéaire d'inversions***

Inversions et tri par insertion

Théorème

*la complexité **moyenne** du tri par insertion est en* $\Theta(n^2)$

plus généralement, la complexité du tri par insertion d'une permutation de taille n ayant ℓ inversions est en $\Theta(\ell + n)$:
ℓ comparaisons-échanges et $\Theta(n)$ comparaisons supplémentaires

le tri par insertion est donc un tri de complexité ***linéaire*** lorsqu'il est appliqué sur des (familles de) permutations ayant un ***nombre sous-linéaire d'inversions***

l'hypothèse d'un nombre d'inversions au plus linéaire est en fait « ***assez probable*** » en pratique : c'est le cas par exemple des tableaux qui ont un jour été triés et n'ont depuis subi qu'un nombre limité de modifications

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* et *en moyenne*,
- $\Theta(n)$ comparaisons *au mieux* (CNS : $O(n)$ inversions),
- trie *en place*

Tri rapide (*Quicksort*)

Observation :

- les qualités du tri fusion proviennent de la stratégie « *diviser-pour-régner* »
- ses défauts proviennent en partie du fait qu'il s'agit de récursivité *non terminale* : la fusion est réalisée *après* les appels récursifs

Tri rapide (*Quicksort*)

Observation :

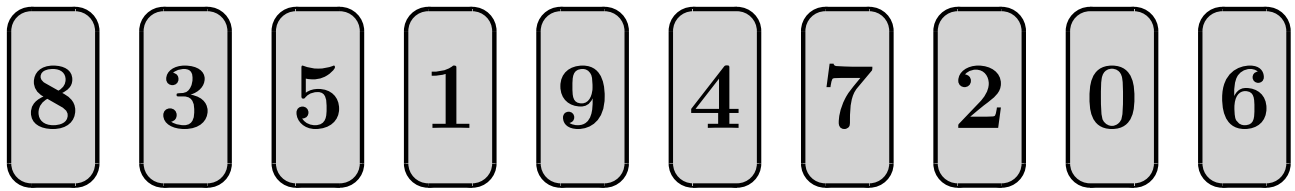
- les qualités du tri fusion proviennent de la stratégie « ***diviser-pour-régner*** »
- ses défauts proviennent en partie du fait qu'il s'agit de récursivité ***non terminale*** : la fusion est réalisée ***après*** les appels récursifs

Idée : faire un prétraitement ***avant*** les appels récursifs pour éviter d'en avoir besoin ***après***

Tri rapide (*Quicksort*)

Idée : faire un prétraitement ***avant*** les appels récursifs pour éviter d'en avoir besoin ***après***

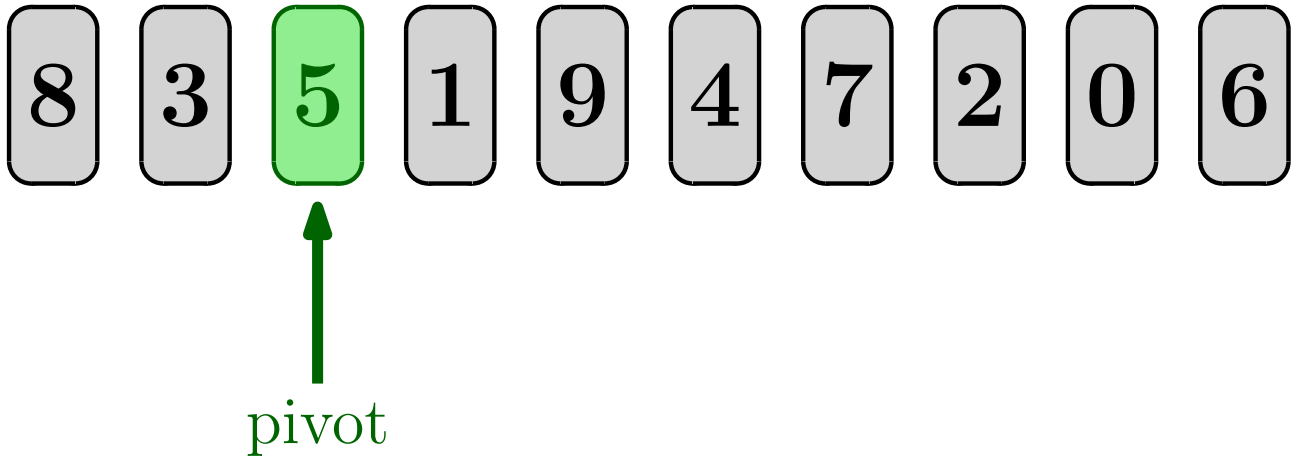
Exemple :



Tri rapide (*Quicksort*)

Idée : faire un prétraitement ***avant*** les appels récursifs pour éviter d'en avoir besoin ***après***

