Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
`dominique.poulalhon@irif.fr`

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2019-2020

Rappel des épisodes précédents

Borne inférieure de complexité

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

Rappel des épisodes précédents

Borne inférieure de complexité

tout algorithme de tri par comparaisons nécessite au moins $\Theta(n \log n)$ 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 au moins $\Theta(n \log n)$ 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 au moins $\Theta(n \log n)$ 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 ?

Action des transpositions sur les permutations

Action des transpositions sur les permutations

- produit à gauche par $(i\ j) \iff$ échange des valeurs i et j

Rappels (bis) : 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

Rappels (bis) : transpositions

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

Rappels (bis) : 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 exactement 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 exactement ce que calcule n'importe quel algorithme de tri par comparaisons/échanges)

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

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 exactement ce que calcule n'importe quel 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 :*

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

Aparté : génération aléatoire de permutations

`RandomPermutation(n)`

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

`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

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
```

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

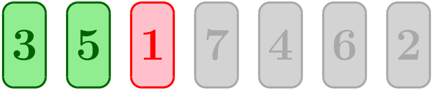
```python
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

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




```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	**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

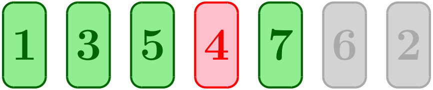

```python
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Tri par insertion dans un tableau

1	3	5	7	4	6	2

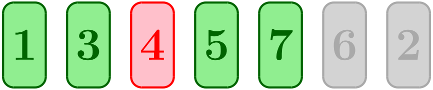
```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Tri par insertion dans un tableau



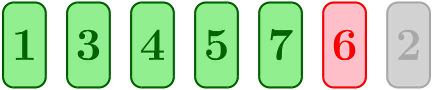

```python
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Tri par insertion dans un tableau




```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




```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	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**	**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 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

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

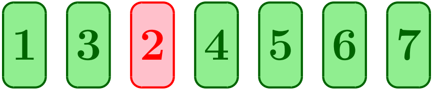
```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Tri par insertion dans un tableau

1 3 4 2 5 6 7

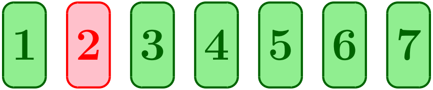
```
def tri_insertion(T) : # version "par échanges successifs"
  for i in range(1, len(T)) :
    for j in range(i, 0, -1) : #parcours de droite à gauche
      if T[j-1] > T[j] :
        T[j-1], T[j] = T[j], T[j-1]
      else : break
  return T
```

Tri par insertion dans un tableau




