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 2020-2021

- CM sur BBB, maintenu à 13h30
- TD ***sur site pour tous les groupes*** (INFO1 déplacé au vendredi 15h45, INFO2 au jeudi 15h15)
- TP en autonomie, avec une permanence des enseignants sur discord ***le jeudi et le vendredi, de 8h30 à 13h***

En particulier, long TP sur les tris cette semaine et la semaine prochaine (à rendre vendredi 19 mars)

Contrôle n° 1 : sur moodle

- mardi 16 mars, 18h-19h, pour les groupes INFO 1-2-3-4
- mercredi 17 mars, 9h-10h pour les groupes INFO 5-6 et MI 1-2

Programme du contrôle : cours 1 à 4

Borne inférieure de complexité

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

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 $\in \Theta(n)$

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***

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 ?

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

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

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

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

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

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

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

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

= 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

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

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

Action des transpositions sur les permutations

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

Lemme

toute permutation peut être décomposée en produit de transpositions

c'est ce que calcule n'importe quel algorithme de tri par comparaisons/échanges

□

Action des transpositions sur les permutations

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

Lemme

toute permutation peut être décomposée en produit de transpositions

c'est ce que calcule n'importe quel algorithme de tri par comparaisons/échanges $\square$

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

Lemme

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

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

Lemme

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

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

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 $n!$

Lemme

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

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

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

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

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

Ou encore : tout tableau peut être trié en échangeant

- l'élément en position 1 avec l'élément en position b_1,
- puis l'élément en position 2 avec l'élément en position b_2,
- ...

Lemme

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

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

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

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

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

Ou encore : tout tableau peut être trié en échangeant

- l'élément en position 1 avec l'élément en position b_1,
- puis l'élément en position 2 avec l'élément en position b_2,
- ...

Démonstration.

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

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

Tri par insertion dans un tableau

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

```python
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
```

Tri par insertion dans un tableau

3	5	1	7	4	6	2

```
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
```

Tri par insertion dans un tableau

3 5 1 7 4 6 2

```
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
```

Tri par insertion dans un tableau

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

```
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
```

Tri par insertion dans un tableau

3 1 5 7 4 6 2

```
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
```

Tri par insertion dans un tableau

1	3	5	7	4	6	2

```
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
```

Tri par insertion dans un tableau

1 3 5 7 4 6 2

```
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
```

Tri par insertion dans un tableau

1	3	5	7	4	6	2

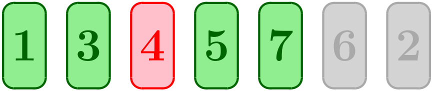
```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Tri par insertion dans un tableau

1	3	5	4	7	6	2

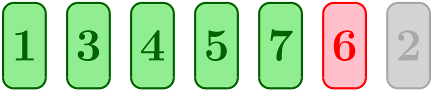
```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

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
```

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
```

Tri par insertion dans un tableau

1	3	4	5	6	7	2

```
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
```

Tri par insertion dans un tableau

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

```python
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
```

Tri par insertion dans un tableau

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

```
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
```

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
```

Tri par insertion dans un tableau

1	3	4	2	5	6	7

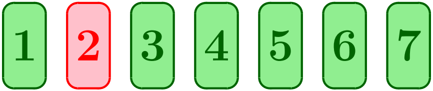
```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Tri par insertion dans un tableau

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

```
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
```

