Module EA4 – Éléments d’Algorithmique II

Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique
Année universitaire 2014-2015

Rappel des épisodes précédents

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

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 ?

Rappel des épisodes précédents

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 $\Theta(n)$?

Rappel des épisodes précédents

Tri par fusion

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

Tri par insertion

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

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

Rappel des épisodes précédents

Tri par fusion

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

Tri par insertion

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

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

Inversions

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

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

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

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

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 :

- **2** 4 6 **1** 5 3
- 2 **4** 6 **1** 5 3
- 2 4 **6** **1** 5 3
- 2 **4** 6 1 5 **3**
- 2 4 6 1 **5** **3**
- 2 4 **6** 1 5 **3**
- 2 4 **6** 1 **5** 3

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

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

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

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

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

Inversions et tri par insertion

échange de deux valeurs contiguës $\Longleftrightarrow$ multiplication (à droite) par une transposition de type $(i \ \ i+1)$

effet de la multiplication (à droite) par une transposition $(i \ \ i+1)$: ajout ou suppression d'une inversion échange de deux valeurs contiguës $\Longleftrightarrow$ multiplication (à droite) par une transposition de type $(i \;\; i+1)$

effet de la multiplication (à droite) par une transposition $(i \;\; i+1)$: ajout ou suppression d'une inversion

Proposition

le tri par insertion supprime exactement une inversion à chaque échange

Inversions et tri par insertion

échange de deux valeurs contiguës $\Longleftrightarrow$ multiplication (à droite) par une transposition de type $(i \;\; i+1)$

effet de la multiplication (à droite) par une transposition $(i \;\; i+1)$: ajout ou suppression d'une inversion

Proposition

le tri par insertion supprime exactement une inversion à chaque échange

Théorème

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

Tri rapide (*Quicksort*)

Exemple :

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

Tri rapide (*Quicksort*)

Exemple :

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

Tri rapide (*Quicksort*)

Exemple :

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

Tri rapide (*Quicksort*)

Exemple :

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

Tri rapide (*Quicksort*)

Exemple :