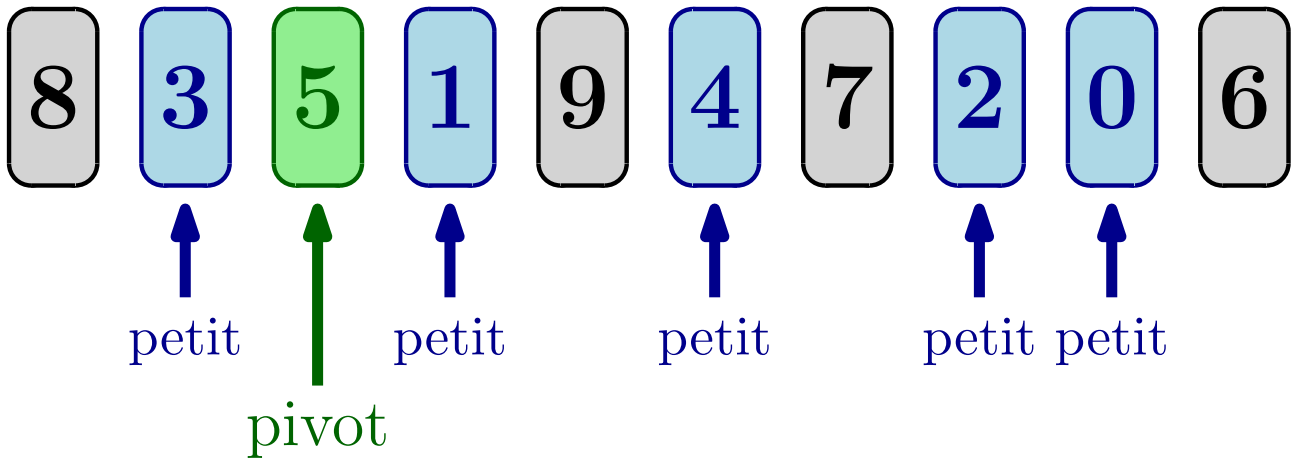
Exemple :



Tri rapide (*Quicksort*)

Idée : faire un prétraitement ***avant*** les appels récursifs pour éviter d'en avoir besoin ***après***

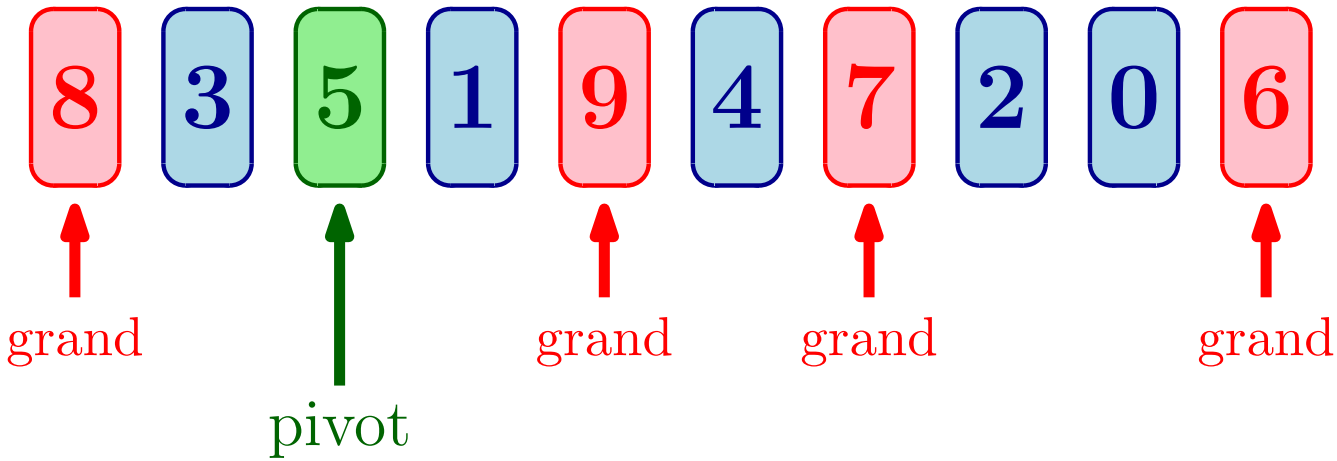
Exemple :



Tri rapide (*Quicksort*)

Idée : faire un prétraitement ***avant*** les appels récursifs pour éviter d'en avoir besoin ***après***

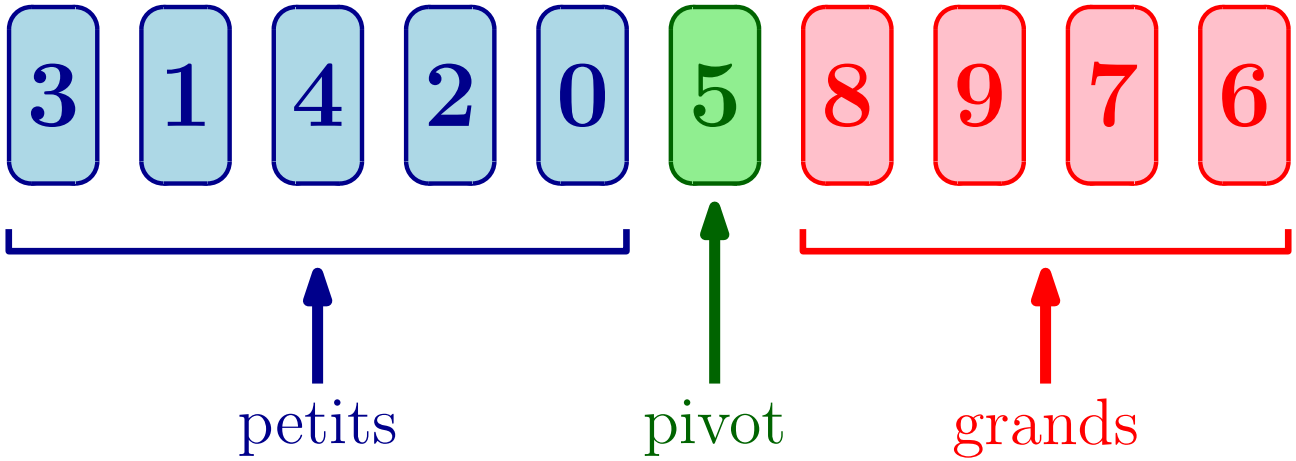
Exemple :