```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




```
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

Inversions

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 :

- 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

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

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

Inversions

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

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'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'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

$$\iff$$

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

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

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

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

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 récusivité *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 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*

Tri rapide (*Quicksort*)

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

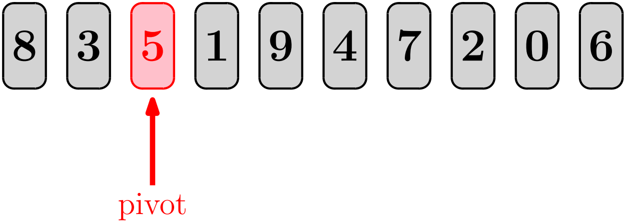
Exemple :

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

Tri rapide (*Quicksort*)

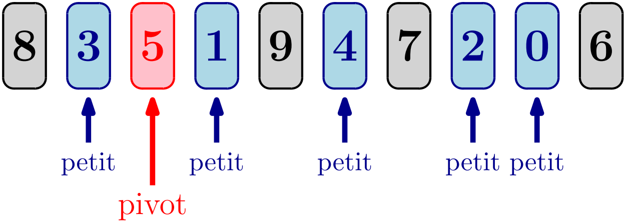
Idée : faire un prétraitement *avant* les appels récursifs pour éviter d'en avoir besoin *après*

Exemple :



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

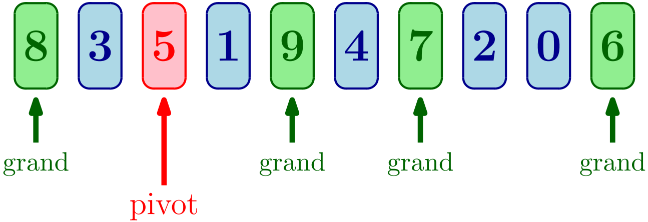
Exemple :



Tri rapide (*Quicksort*)

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

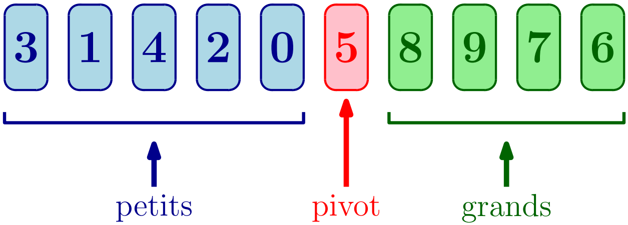
Exemple :



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 :

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

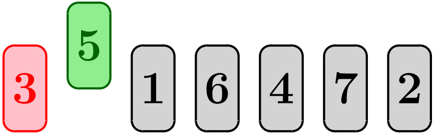
Tri rapide (*Quicksort*), version 1

Exemple :

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

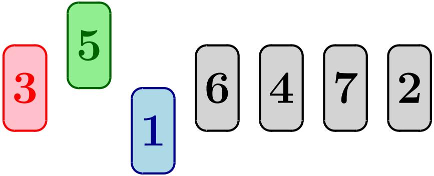
Tri rapide (*Quicksort*), version 1

Exemple :



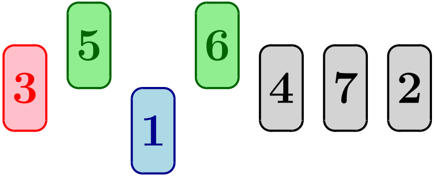
Tri rapide (*Quicksort*), version 1

Exemple :



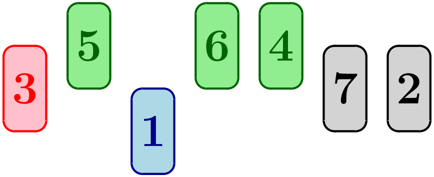
Tri rapide (*Quicksort*), version 1

Exemple :



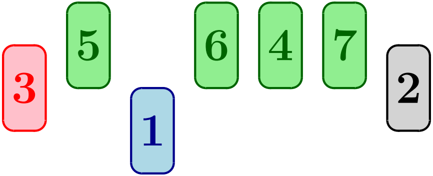
Tri rapide (*Quicksort*), version 1

Exemple :



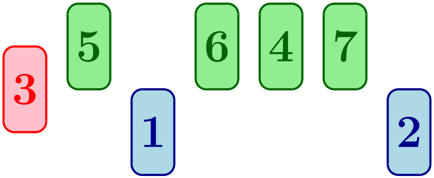
Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :



Tri rapide (*Quicksort*), version 1

Exemple :

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

Tri rapide (*Quicksort*), version 1

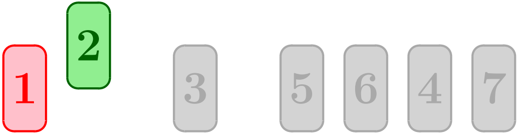
Exemple :

Tri rapide (*Quicksort*), version 1

Exemple :

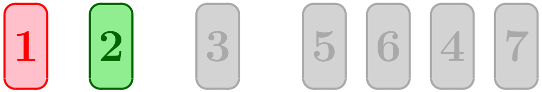
TRI RAPIDE (*Quicksort*), VERSION 1

Exemple :



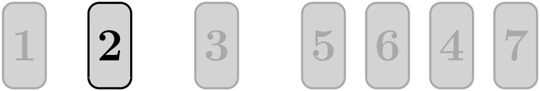
Tri rapide (*Quicksort*), version 1

Exemple :



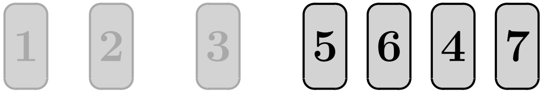
Tri rapide (*Quicksort*), version 1

Exemple :



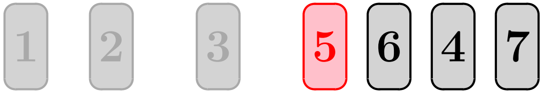
TRI RAPIDE (*Quicksort*), VERSION 1

Exemple :



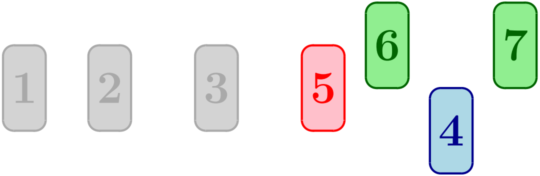
Tri rapide (*Quicksort*), version 1

Exemple :



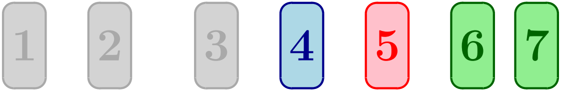
TRI RAPIDE (*Quicksort*), VERSION 1

Exemple :



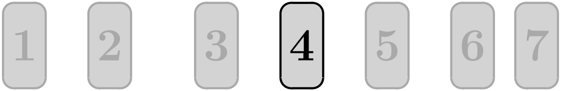
Tri rapide (*Quicksort*), version 1

Exemple :



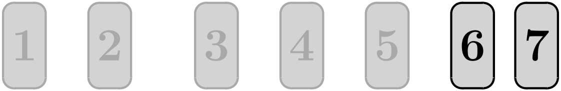
Tri rapide (*Quicksort*), version 1

Exemple :



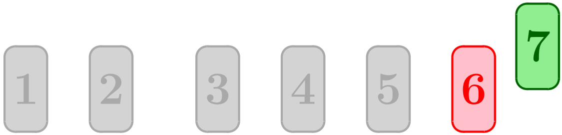
Tri rapide (*Quicksort*), version 1

Exemple :



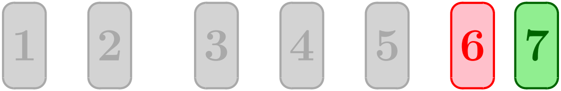
Tri rapide (*Quicksort*), version 1

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