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

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)
```

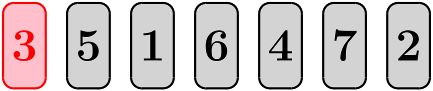
Tri rapide (*Quicksort*), version 1

Exemple :

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

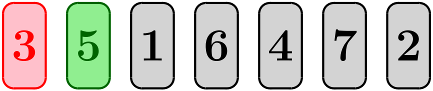
Tri rapide (*Quicksort*), version 1

Exemple :



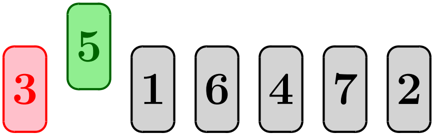
Tri rapide (*Quicksort*), version 1

Exemple :



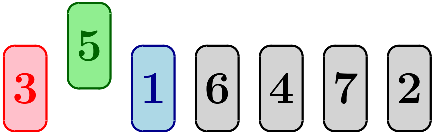
Tri rapide (*Quicksort*), version 1

Exemple :



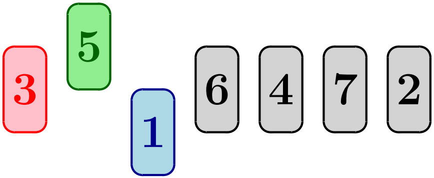
Tri rapide (*Quicksort*), version 1

Exemple :



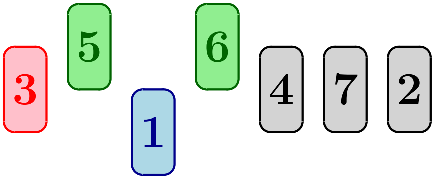
Tri rapide (*Quicksort*), version 1

Exemple :



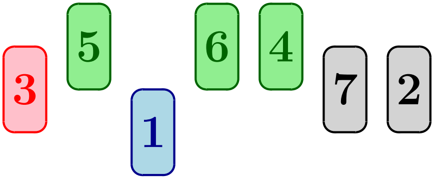
Tri rapide (*Quicksort*), version 1

Exemple :



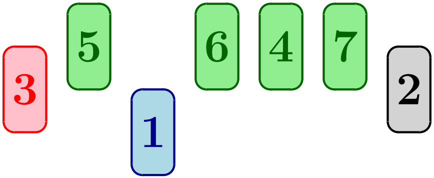
Tri rapide (*Quicksort*), version 1

Exemple :



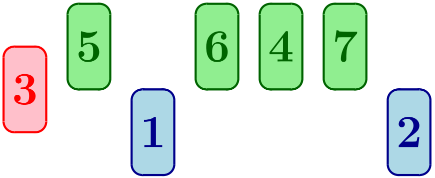
Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :

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

Tri rapide (*Quicksort*), version 1

Exemple :

1 2 3 5 6 4 7

Tri rapide (*Quicksort*), version 1

Exemple :

Tri rapide (*Quicksort*), version 1

Exemple :

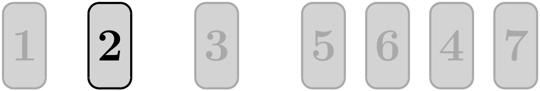
Tri rapide (*Quicksort*), version 1

Exemple :

1 2 3 5 6 4 7

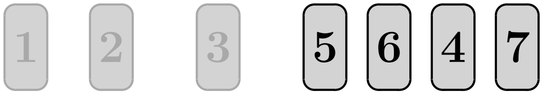
Tri rapide (*Quicksort*), version 1

Exemple :



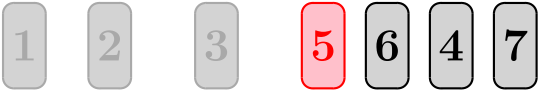
Tri rapide (*Quicksort*), version 1

Exemple :



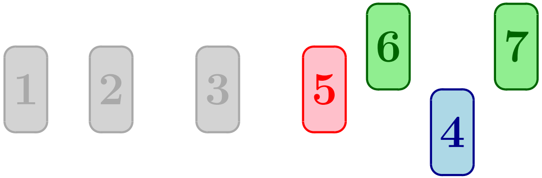
Tri rapide (*Quicksort*), version 1

Exemple :



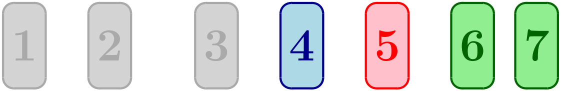
Tri rapide (*Quicksort*), version 1

Exemple :



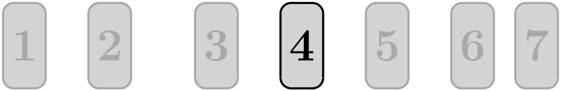
Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :



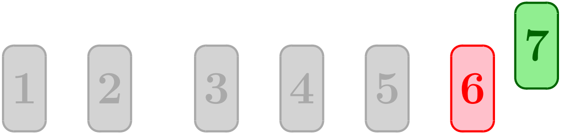
Tri rapide (*Quicksort*), version 1

Exemple :

1 2 3 4 5 6 7

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 :

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

