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

Prochaines dates diverses

Vendredi 28 février, 10h-12h, salle 1021 (Sophie Germain) : débat sur la réforme des retraites avec Michaël Zemmour, économiste spécialiste du financement de la protection sociale

Contrôle n° 1 mercredi 4 mars après le cours

Amphis 8C et 13E de 16h à 17h30

Jeudi 5 mars : journée « l'université et la recherche s'arrêtent »

Algorithmes pour les ensembles

`recherche(x, L)`

Étant donné une liste `L` et un élément `x`, déterminer si `x` apparaît dans `L`

`recherche(x, L)`

Étant donné une liste `L` et un élément `x`, déterminer si `x` apparaît dans `L`

```
def recherche_sequentielle(x, L) :
  for elt in L :
    if elt == x : return True
  return False
```

(remarque : c'est ce que fait le test (`x in L`))

recherche(x, L)

Étant donné une liste L et un élément x, déterminer si x apparaît dans L

variante : retourner une position où x apparaît

```
def recherche_sequentielle(x, L) :
  for (i, elt) in enumerate(L) :
    # liste des couples (position, contenu)
    if elt == x : return i
  return -1
```

recherche(x, L)

Étant donné une liste L et un élément x, déterminer si x apparaît dans L

variante : retourner une position où x apparaît

```
def recherche_sequentielle(x, L) :
  for (i, elt) in enumerate(L) :
    # liste des couples (position, contenu)
    if elt == x : return i
  return -1
```

(remarque : c'est très exactement ce que fait `L.index(x)`)

`occurrences(x, L)`

Étant donné une liste `L` et un élément `x`, compter les occurrences de `x` dans `L`

occurrences(x, L)

Étant donné une liste L et un élément x, compter les occurrences de x dans L

```
def occurrences(x, L) :
  res = 0
  for elt in L :
    if elt == x : res += 1
  return res
```

(remarque : c'est ce que fait `L.count(x)`)

`max(L)`

Étant donné une liste `L` contenant des éléments *comparables*, déterminer le plus grand élément qui apparaît dans `L`

max(L)

Étant donné une liste L contenant des éléments *comparables*, déterminer le plus grand élément qui apparaît dans L

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

Complexité de ces recherches

opération(s) élémentaire(s)

- déplacements dans la liste
- comparaisons d'éléments
- (parfois) affectations, incrémentations de compteurs

opération(s) élémentaire(s)

- déplacements dans la liste
- comparaisons d'éléments
- (parfois) affectations, incrémentations de compteurs

toutes effectuées en nombre équivalent

$\Longrightarrow$ pour simplifier, on ne compte que les comparaisons

Complexité de ces recherches

opération(s) élémentaire(s)

- comparaisons d'éléments

`max(L)` $\Longrightarrow n - 1 = \Theta(n)$ comparaisons

Complexité de ces recherches

opération(s) élémentaire(s)

- comparaisons d'éléments

`max(L)` $\Longrightarrow n-1=\Theta(n)$ comparaisons

`occurrences(x, L)` $\Longrightarrow n=\Theta(n)$ comparaisons

Complexité de ces recherches

opération(s) élémentaire(s)

- comparaisons d'éléments

`max(L)` $\Longrightarrow$ $n-1=\Theta(n)$ comparaisons

`occurrences(x, L)` $\Longrightarrow$ $n=\Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`
$\Longrightarrow$ selon les cas, entre 1 et n comparaisons

Complexité de ces recherches

opération(s) élémentaire(s)

- comparaisons d'éléments

`max(L)` $\Longrightarrow$ $n-1=\Theta(n)$ comparaisons

`occurrences(x, L)` $\Longrightarrow$ $n=\Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

$\Longrightarrow$ selon les cas, entre 1 et n comparaisons

$\Longrightarrow$ on ne peut plus parler de « la » complexité

Complexité de ces recherches

opération(s) élémentaire(s)

- comparaisons d'éléments

`max(L)` $\Longrightarrow$ $n - 1 = \Theta(n)$ comparaisons

`occurrences(x, L)` $\Longrightarrow$ $n = \Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

$\Longrightarrow$ selon les cas, entre 1 et n comparaisons

$\Longrightarrow$ on ne peut plus parler de « la » complexité

- $\Theta(n)$ comparaisons au pire – en particulier dans le cas *défavorable*

Complexité de ces recherches

opération(s) élémentaire(s)

- comparaisons d'éléments

`max(L)` $\Longrightarrow$ $n-1=\Theta(n)$ comparaisons

`occurrences(x, L)` $\Longrightarrow$ $n=\Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