Tri rapide (*Quicksort*)

Idée : faire un prétraitement ***avant*** les appels récursifs pour éviter d'en avoir besoin ***après***

Exemple :



Tri rapide (*Quicksort*), version 1

```
def partition(T) :  # les éléments sont supposés distincts
  pivot = T[0]
  gauche = [ elt for elt in T if elt < pivot ]
  droite = [ elt for elt in T if elt > pivot ]
  return pivot, gauche, droite
```

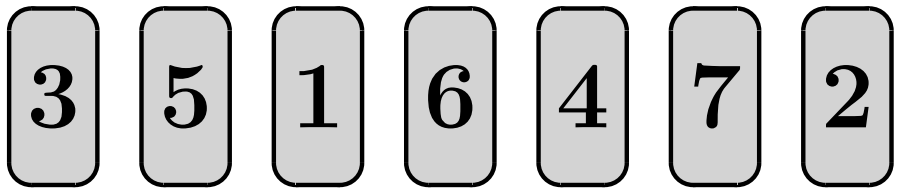
Tri rapide (*Quicksort*), version 1

```
def partition(T) :  # les éléments sont supposés distincts
  pivot = T[0]
  gauche = [ elt for elt in T if elt < pivot ]
  droite = [ elt for elt in T if elt > pivot ]
  return pivot, gauche, droite

def tri_rapide(T) :
  if len(T) < 2 : return T
  pivot, gauche, droite = partition(T)
  return tri_rapide(gauche) + [pivot] + tri_rapide(droite)
```

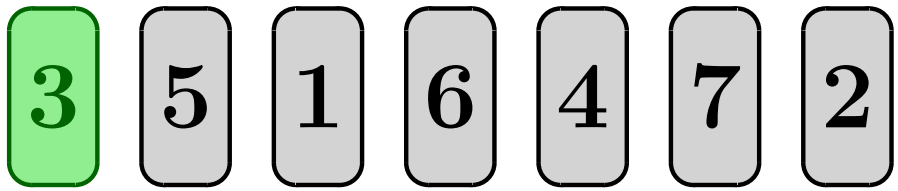
Tri rapide (*Quicksort*), version 1

Exemple :



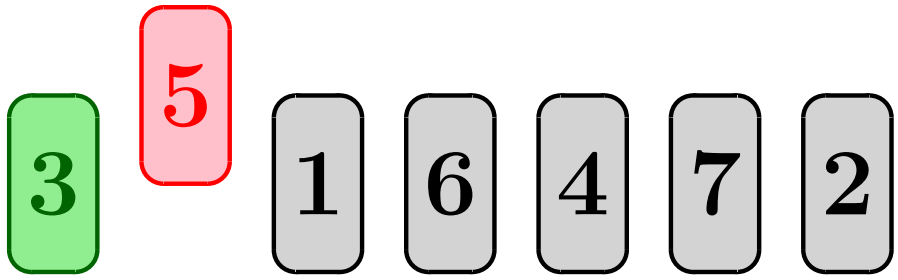
Tri rapide (*Quicksort*), version 1

Exemple :



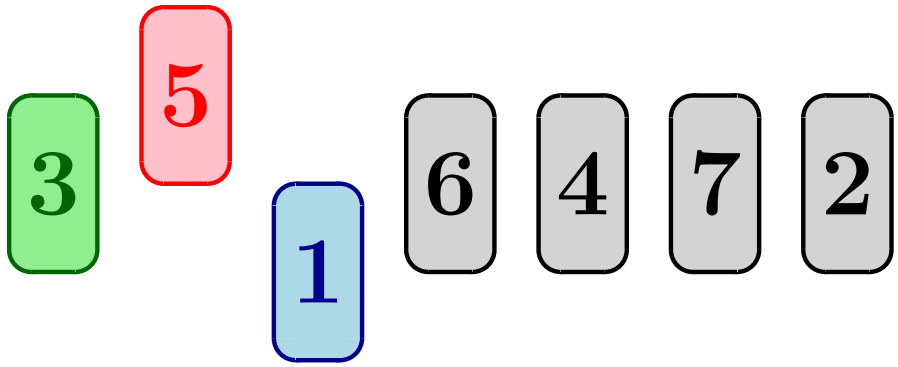
Tri rapide (*Quicksort*), version 1

Exemple :



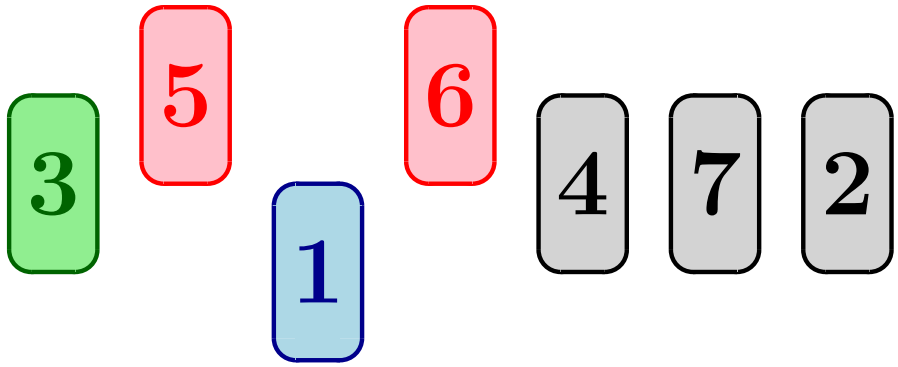
Tri rapide (*Quicksort*), version 1

Exemple :