```
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

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

Tri rapide (*Quicksort*), version 1

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

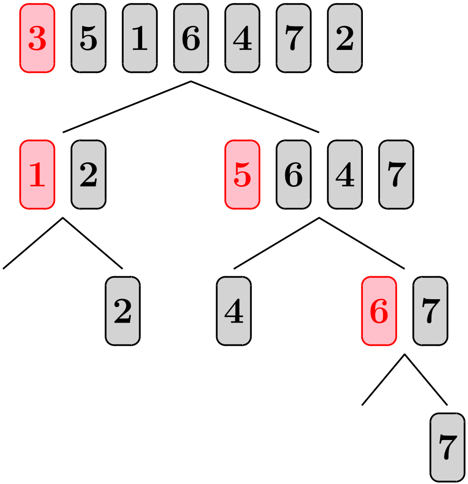
Complexité de `partition` : $\Theta(n)$ comparaisons

```
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 :

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

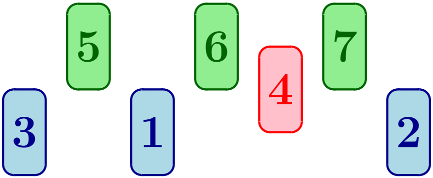
Tri rapide (*Quicksort*), version 1

Exemple :

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

Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :

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**

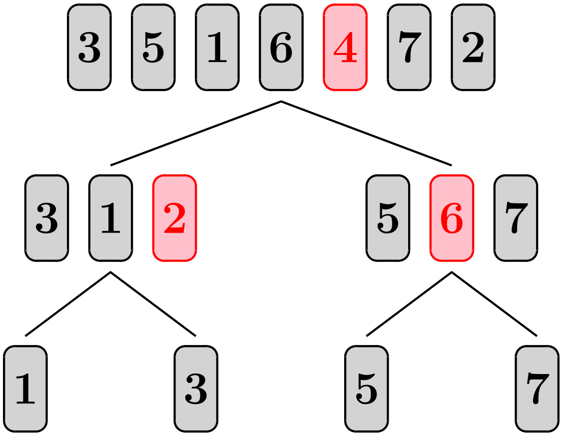
Tri rapide (*Quicksort*), version 1

Exemple :

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

Tri rapide (*Quicksort*), version 1

Exemple :



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*) :
$\Theta(n \log n)$ comparaisons

Tri rapide (*Quicksort*), version 1

Inconvénients

- `partition` fait deux parcours, là où un seul suffit manifestement
- ne trie pas en place – 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

Tri rapide (*Quicksort*), version 2

```
def tri_rapide(T, debut, fin) : # trie T[debut:fin]
  if fin - debut < 2 : return
  indice_pivot = partition(T, debut, fin)
  tri_rapide(T, debut, indice_pivot)
  tri_rapide(T, indice_pivot + 1, fin)
```

Tri rapide (*Quicksort*), version 2

```
def tri_rapide(T, debut, fin) : # trie T[debut:fin]
  if fin - debut < 2 : return
  indice_pivot = partition(T, debut, fin)
  tri_rapide(T, debut, indice_pivot)
  tri_rapide(T, indice_pivot + 1, fin)