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
```

Tri par insertion dans un tableau

1	2	3	4	5	6	7

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

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

(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

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

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

(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 :

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

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

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

(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

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

(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* $\dfrac{n(n-1)}{4}$

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'une ***inversion***

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'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)

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

$\iff$

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

$\iff$

ajout ou suppression d'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* $\Theta(n^2)$

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

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 permutations ayant un ***nombre sous-linéaire d'inversions***

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 permutations ayant un ***nombre sous-linéaire d'inversions***

l'hypothèse d'un nombre d'inversions borné 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*

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 récusivité ***non terminale*** : la fusion est réalisée ***après*** les appels récursifs

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 récusivité *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*

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

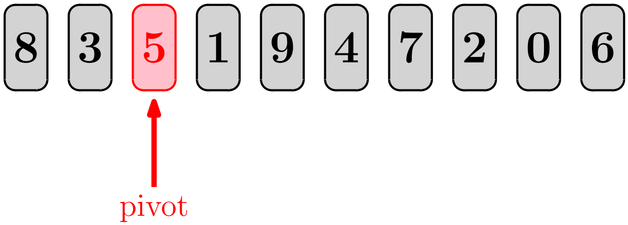
Exemple :

8	**3**	**5**	**1**	**9**	**4**	**7**	**2**	**0**	**6**

Tri rapide (*Quicksort*)

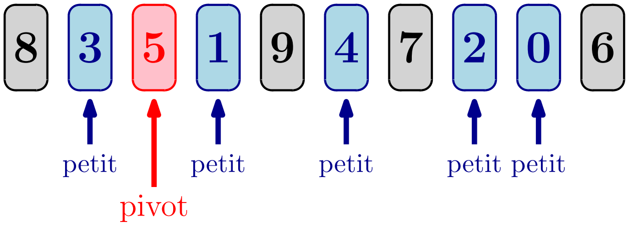
Idée : faire un prétraitement *avant* les appels récursifs pour éviter d'en avoir besoin *après*

Exemple :



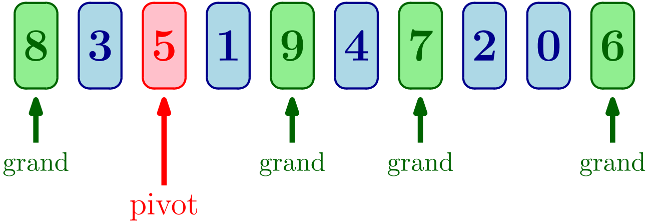
Idée : faire un prétraitement *avant* les appels récursifs pour éviter d'en avoir besoin *après*

Exemple :



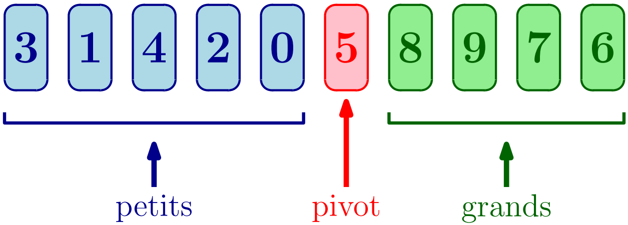
Idée : faire un prétraitement *avant* les appels récursifs pour éviter d'en avoir besoin *après*

Exemple :



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 :

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

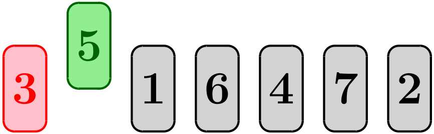
Tri rapide (*Quicksort*), version 1

Exemple :

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

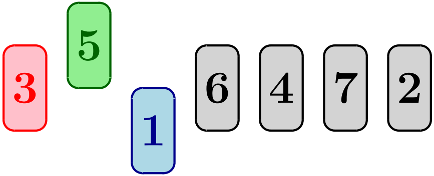
Tri rapide (*Quicksort*), version 1

Exemple :

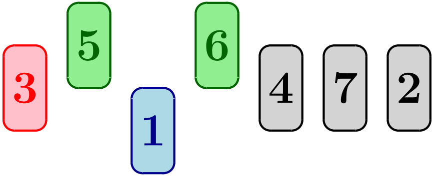


Tri rapide (*Quicksort*), version 1

Exemple :

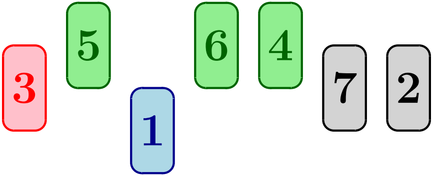


Exemple :



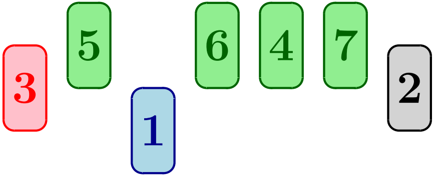
Tri rapide (*Quicksort*), version 1

Exemple :



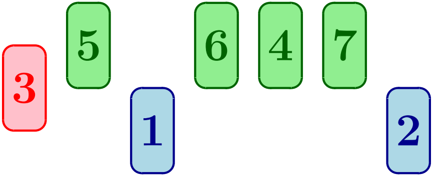
TRI RAPIDE (*Quicksort*), VERSION 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :



Exemple :

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

Tri rapide (*Quicksort*), version 1

Exemple :

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

Exemple :

Tri rapide (*Quicksort*), version 1

Exemple :

Tri rapide (*Quicksort*), version 1

Exemple :

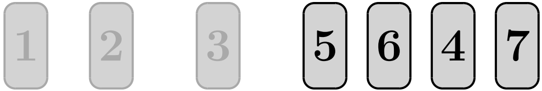
1 **2** **3** **5** **6** **4** **7**

Tri rapide (*Quicksort*), version 1

Exemple :

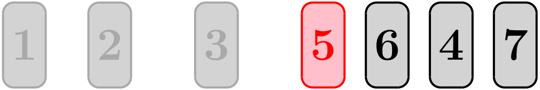
1 **2** 3 5 6 4 7

Exemple :



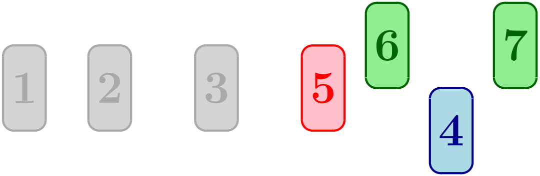
Tri rapide (*Quicksort*), version 1

Exemple :



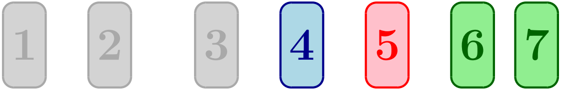
Tri rapide (*Quicksort*), version 1

Exemple :



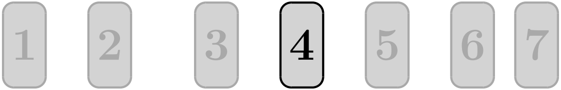
Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :



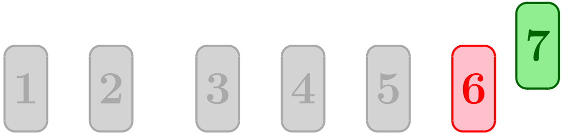
Tri rapide (*Quicksort*), version 1

Exemple :

1 2 3 4 5 6 7

Tri rapide (*Quicksort*), version 1

Exemple :



Exemple :

Exemple :

1	2	3	4	5	6	7

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

```python
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

Exemple :