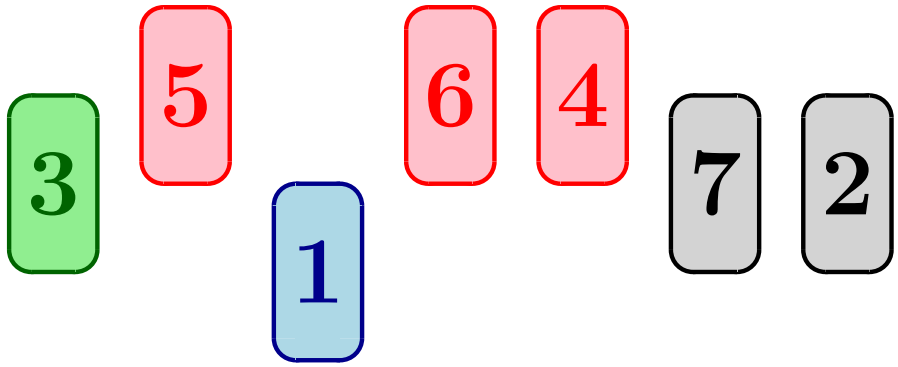
Tri rapide (*Quicksort*), version 1

Exemple :



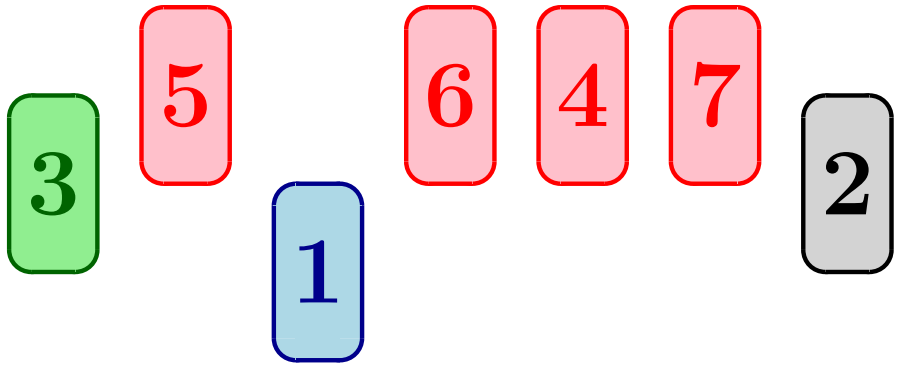
Tri rapide (*Quicksort*), version 1

Exemple :



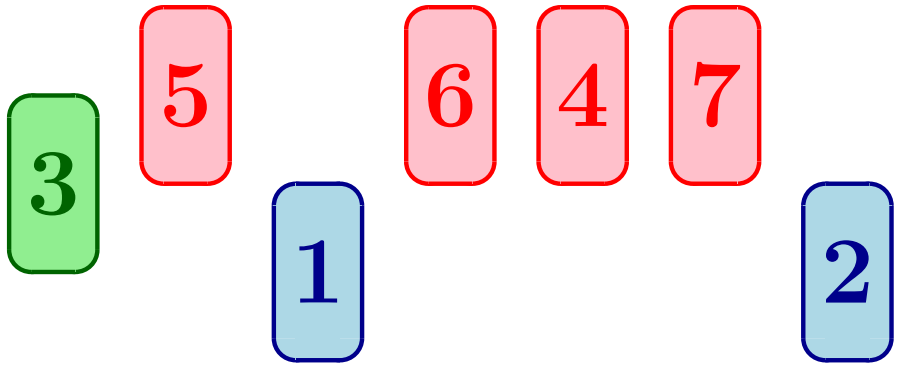
Tri rapide (*Quicksort*), version 1

Exemple :



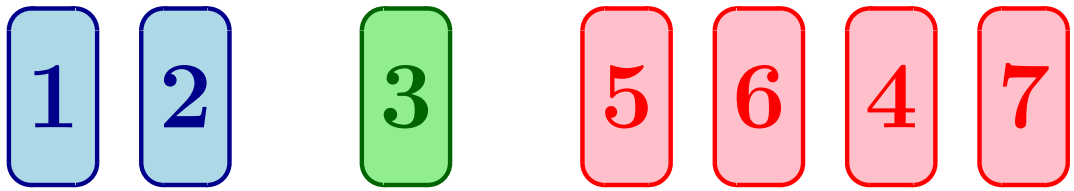
Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

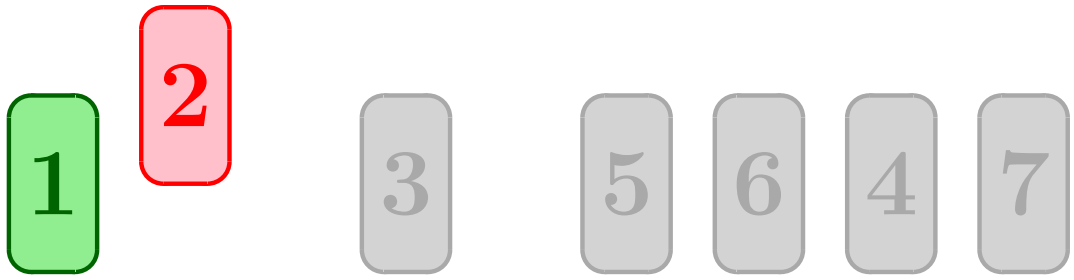
Exemple :

