Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon

dominique.poulalhon@irif.fr

Université Paris Cité

L2 Informatique & DL Bio-Info, Jap-Info, Math-Info

Année universitaire 2023-2024

Auto-évaluations n° 1 et 2 disponibles sur moodle

Évaluation n° 1 (sur moodle) : samedi 16 mars, plage strictement incluse dans la plage 10h30-12h30

Partiel : vendredi 22 mars, 16h15-18h00

(Rappel) 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

(Rappel) 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*

$\Longrightarrow$ partitionnement du tableau pour séparer les « petits » éléments des « grands » éléments

(Rappel) 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
```

(Rappel) 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

(Rappel) 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)
```

(Rappel) 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

(Rappel) 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

(Rappel) 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_en_place(T, debut=0, fin=None) :
  # trie T[debut:fin] : indice debut inclus, fin exclu
  # appel initial tri_rapide_en_place(T) : on veut debut=0, fin=len(T)
  if fin is None : fin = len(T)

  if fin - debut < 2 : return

  indice_pivot = partition_en_place(T, debut, fin)
  # place pivot à sa place finale (indice_pivot),
  # les éléments plus petits à sa gauche et les plus grands à sa droite

  tri_rapide_en_place(T, debut, indice_pivot)
  tri_rapide_en_place(T, indice_pivot + 1, fin)
```

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

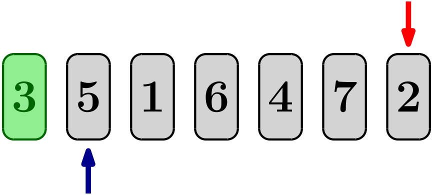
Tri rapide (*Quicksort*), version 2

Exemple :

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

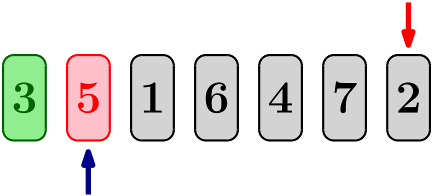
TRI RAPIDE (*Quicksort*), VERSION 2

Exemple :



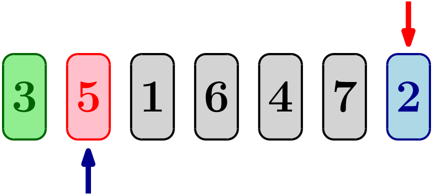
Tri rapide (*Quicksort*), version 2

Exemple :



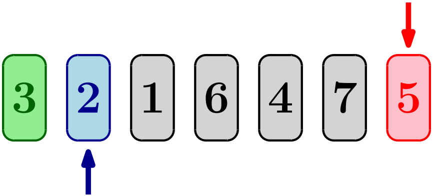
Tri rapide (*Quicksort*), version 2

Exemple :



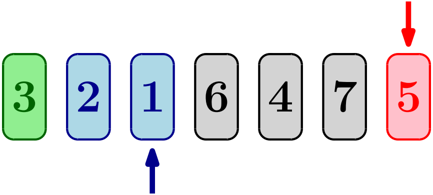
Tri rapide (*Quicksort*), version 2

Exemple :



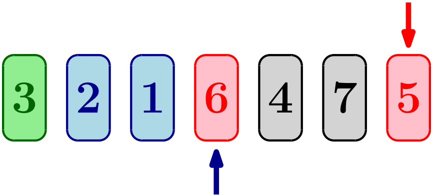
Tri rapide (*Quicksort*), version 2

Exemple :



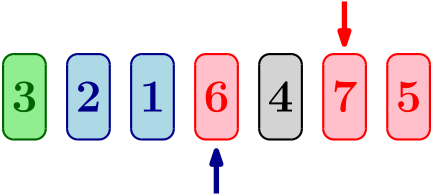
TRI RAPIDE (*Quicksort*), VERSION 2

Exemple :