$\Longrightarrow$ selon les cas, entre 1 et n comparaisons

$\Longrightarrow$ on ne peut plus parler de « la » complexité

- $\Theta(n)$ comparaisons au pire – en particulier dans le cas *défavorable*
- $\frac{n+1}{2}=\Theta(n)$ en moyenne dans le cas *favorable* *(sous l'hypothèse que la position de l'élément cherché suit la probabilité uniforme)*

Complexité de ces recherches

opération(s) élémentaire(s)

- comparaisons d'éléments

`max(L)` $\implies n-1 = \Theta(n)$ comparaisons

`occurrences(x, L)` $\implies n = \Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

$\implies$ selon les cas, entre 1 et n comparaisons

$\implies$ on ne peut plus parler de « la » complexité

- $\Theta(n)$ comparaisons au pire – en particulier dans le cas *défavorable*
- $\frac{n+1}{2} = \Theta(n)$ en moyenne dans le cas *favorable*
- $\Theta(n)$ comparaisons en moyenne

Complexité de ces recherches

Peut-on faire mieux que $\Theta(n)$?

max(T)

Étant donné un *tableau* T *trié*, déterminer le plus grand élément qui apparaît dans T

Recherche dans un *tableau trié*

max(T)

Étant donné un *tableau* T *trié*, déterminer le plus grand élément qui apparaît dans T

```
def max_si_trie(T) :
  if len(T) == 0 : return None
  return T[-1]
```

Recherche dans un *tableau trié*

max(T)

Étant donné un *tableau* T *trié*, déterminer le plus grand élément qui apparaît dans T

```
def max_si_trie(T) :
  if len(T) == 0 : return None
  return T[-1]
```

$\Longrightarrow$ $\Theta(1)$ comparaisons

Recherche dans un *tableau trié*

`recherche(x, T)`

Étant donné un *tableau* `T` *trié* et un élément `x`, déterminer si `x` apparaît dans `T`

Recherche dans un *tableau trié*

recherche(x, T)

Étant donné un *tableau* T *trié* et un élément x, déterminer si x apparaît dans T

Idée 1 : interrompre la recherche séquentielle

```
def recherche_sequentielle(x, L) :
  for elt in L :
    if elt == x : return True
    else if elt > x : return False
  return False
```

Recherche dans un *tableau trié*

recherche(x, T)

Étant donné un *tableau* T *trié* et un élément x, déterminer si x apparaît dans T

Idée 1 : interrompre la recherche séquentielle

```
def recherche_sequentielle(x, L) :
  for elt in L :
    if elt == x : return True
    else if elt > x : return False
  return False
```

$\Longrightarrow$ cas favorable inchangé, et tout de même $\Theta(n)$ comparaisons au pire et en moyenne dans le cas défavorable

recherche(x, T)

Étant donné un *tableau* T *trié* et un élément x, déterminer si x apparaît dans T

Idée 2 : la dichotomie (stratégie « diviser pour régner »)

```
def recherche_dicho(x, T) : # ATTENTION version trop naïve
  if len(T) == 0 : return False
  milieu = len(T)//2
  if x == T[milieu] : return True
  elif x < T[milieu] : return recherche_dicho(x, T[:milieu])
  else : return recherche_dicho(x, T[milieu+1:])
```

Recherche dans un *tableau trié*

```
def recherche_dicho(x, T) : # ATTENTION version trop naïve
  if len(T) == 0 : return False
  milieu = len(T)//2
  if x == T[milieu] : return True
  elif x < T[milieu] : return recherche_dicho(x, T[:milieu])
  else : return recherche_dicho(x, T[milieu+1:])
```

Quelle complexité ?

Recherche dans un *tableau trié*

```
def recherche_dicho(x, T) : # ATTENTION version trop naïve
  if len(T) == 0 : return False
  milieu = len(T)//2
  if x == T[milieu] : return True
  elif x < T[milieu] : return recherche_dicho(x, T[:milieu])
  else : return recherche_dicho(x, T[milieu+1:])
```

Quelle complexité ?

$C(n) = 2 + C(\lfloor \frac{n}{2} \rfloor)$ comparaisons (au pire) pour T de taille n

Recherche dans un *tableau trié*

```
def recherche_dicho(x, T) : # ATTENTION version trop naïve
  if len(T) == 0 : return False
  milieu = len(T)//2
  if x == T[milieu] : return True
  elif x < T[milieu] : return recherche_dicho(x, T[:milieu])
  else : return recherche_dicho(x, T[milieu+1:])
```

Quelle complexité ?