```
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

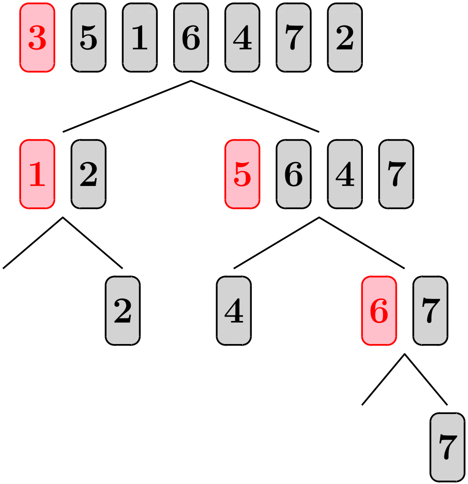
Complexité de `partition` : $\Theta(n)$ comparaisons

```
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

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

Démonstration pour tout tableau T de longueur n, si partition coupe T en gauche et droite, le nombre de comparaisons du tri rapide s'exprime comme $C(\mathtt{T}) = (n-1) + C(\mathtt{gauche}) + C(\mathtt{droite})$ (coût de partition et des deux appels récursifs).

Or toutes les étapes de tri_rapide(gauche) et tri_rapide(droite) sont incluses dans tri_rapide(gauche+droite). Cela peut se démontrer par récurrence sur len(gauche) :

- c'est vrai pour gauche de longueur 0 ;
- si len(gauche) $\geqslant 1$, la première étape de tri_rapide(gauche+droite) consiste à le partitionner selon gauche[0], ce qui inclut la première étape de tri_rapide(gauche). On obtient une partition de la forme (gg, gd+droite). Par hypothèse de récurrence, tri_rapide(gd+droite) inclut toutes les étapes de tri_rapide(gd) et tri_rapide(droite), donc tri_rapide(gauche+droite) inclut :
  - la première étape de tri_rapide(gauche)
  - tri_rapide(gg)
  - tri_rapide(gd)
  - tri_rapide(droite)

  donc tri_rapide(gauche) et tri_rapide(droite), cqfd.

Donc : $$C(\mathtt{gauche}) + C(\mathtt{droite}) \leqslant C(\mathtt{gauche + droite}) \leqslant C_{\mathrm{pire}}(n-1),$$

le pire cas étant réalisé par exemple si T est trié, d'où finalement la relation de récurrence :

$$C_{\mathrm{pire}}(n) = (n-1) + C_{\mathrm{pire}}(n-1),$$

dont la solution est en $\Theta(n^2)$.

Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :

$\boxed{\mathbf{3}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{6}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{2}}$