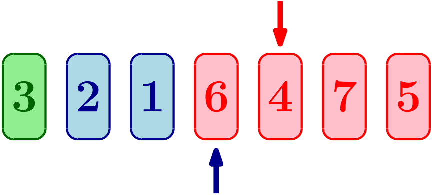
Tri rapide (*Quicksort*), version 2

Exemple :



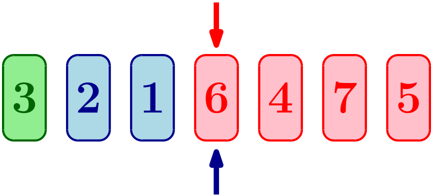
Tri rapide (*Quicksort*), version 2

Exemple :



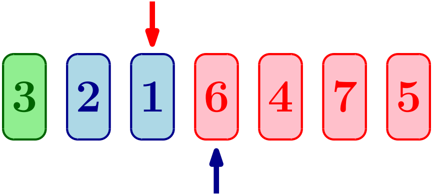
Tri rapide (*Quicksort*), version 2

Exemple :



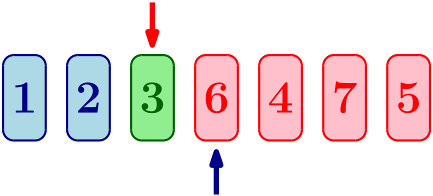
Tri rapide (*Quicksort*), version 2

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, et donc de faire des appels récursifs sur de plus petits sous-tableaux

TRI RAPIDE (*Quicksort*), VERSION 2

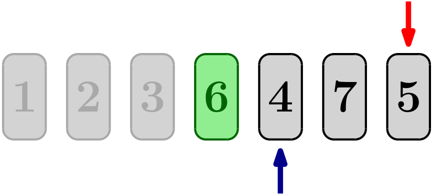
Exemple :

Tri rapide (*Quicksort*), version 2

Exemple :

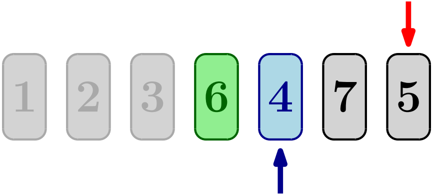
Tri rapide (*Quicksort*), version 2

Exemple :



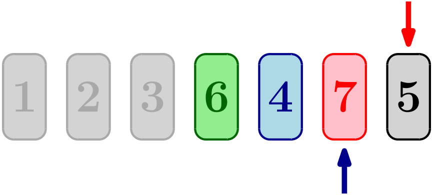
Tri rapide (*Quicksort*), version 2

Exemple :



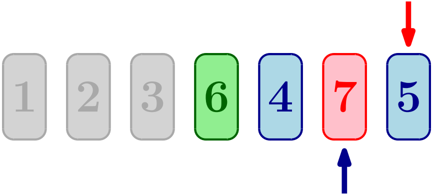
Tri rapide (*Quicksort*), version 2

Exemple :



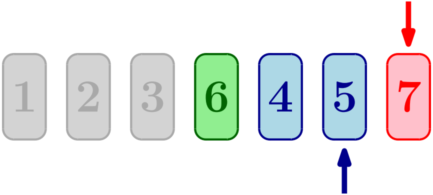
Tri rapide (*Quicksort*), version 2

Exemple :



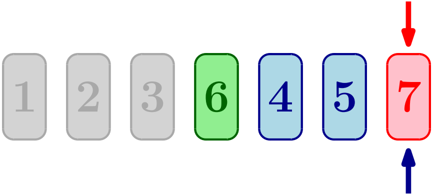
TRI RAPIDE (*Quicksort*), VERSION 2

Exemple :



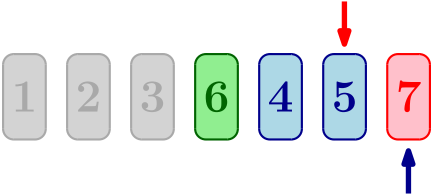
Tri rapide (*Quicksort*), version 2

Exemple :



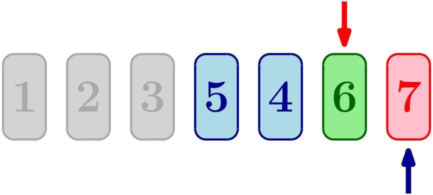
Tri rapide (*Quicksort*), version 2

Exemple :



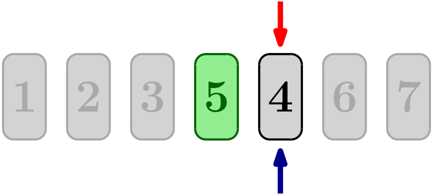
Tri rapide (*Quicksort*), version 2

Exemple :



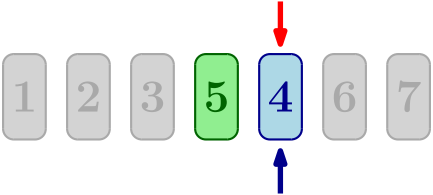
Tri rapide (*Quicksort*), version 2

Exemple :



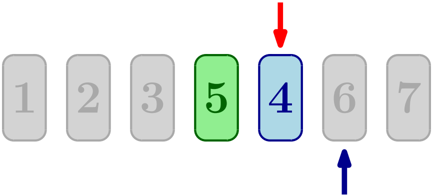
Tri rapide (*Quicksort*), version 2

Exemple :



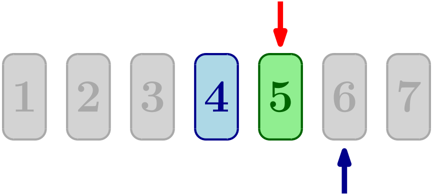
Tri rapide (*Quicksort*), version 2

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_en_place(T, debut, fin) : # T supposé sans doublon
  # initialisation des curseurs
  pivot, gauche, droite = T[debut], debut + 1, fin - 1

  # déplacement des curseurs
  while gauche <= droite :
    while gauche < fin and T[gauche] < pivot : gauche += 1
    while droite > debut and T[droite] > pivot : droite -= 1
    # avec <= ou >= si T contient des doublons
    if gauche < droite :
      T[gauche], T[droite] = T[droite], T[gauche]

  # ici : gauche = droite + 1, T[droite] <= pivot < T[gauche]
  # (et même < sauf si droite = debut)

  # mise en place du pivot
  T[debut], T[droite] = T[droite], pivot

  return droite
```