Tri rapide (*Quicksort*), version 1

Exemple :

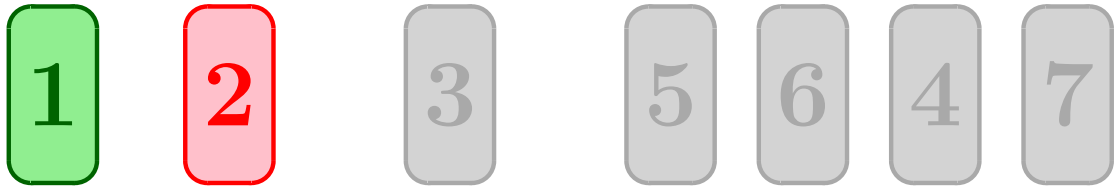
Tri rapide (*Quicksort*), version 1

Exemple :



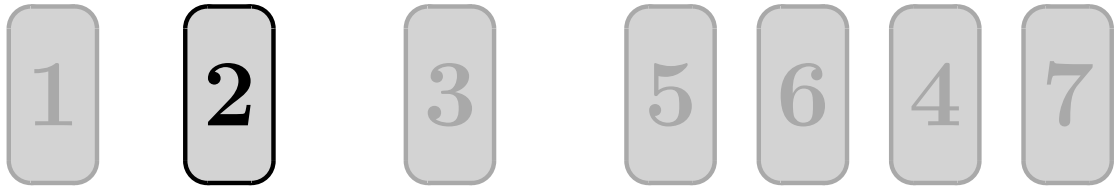
Tri rapide (*Quicksort*), version 1

Exemple :



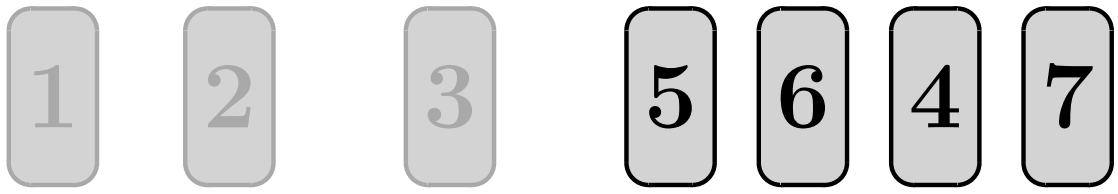
Tri rapide (*Quicksort*), version 1

Exemple :



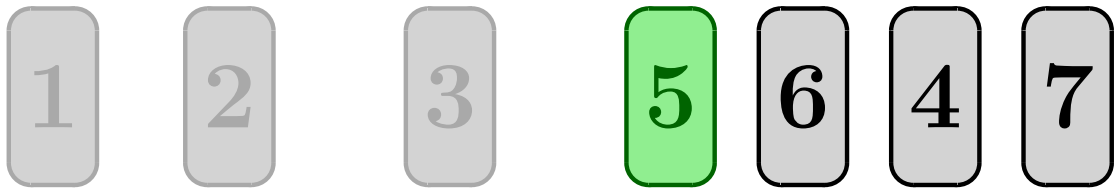
Tri rapide (*Quicksort*), version 1

Exemple :



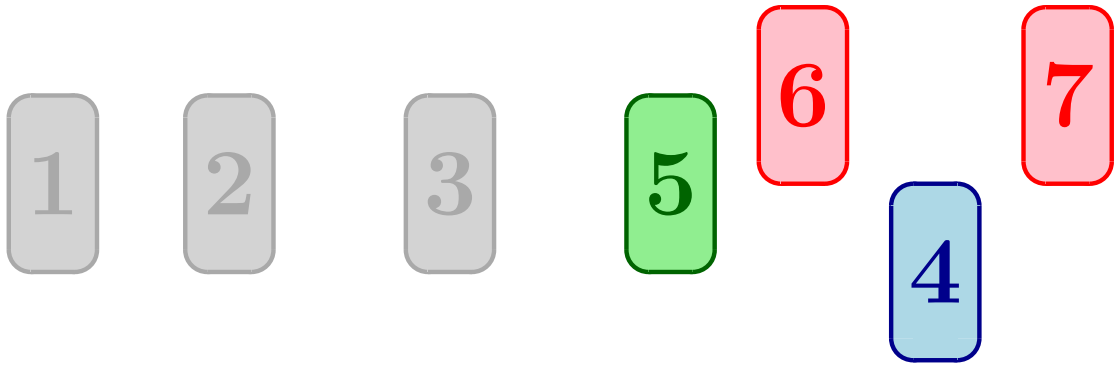
Tri rapide (*Quicksort*), version 1

Exemple :