def partition(T, debut, fin) :
  pivot, gauche, droite = T[debut], debut + 1, fin - 1
  while gauche < droite :
    while T[gauche] < pivot : gauche += 1
    while T[droite] > pivot : droite -= 1
    if gauche < droite :
      T[gauche], T[droite] = T[droite], T[gauche]
      gauche, droite = gauche + 1, droite - 1
  T[debut], T[droite] = T[droite], pivot
  return droite
```

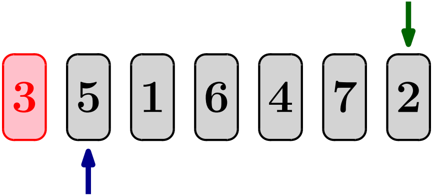
Tri rapide (*Quicksort*), version 2

Exemple :

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

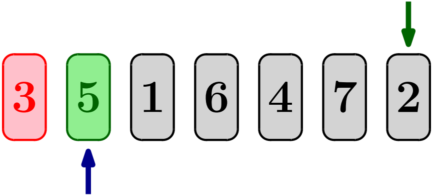
Tri rapide (*Quicksort*), version 2

Exemple :



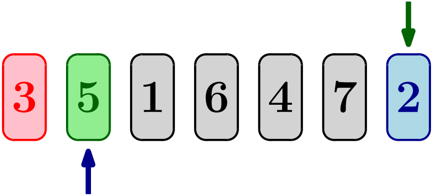
Tri rapide (*Quicksort*), version 2

Exemple :



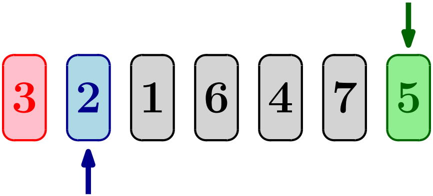
Tri rapide (*Quicksort*), version 2

Exemple :



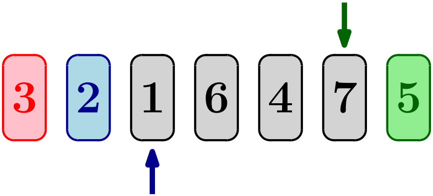
Tri rapide (*Quicksort*), version 2

Exemple :



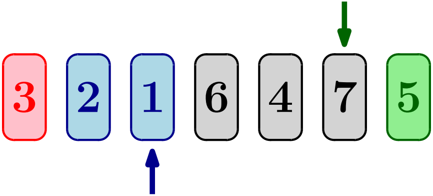
TRI RAPIDE (*Quicksort*), VERSION 2

Exemple :



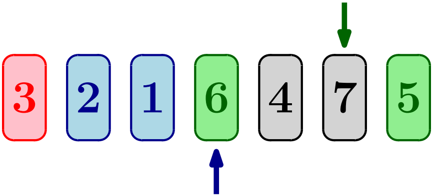
Tri rapide (*Quicksort*), version 2

Exemple :



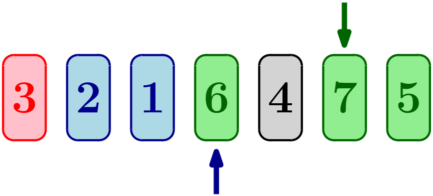
TRI RAPIDE (*Quicksort*), VERSION 2

Exemple :



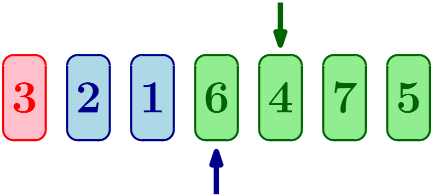
Tri rapide (*Quicksort*), version 2

Exemple :



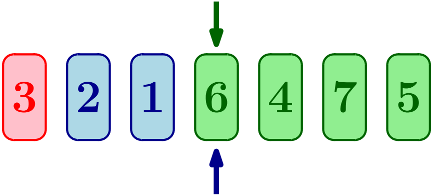
Tri rapide (*Quicksort*), version 2

Exemple :



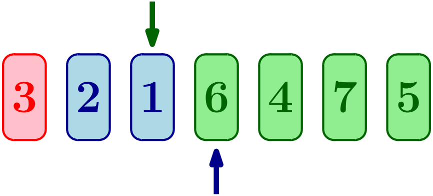
Tri rapide (*Quicksort*), version 2

Exemple :



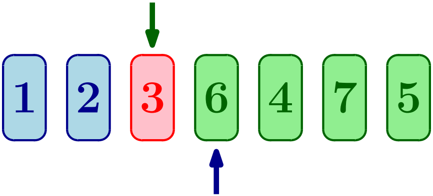