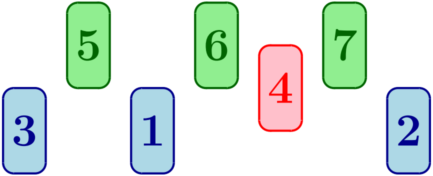
Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :

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

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 :

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

Tri rapide (*Quicksort*), version 1

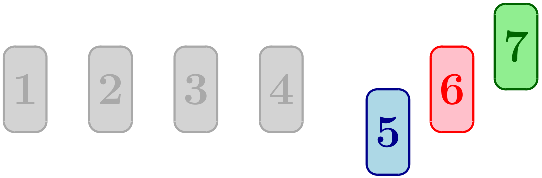
Exemple en choisissant miraculeusement toujours le pivot optimal :

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 :

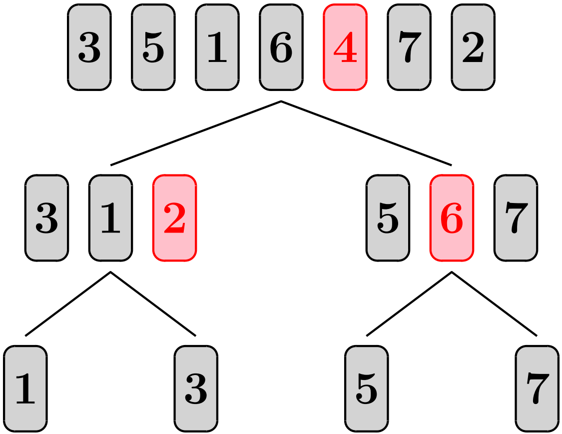
Tri rapide (*Quicksort*), version 1

Exemple en choisissant miraculeusement toujours le pivot optimal :

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

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

Tri rapide (*QuickSort*), version 2

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

avec une partition en place à la manière du *tri-drapeau* (*cf.* TD ?)

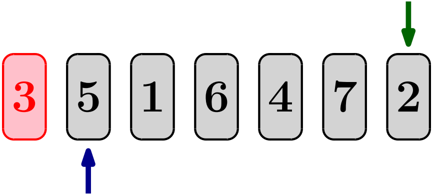
Tri rapide (*QuickSort*), version 2

Exemple :

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

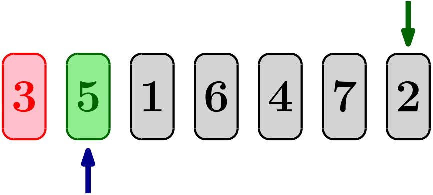
Tri rapide (*QuickSort*), version 2

Exemple :



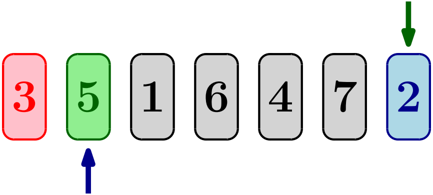
Tri rapide (*QuickSort*), version 2

Exemple :



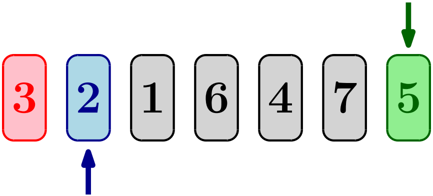
Tri rapide (*QuickSort*), version 2

Exemple :



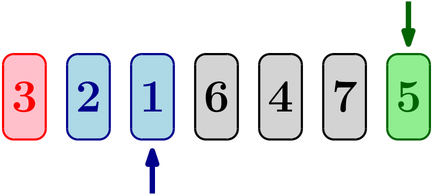
Tri rapide (*QuickSort*), version 2

Exemple :



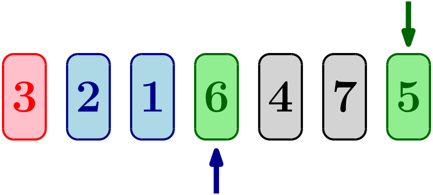
Tri rapide (*QuickSort*), version 2

Exemple :



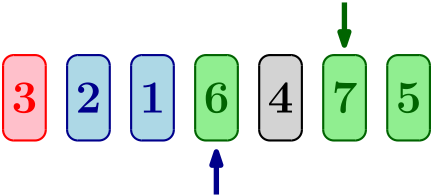
Tri rapide (*QuickSort*), version 2

Exemple :



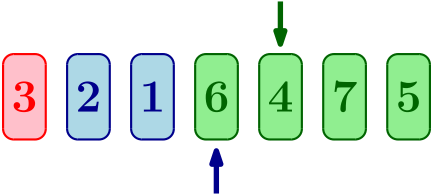
Tri rapide (*QuickSort*), version 2

Exemple :



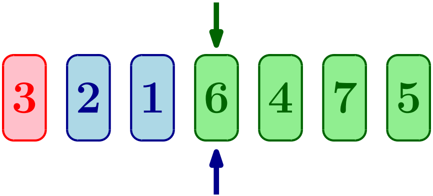