$C(n) = 2 + C(\lfloor \frac{n}{2} \rfloor)$ comparaisons (au pire) pour T de taille n

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

Recherche dans un *tableau trié*

```
def recherche_dicho(x, T) : # ATTENTION version trop naïve
  if len(T) == 0 : return False
  milieu = len(T)//2
  if x == T[milieu] : return True
  elif x < T[milieu] : return recherche_dicho(x, T[:milieu])
  else : return recherche_dicho(x, T[milieu+1:])
```

Quelle complexité ?

$C(n) = 2 + C(\lfloor \frac{n}{2} \rfloor)$ comparaisons (au pire) pour `T` de taille n

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

mais *cette implémentation* n'est pas de complexité $\Theta(\log n)$ à cause des recopies de tableaux $\Longrightarrow$ il faut être plus soigneux

Supprimer les doublons dans une liste

sans_doublons(L)

Étant donné une liste L, construire une liste contenant une et une seule occurrence de chaque élément apparaissant dans L

Supprimer les doublons dans une liste

sans_doublons(L)

Étant donné une liste L, construire une liste contenant une et une seule occurrence de chaque élément apparaissant dans L

```
def sans_doublons(L) :
  res = []
  for elt in L :
    if not recherche(elt, res) : res += [elt]
  return res
```

Supprimer les doublons dans une liste

sans_doublons(L)

Étant donné une liste L, construire une liste contenant une et une seule occurrence de chaque élément apparaissant dans L

```
def sans_doublons(L) :
  res = []
  for elt in L :
    if not recherche(elt, res) : res += [elt]
  return res
```

n tours de boucle, le i^e faisant $\Theta(i)$ comparaisons (au pire)

Supprimer les doublons dans une liste

sans_doublons(L)

Étant donné une liste L, construire une liste contenant une et une seule occurrence de chaque élément apparaissant dans L

```
def sans_doublons(L) :
  res = []
  for elt in L :
    if not recherche(elt, res) : res += [elt]
  return res
```

n tours de boucle, le i^e faisant $\Theta(i)$ comparaisons (au pire)

$\Longrightarrow \Theta(n^2)$ comparaisons (au pire)

Supprimer les doublons d'une liste *triée*

`sans_doublons(L)`

Étant donné une liste `L` *triée*, construire une liste contenant une et une seule occurrence de chaque élément apparaissant dans `L`

Supprimer les doublons d'une liste *triée*

sans_doublons(L)

Étant donné une liste L *triée*, construire une liste contenant une et une seule occurrence de chaque élément apparaissant dans L

```
def sans_doublons(L) :
  if len(L) == 0 : return []
  res = [L[0]]
  for elt in L[1:] :
    if elt != res[-1] : res += [elt]
    # res[-1] : dernier élément de res
  return res
```

Supprimer les doublons d'une liste *triée*

sans_doublons(L)

Étant donné une liste L *triée*, construire une liste contenant une et une seule occurrence de chaque élément apparaissant dans L

```
def sans_doublons(L) :
  if len(L) == 0 : return []
  res = [L[0]]
  for elt in L[1:] :
    if elt != res[-1] : res += [elt]
    # res[-1] : dernier élément de res
  return res
```

$\Longrightarrow \Theta(n)$ comparaisons dans tous les cas

Récapitulons…

	liste chaînée		tableau	
	non triée	triée	non trié	trié
minimum/maximum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
test d'appartenance	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
nombre d'occurrences	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
sans doublons	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$
sélection du k^e	$\Theta(kn)$	$\Theta(k)$	$\Theta(kn)$	$\Theta(1)$

Récapitulons...

	liste chaînée		tableau	
	non triée	triée	non trié	trié
minimum/maximum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
test d'appartenance	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
nombre d'occurrences	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
sans doublons	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$
sélection du k^e	$\Theta(kn)$	$\Theta(k)$	$\Theta(kn)$	$\Theta(1)$

Moralité...

peut-être que ça vaut le coup de trier les listes !

`tri(L)`

Étant donné une liste `L` d'éléments comparables, construire la liste des éléments de `L` classés en ordre croissant

Trier une liste

`tri(L)`

Étant donné une liste `L` d'éléments comparables, construire la liste des éléments de `L` classés en ordre croissant

`tri_en_place(L)`

Étant donné une liste `L` d'éléments comparables, réordonner les éléments de `L` en ordre croissant

(sans création de liste supplémentaire)

Tri par sélection

Exemple :

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

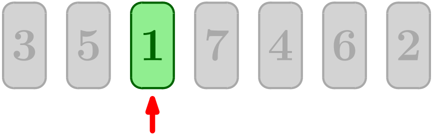
Tri par sélection

Exemple :