Tri rapide (*Quicksort*), version 2

Exemple :



Tri rapide (*Quicksort*), version 2

Exemple :

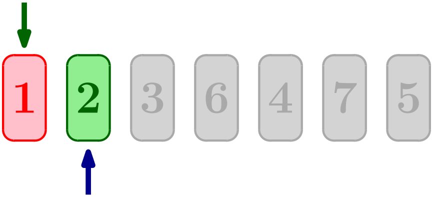


Tri rapide (*Quicksort*), version 2

Exemple :

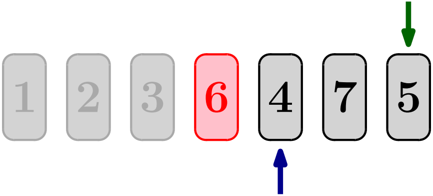
Tri rapide (*Quicksort*), version 2

Exemple :



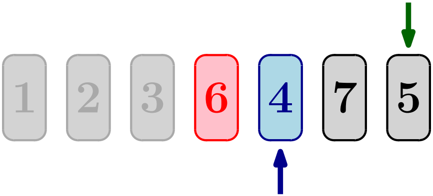
Tri rapide (*Quicksort*), version 2

Exemple :



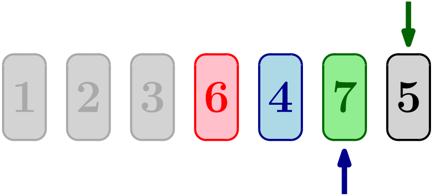
Tri rapide (*Quicksort*), version 2

Exemple :



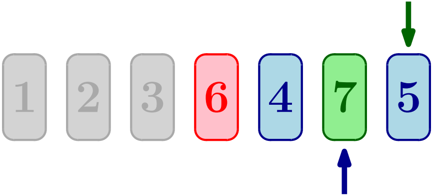
Tri rapide (*Quicksort*), version 2

Exemple :



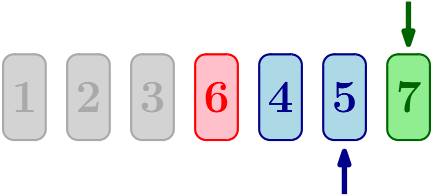
Tri rapide (*Quicksort*), version 2

Exemple :



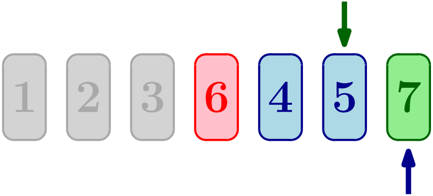
Tri rapide (*Quicksort*), version 2

Exemple :



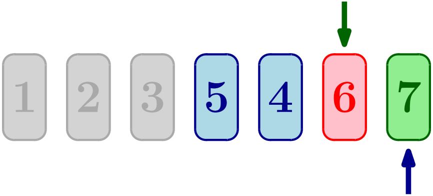
Tri rapide (*Quicksort*), version 2

Exemple :



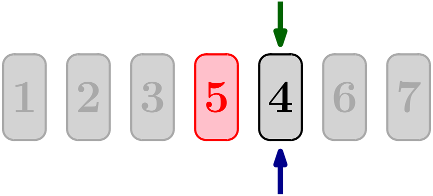
Tri rapide (*Quicksort*), version 2

Exemple :



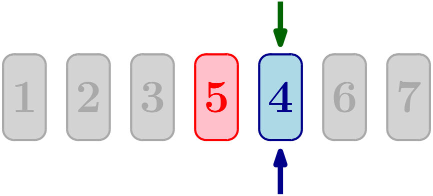
Tri rapide (*Quicksort*), version 2

Exemple :



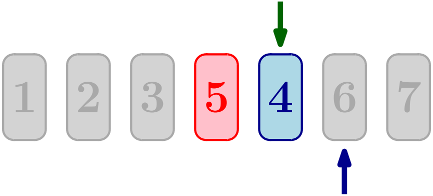
Tri rapide (*Quicksort*), version 2

Exemple :



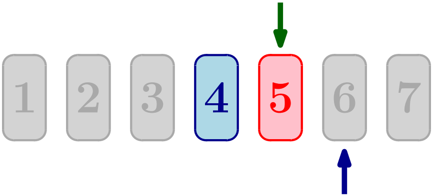
Tri rapide (*Quicksort*), version 2

Exemple :



Tri rapide (*Quicksort*), version 2

Exemple :



Tri rapide (*Quicksort*), version 2

Exemple :

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

Tri rapide, version *randomisée*

Il reste le cas problématique des tableaux (presque) triés