Tri rapide (*QuickSort*), version 2

Exemple :



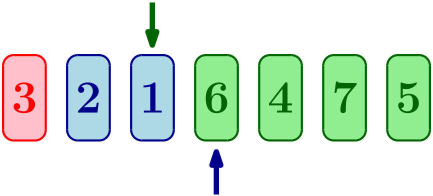
Tri rapide (*QuickSort*), version 2

Exemple :



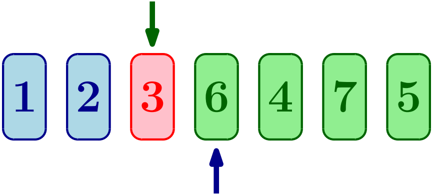
Tri rapide (*QuickSort*), version 2

Exemple :



Tri rapide (*QuickSort*), version 2

Exemple :



Remarque : si le tableau a des répétitions, un vrai tri-drapeau à 3 valeurs permet de regrouper tous les éléments égaux au pivot

Tri rapide (*QuickSort*), version 2

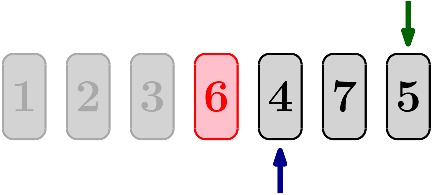
Exemple :

Tri rapide (*QuickSort*), version 2

Exemple :

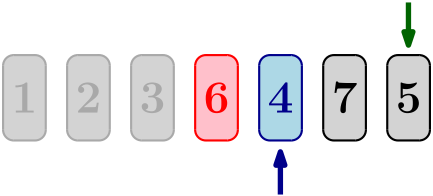
Tri rapide (*QuickSort*), version 2

Exemple :



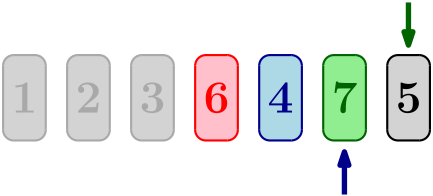
Tri rapide (*QuickSort*), version 2

Exemple :



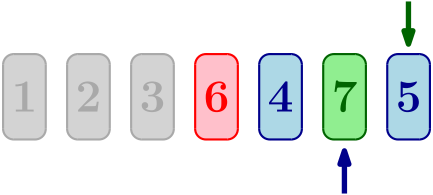
Tri rapide (*QuickSort*), version 2

Exemple :



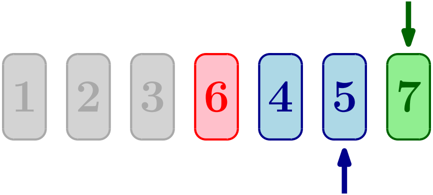
Tri rapide (*QuickSort*), version 2

Exemple :



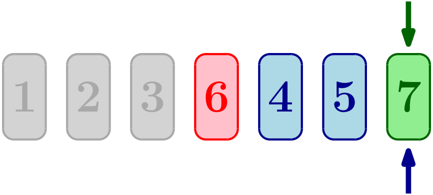
Tri rapide (*QuickSort*), version 2

Exemple :



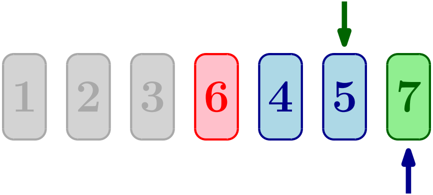
Tri rapide (*QuickSort*), version 2

Exemple :



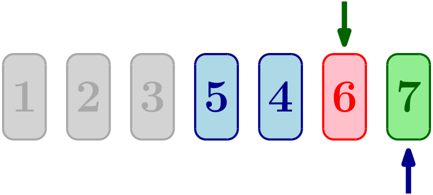
Tri rapide (*QuickSort*), version 2

Exemple :



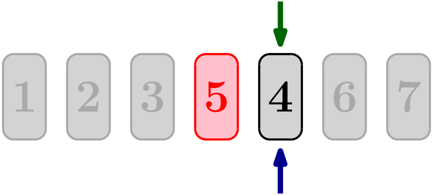
Tri rapide (*QuickSort*), version 2

Exemple :



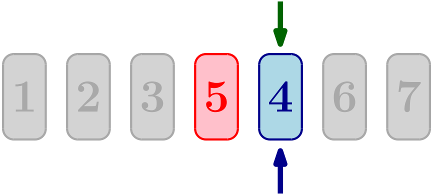
Tri rapide (*QuickSort*), version 2

Exemple :



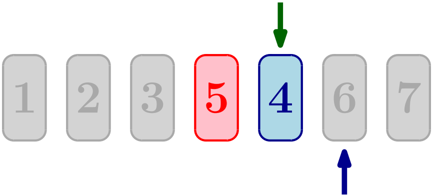
Tri rapide (*QuickSort*), version 2

Exemple :



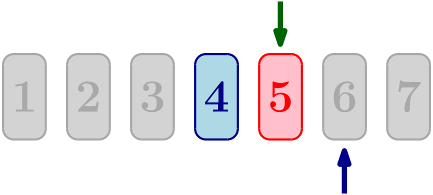
Tri rapide (*QuickSort*), version 2

Exemple :



Tri rapide (*QuickSort*), version 2

Exemple :



Tri rapide (*QuickSort*), version 2

Exemple :

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

Tri rapide (*QuickSort*), version 2