TRI RAPIDE (*Quicksort*), VERSION 2

Ou bien :

```
def partition_en_place(T, debut, fin) : # T supposé sans doublon
  # initialisation des curseurs
  pivot, gauche, droite = T[debut], debut + 1, fin - 1

  # déplacement des curseurs
  while gauche <= droite :
    if T[gauche] < pivot : gauche += 1
    elif T[droite] > pivot : droite -= 1
    # avec <= ou >= 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)

  # mise en place du pivot
  T[debut], T[droite] = T[droite], pivot

  return droite
```

Tri rapide, version *randomisée*

reste le cas problématique des tableaux (presque) triés : il suffit de choisir un pivot aléatoire pour se placer dans le « cas moyen »

Tri rapide, version *randomisée*

reste le cas problématique des tableaux (presque) triés : il suffit de choisir un pivot aléatoire pour se placer dans le « cas moyen »

```
def partition_en_place_randomisee(T, debut, fin) :
  # T supposé sans doublon

  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
```

Complément : la sélection rapide

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)`)
  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)*

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)*

`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
```

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

Sélection – cas général

```
def selection(T, k) : # comme un tri par sélection interrompu
  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) : # comme un tri par sélection interrompu
  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*)

Idée : utiliser le partitionnement du tri rapide

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si $\mathtt{r} = \mathtt{k}$: le pivot est l'élément de rang `k` $\implies$ recherche terminée

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si `r` $=$ `k` : le pivot est l'élément de rang `k` $\Longrightarrow$ recherche terminée
- si `r` $>$ `k` : le pivot est supérieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche ***à gauche***