```
def partition(T, debut, fin) :
  alea = random.randint(debut, fin - 1)
  T[debut], T[alea] = T[alea], T[debut]
  pivot, gauche, droite = T[debut], debut + 1, fin - 1
  while gauche < droite :
    while T[gauche] < pivot : gauche += 1
    while T[droite] > pivot : droite -= 1
    if gauche < droite :
      T[gauche], T[droite] = T[droite], T[gauche]
      gauche, droite = gauche + 1, droite - 1
  T[debut], T[droite] = T[droite], pivot
  return droite
```

Rang

l'élément de rang `k` d'un tableau `T` est l'unique `x` de `T` tel que

- `T` contient $k - 1$ éléments plus petits que `x`
- `T` contient $\texttt{len(T)} - \texttt{k}$ éléments plus grands que `x`

Rang

l'élément de rang `k` d'un tableau `T` est l'unique `x` de `T` tel que

- `T` contient `k` $-$ 1 éléments plus petits que `x`
- `T` contient `len(T)` $-$ `k` éléments plus grands que `x`

Cas particuliers

- *si* `T` est trié : `T[k-1]`

Rang

l'élément de rang k d'un tableau T est l'unique x de T tel que

- T contient $k - 1$ éléments plus petits que x
- T contient len(T) $- k$ éléments plus grands que x

Cas particuliers

- *si* T est trié : T[k-1]
- élément de rang 1 : minimum(T)

Rang

l'élément de rang k d'un tableau T est l'unique x de T tel que

- T contient $k-1$ éléments plus petits que x
- T contient len(T) $-$ k éléments plus grands que x

Cas particuliers

- *si* T est trié : `T[k-1]`
- élément de rang 1 : `minimum(T)`
- élément de rang `len(T)` : `maximum(T)`

Rang

l'élément de rang `k` d'un tableau `T` est l'unique `x` de `T` tel que

- `T` contient $k - 1$ éléments plus petits que `x`
- `T` contient `len(T)` $-$ `k` éléments plus grands que `x`

Cas particuliers

- *si* `T` est trié : `T[k-1]`
- élément de rang 1 : `minimum(T)`
- élément de rang `len(T)` : `maximum(T)`
- élément « du milieu » : `médian(T)` (ou `médiane(T)`)

Rang

l'élément de rang `k` d'un tableau `T` est l'unique `x` de `T` tel que

- `T` contient $k - 1$ éléments plus petits que `x`
- `T` contient `len(T)` $-$ `k` éléments plus grands que `x`

Cas particuliers

- *si* `T` est trié : `T[k-1]`
- élément de rang 1 : `minimum(T)`
- élément de rang `len(T)` : `maximum(T)`
- élément « du milieu » : `médian(T)` (ou `médiane(T)`)

Rang

l'élément de rang `k` d'un tableau `T` est l'unique `x` de `T` tel que

- `T` contient $k - 1$ éléments plus petits que `x`
- `T` contient `len(T)` $-$ `k` éléments plus grands que `x`

Cas particuliers

- *si* `T` est trié : `T[k-1]`
- élément de rang 1 : `minimum(T)`
- élément de rang `len(T)` : `maximum(T)`
- élément « du milieu » : `médian(T)` (ou `médiane(T)`)
  si $n =$ `len(T)` impair : rang $\frac{1}{2}(n+1)$
  si ℓ pair : rang $\frac{1}{2}n$ ou $\frac{1}{2}n + 1$

Sélection dans un tableau

`selection(T, k)`

étant donné un tableau `T` et un entier `k`, déterminer l'élément de rang `k` de `T`

`selection(T, k)`

étant donné un tableau `T` et un entier `k`, déterminer l'élément de rang `k` de `T`

Solution n° 1

- trier `T`
- retourner `T[k-1]`

selection(T, k)

étant donné un tableau T et un entier k, déterminer l'élément de rang k de T

Solution n° 1

- trier T
- retourner T[k-1]

$\Longrightarrow$ $\Theta(n \log n)$ comparaisons (au pire)

minimum(T)

étant donné un tableau T, déterminer le plus petit élément de T