```
def partition(T, debut, fin) :
  if fin - debut == 2 and T[debut] < T[debut+1] :
    return debut+1
  pivot, gauche, droite = T[debut], debut + 1, fin - 1
  while gauche <= droite :
    if T[gauche] < pivot : gauche += 1
    elif T[droite] > pivot : droite -= 1
    # avec >= si T contient des doublons
    else : T[gauche], T[droite] = T[droite], T[gauche]
  # ici : gauche = droite + 1, T[droite] <= pivot < T[gauche]
  # (et même < sauf si droite = debut)
  T[debut], T[droite] = T[droite], pivot
  return droite
```

Tri rapide, version *randomisée*

reste le cas problématique des tableaux (presque) triés

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

  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 :
    if T[gauche] < pivot : gauche += 1
    elif T[droite] > pivot : droite -= 1
    else : T[gauche], T[droite] = T[droite], T[gauche]
  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 au plus $k-1$ éléments strictement plus petits que `x`
- `T` contient au plus `len(T)` $-$ `k` éléments strictement plus grands que `x`

Rang

si T est un tableau sans doublon, l'élément de rang k de 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

Rang

si `T` est un tableau sans doublon, l'élément de rang `k` de `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

si T est un tableau sans doublon, l'élément de rang k de 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

si T est un tableau sans doublon, l'élément de rang k de 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

si T est un tableau sans doublon, l'élément de rang k de 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

si T est un tableau sans doublon, l'élément de rang k de 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

si `T` est un tableau sans doublon, l'élément de rang `k` de `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 n 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]`

Sélection dans un tableau

`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)

Sélection – cas particuliers

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)

Sélection – cas particuliers

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`

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) :
  min = max = T[-1]
  for elt1, elt2 in zip(T[0::2], T[1::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
```

$\Longrightarrow \dfrac{3n}{2}$ comparaisons (si n 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 $\Theta(n \log n)$
- si k est en $\Theta(n)$, c'est sensiblement moins bien

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
  elif position > k :
    return selection_rapide(gauche, k)
  else :
    return selection_rapide(droite, k - position)
```

Sélection rapide (*QuickSelect*)

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

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

Complexité de `selection_rapide` en moyenne (*admis*) :
$\Theta(n)$ comparaisons

Sélection rapide (*QuickSelect*)

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

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

Complexité de `selection_rapide` en moyenne (*admis*) : $\Theta(n)$ comparaisons

En choisissant comme pivot la médiane des $\frac{n}{5}$ médianes de paquets de 5 éléments, on obtient un algorithme de complexité $\Theta(n)$ dans le pire des cas (admis)