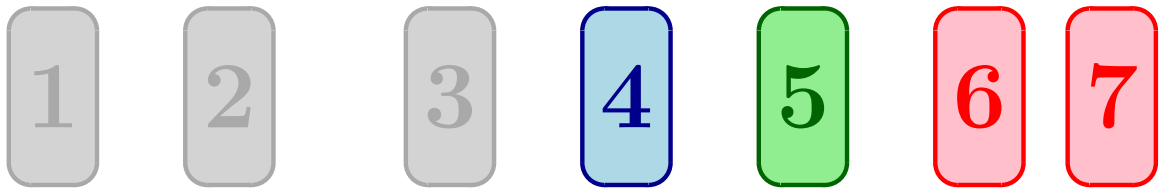
Tri rapide (*Quicksort*), version 1

Exemple :



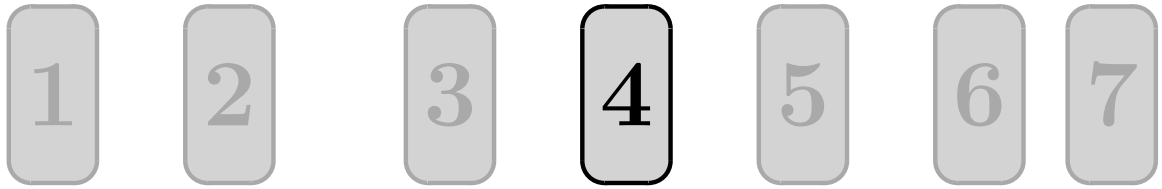
Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :

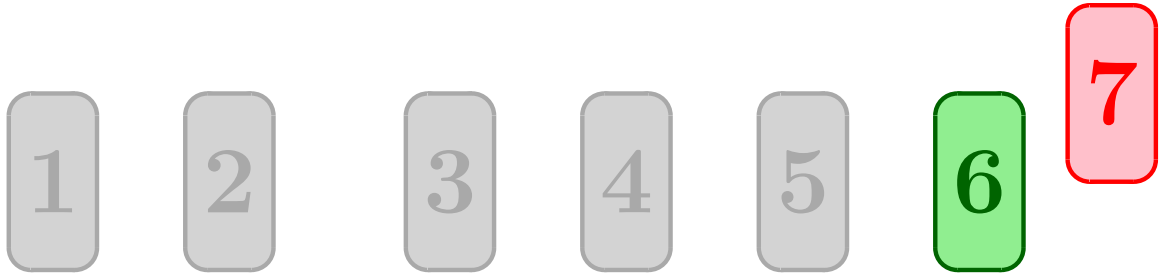


Tri rapide (*Quicksort*), version 1

Exemple :

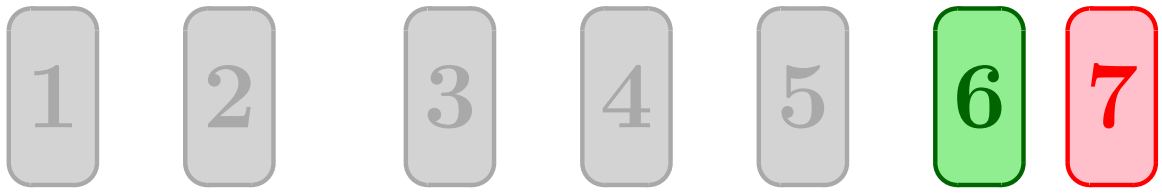
Tri rapide (*Quicksort*), version 1

Exemple :



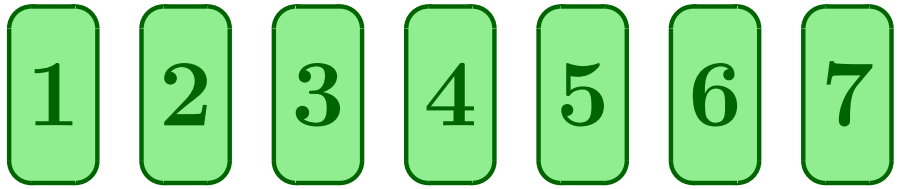
Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

```
def partition(T) :  # les éléments sont supposés distincts
  pivot = T[0]
  gauche = [ elt for elt in T if elt < pivot ]
  droite = [ elt for elt in T if elt > pivot ]
  return pivot, gauche, droite
```

Tri rapide (*Quicksort*), version 1

```
def partition(T) :  # les éléments sont supposés distincts
  pivot = T[0]
  gauche = [ elt for elt in T if elt < pivot ]
  droite = [ elt for elt in T if elt > pivot ]
  return pivot, gauche, droite
```

Complexité de `partition` : $\Theta(n)$ comparaisons

Tri rapide (*Quicksort*), version 1

```
def partition(T) :  # les éléments sont supposés distincts
  pivot = T[0]
  gauche = [ elt for elt in T if elt < pivot ]
  droite = [ elt for elt in T if elt > pivot ]
  return pivot, gauche, droite
```

Complexité de `partition` : $\Theta(n)$ comparaisons

```
def tri_rapide(T) :
  if len(T) < 2 : return T
  pivot, gauche, droite = partition(T)
  return tri_rapide(gauche) + [pivot] + tri_rapide(droite)
```

Tri rapide (*Quicksort*), version 1

```
def partition(T) :  # les éléments sont supposés distincts
  pivot = T[0]
  gauche = [ elt for elt in T if elt < pivot ]
  droite = [ elt for elt in T if elt > pivot ]
  return pivot, gauche, droite
```