```
def min(T) :
  tmp = T[0]
  for elt in T :
    if elt < tmp : tmp = elt
  return tmp
```

$\Longrightarrow$ $n-1$ comparaisons (exactement)

maximum(T)

étant donné un tableau T, déterminer le plus grand élément de T

```
def max(T) :
  tmp = T[0]
  for elt in T :
    if elt > tmp : tmp = elt
  return tmp
```

$\Longrightarrow$ $n-1$ comparaisons (exactement)

Sélection – cas particuliers

`min_et_max_simultanés(T)`

étant donné un tableau `T`, déterminer le plus petit et le plus grand éléments de `T`

Sélection – cas particuliers

min_et_max_simultanés(T)

étant donné un tableau T, déterminer le plus petit et le plus grand éléments de T

```
def min_et_max(T) :       # cas où len(T) est impaire
  min = max = T[0]
  for elt1, elt2 in zip(T[1::2], T[2::2]) : # 2 par 2
    if elt1 < elt2 :
      if elt1 < min : min = elt1
      if elt2 > max : max = elt2
    else : ## échanger le rôle de elt1 et elt2
 return min, max
```

$$\Longrightarrow \frac{3}{2}(n-1) \text{ comparaisons (si } n \text{ impair)}$$

min_et_max_simultanés(T)

étant donné un tableau T, déterminer le plus petit et le plus grand éléments de T

```
def min_et_max(T) :      # cas où len(T) est paire
  if T[0] < T[1] : min, max = T[0], T[1]
  else : min, max = T[1], T[0]
  for elt1, elt2 in zip(T[2::2], T[3::2]) : # 2 par 2
    if elt1 < elt2 :
      if elt1 < min : min = elt1
      if elt2 > max : max = elt2
    else : ## échanger le rôle de elt1 et elt2
 return min, max
```

$$\Longrightarrow \frac{3n}{2} - 2 \text{ comparaisons (si } n \text{ pair)}$$

Sélection – cas général

```
def selection(T, k) :
  for i in range(k) :
    tmp = i
    for j in range(i, len(T)) :
      if T[j] < T[tmp] : tmp = j
    T[i], T[tmp] = T[tmp], T[i]
  return T[k-1]
```

Sélection – cas général

```
def selection(T, k) :
  for i in range(k) :
    tmp = i
    for j in range(i, len(T)) :
      if T[j] < T[tmp] : tmp = j
    T[i], T[tmp] = T[tmp], T[i]
  return T[k-1]
```

$\Longrightarrow$ kn comparaisons (environ)

si k est petit, c'est sensiblement mieux que $\Theta(n \log n)$!

Sélection rapide (*Quickselect*)

```
def selection_rapide(T, k) :
  if len(T) == 1 and k == 1 : return T[0]
  pivot, gauche, droite = partition(T)
  position = len(gauche) + 1
  if position == k : return pivot
  if position > k : return selection_rapide(gauche, k)
  return selection_rapide(droite, k - position)
```

Sélection rapide (*Quickselect*)

```
def selection_rapide(T, k) :
  if len(T) == 1 and k == 1 : return T[0]
  pivot, gauche, droite = partition(T)
  position = len(gauche) + 1
  if position == k : return pivot
  if position > k : return selection_rapide(gauche, k)
  return selection_rapide(droite, k - position)
```

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

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

Sélection rapide (*Quickselect*)

Complexité de `selection_rapide` en moyenne (*admis*) :

$\Theta(n)$ comparaisons

Sélection rapide (*Quickselect*)

Complexité de `selection_rapide` en moyenne (*admis*) :

$\Theta(n)$ comparaisons

En choisissant comme pivot la médiane d'un échantillon de 5 éléments, on obtient un algorithme de complexité $\Theta(n)$ dans le pire des cas (admis)