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si $\texttt{r} = \texttt{k}$: le pivot est l'élément de rang `k` $\Longrightarrow$ recherche terminée
- si $\texttt{r} > \texttt{k}$: le pivot est supérieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche ***à gauche***
- si $\texttt{r} < \texttt{k}$: le pivot est inférieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche ***à droite***

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si $\texttt{r} = \texttt{k}$: le pivot est l'élément de rang `k` $\Longrightarrow$ recherche terminée
- si $\texttt{r} > \texttt{k}$: le pivot est supérieur à l'élément de rang `k` $\Longrightarrow$ poursuivre la recherche ***à gauche***
- si $\texttt{r} < \texttt{k}$: le pivot est inférieur à l'élément de rang `k` $\Longrightarrow$ poursuivre la recherche ***à droite***

$\Longrightarrow$ dans tous les cas, (au plus) un seul appel récursif est nécessaire

Sélection rapide (*Quickselect*)

```
def selection_rapide(T, k) :
  if len(T) == 1 : return T[0] if k == 1 else None

  # version naïve
  pivot, gauche, droite = partition(T)
  rang_pivot = len(gauche) + 1

  if rang_pivot == k :
    return pivot
  elif rang_pivot > k :
    return selection_rapide(gauche, k)
  else :
```

Sélection rapide (*Quickselect*)

```
def selection_rapide(T, k) :
  if len(T) == 1 : return T[0] if k == 1 else None

  # version naïve
  pivot, gauche, droite = partition(T)
  rang_pivot = len(gauche) + 1

  if rang_pivot == k :
    return pivot
  elif rang_pivot > k :
    return selection_rapide(gauche, k)
  else :
    return selection_rapide(droite, k - rang_pivot)
```

Sélection rapide (*Quickselect*)

```
def selection_rapide_en_place(T, k, deb=0, fin=None) :
  if fin is None : fin = len(T)
  if fin-deb == 1 : return T[0] if k == 1 else None

  indice_pivot = partition_en_place(T, debut, fin)
  rang_pivot = indice_pivot + 1

  if rang_pivot == k :
    return T[indice_pivot]
  elif rang_pivot > k :
    return selection_rapide(T, k, deb, indice_pivot)
  else :
    return selection_rapide(T, k - rang_pivot, rang_pivot, fin)
```

Sélection rapide (*Quickselect*)

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

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

variante (algorithme BFPRT) : 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)*

Conclusion : qualités et défauts des algorithmes de tri classiques

Tri par fusion

- $\Theta(n \log n)$ comparaisons *au pire* (mais *dans tous les cas*),
- la *constante cachée* dans le Θ est importante,
- *stable* mais *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),
- *stable* et *en place*

Tri rapide

- $\Theta(n^2)$ comparaisons *au pire*
- $\Theta(n \log n)$ comparaisons *en moyenne*... et *au mieux*,
- version naïve : *stable* mais *pas en place*, mauvais cas « assez probables »,
- version en place et randomisée : *en place* mais *pas stable*, mauvais cas sans caractéristiques particulières (donc peu probables)

Comment conjuguer ces qualités ?

Tri par comparaisons « idéal » :

- pire cas (et cas moyen) en $\Theta(n \log n)$,
- meilleur cas en $\Theta(n)$ (et correspondant à des cas « probables en pratique »),
- en place,
- stable.

pour s'en approcher, on peut concevoir des tris hybrides : tris utilisant des mécanismes inspirés de plusieurs algorithmes de tri différents

- SedgeSort (hybride de tri rapide et de tri par insertion),
- TimSort (hybride de tri fusion et de tri par insertion),
- IntroSort (hybride de tri rapide et de tri par tas)...

et il est parfois possible de tirer parti des caractéristiques des données à trier pour sortir du cadre des tris par comparaisons : tri par dénombrement, tri par paquets, tri par base (RadixSort)...