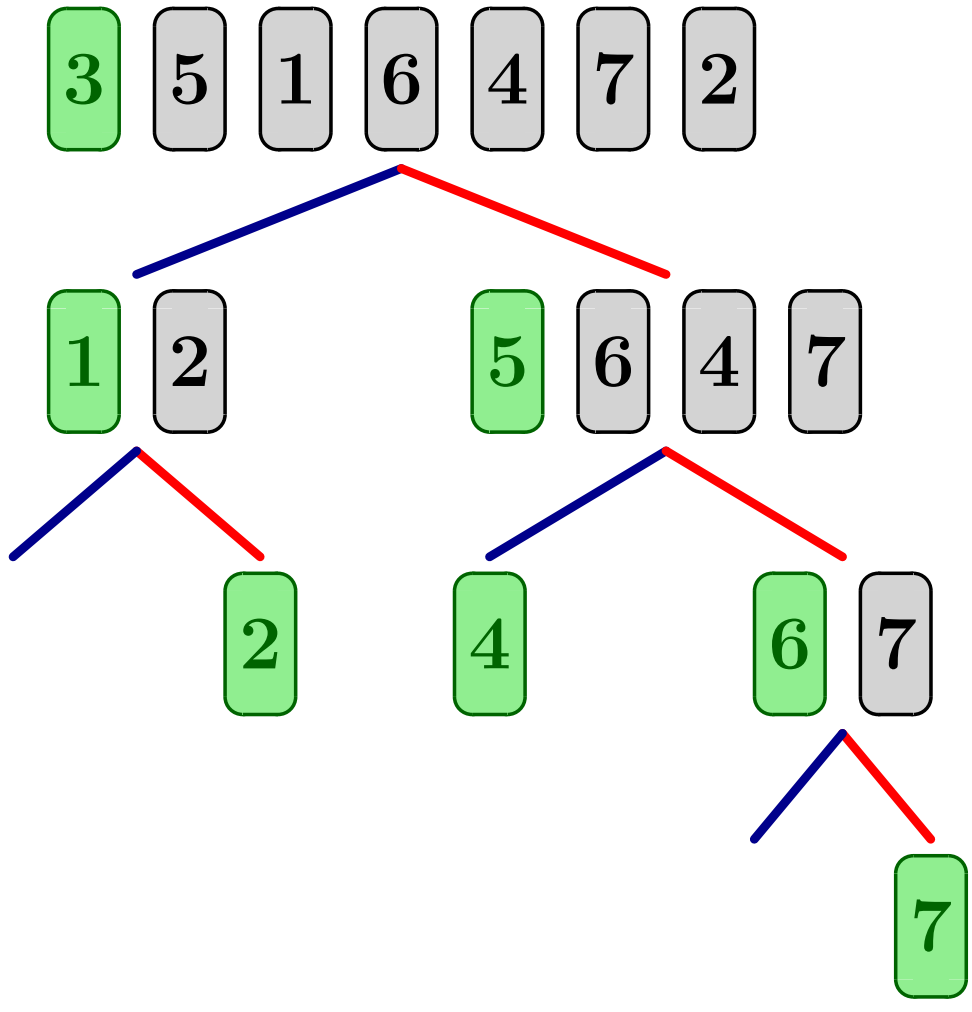
Complexité de `partition` : $\Theta(n)$ comparaisons

```
def tri_rapide(T) :
  if len(T) < 2 : return T
  pivot, gauche, droite = partition(T)
  return tri_rapide(gauche) + [pivot] + tri_rapide(droite)
```

Complexité de `tri_rapide` (au pire) : $\Theta(n^2)$ comparaisons

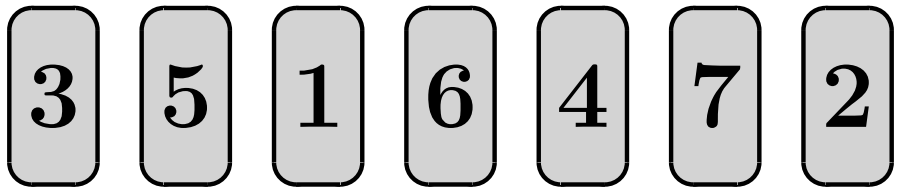
Tri rapide (*Quicksort*), version 1

Exemple :



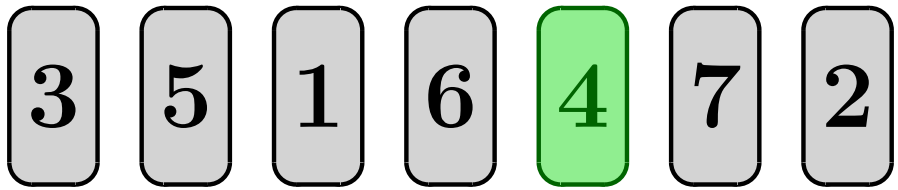
Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :



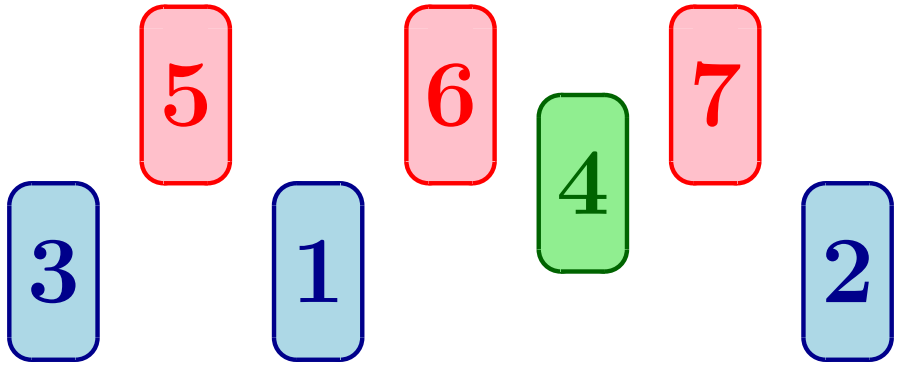
Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :



Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :



Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :

Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :

Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :

Tri rapide (*Quicksort*), version 1

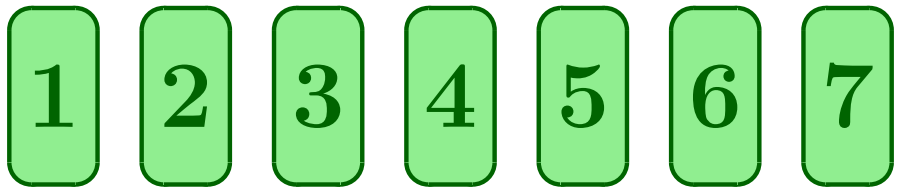
Exemple en choisissant miraculeusement toujours le pivot optimal :

Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :

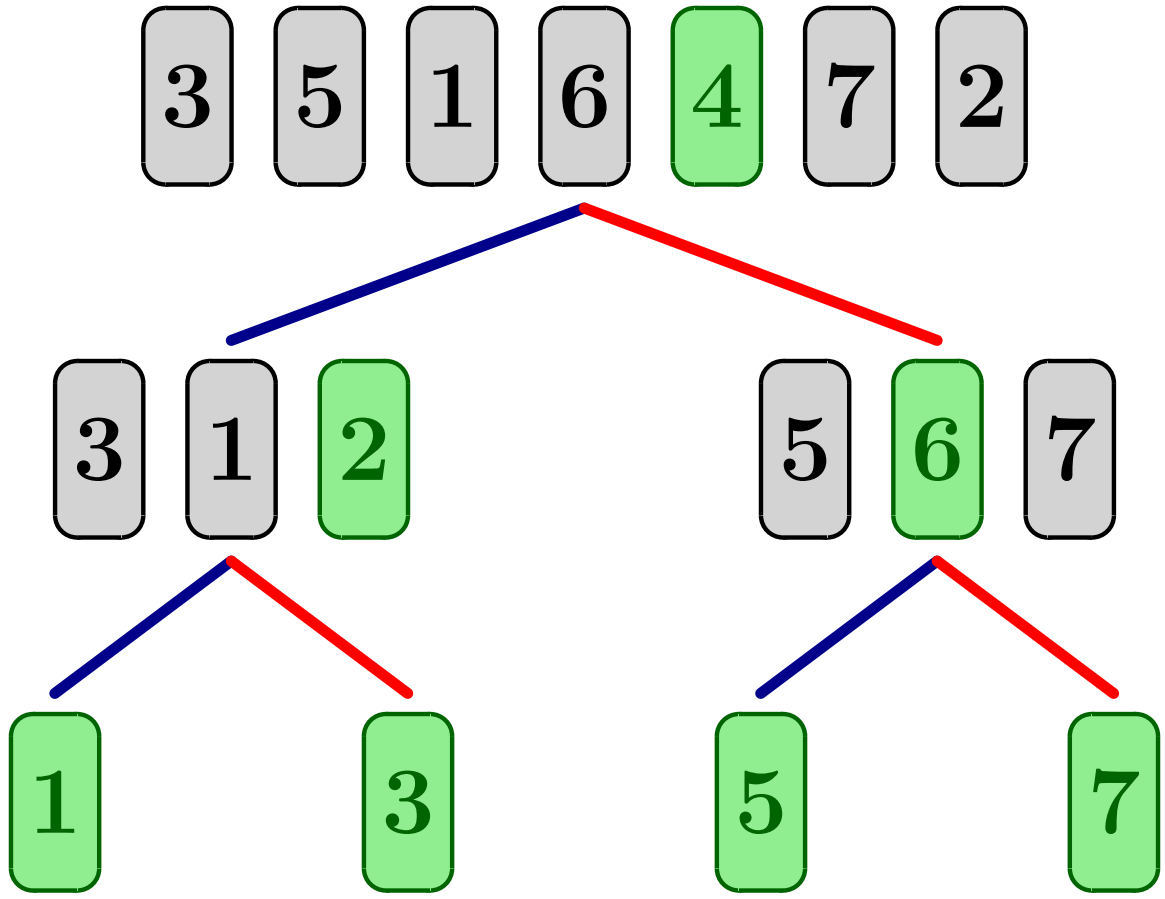
Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :



Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :



Tri rapide (*Quicksort*), version 1

Complexité de `tri_rapide` au pire : $\Theta(n^2)$ comparaisons

Complexité de `tri_rapide` dans le meilleur des cas :
$\Theta(n \log n)$ comparaisons

Complexité de `tri_rapide` en moyenne :
(*admis pour le moment*) $\Theta(n \log n)$ comparaisons

Tri rapide (*Quicksort*), version 1

Inconvénients

- `partition` fait deux parcours, là où un seul suffit manifestement

  (ce point est très facile à corriger)

- ne trie ***pas en place*** – multiples recopies de (portions de) tableaux, même les éléments « bien placés » sont déplacés
- les ***mauvais cas*** sont des cas « ***assez probables*** » : tableaux triés ou presque, à l'endroit ou à l'envers