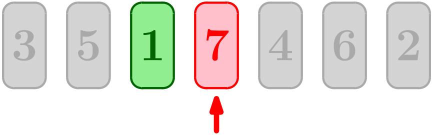
Tri par sélection

Exemple :

Exemple :

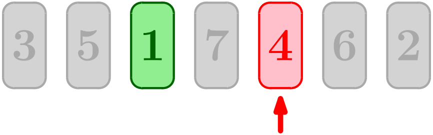


Exemple :



Tri par sélection

Exemple :



Tri par sélection

Exemple :

Tri par sélection

Exemple :

Tri par sélection

Exemple :

3 5 7 4 6 2

1

Tri par sélection

Exemple :

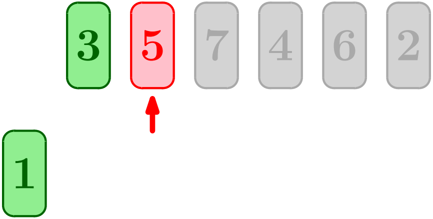
3 5 7 4 6 2

1

Tri par sélection

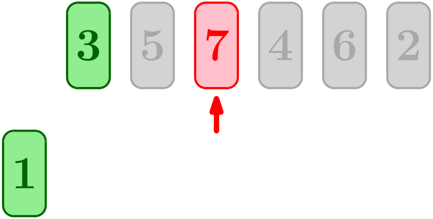
Exemple :

Exemple :

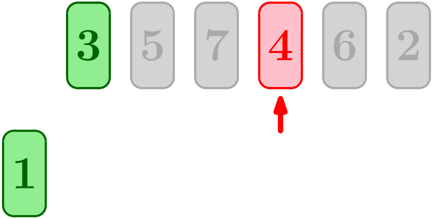


Tri par sélection

Exemple :



Exemple :

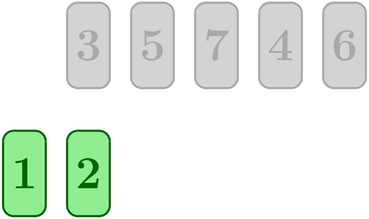


Exemple :

Exemple :

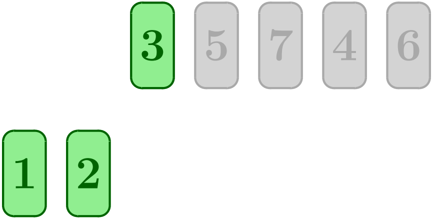
Tri par sélection

Exemple :



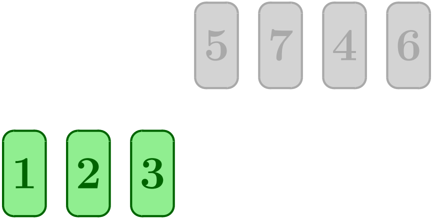
Exemple :

Exemple :



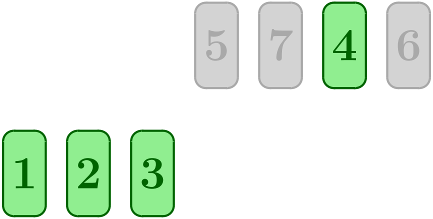
Tri par sélection

Exemple :



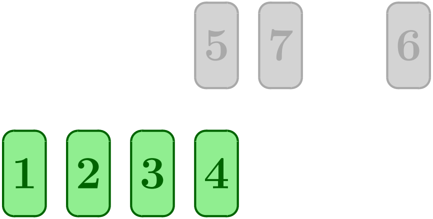
Tri par sélection

Exemple :



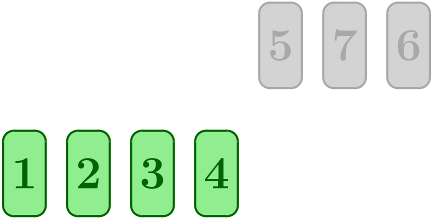
Tri par sélection

Exemple :



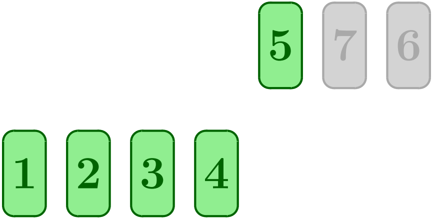
Tri par sélection

Exemple :

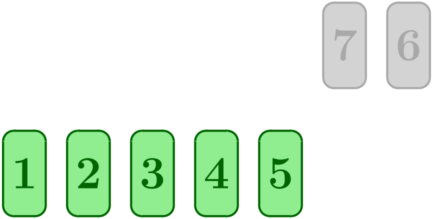


Tri par sélection

Exemple :

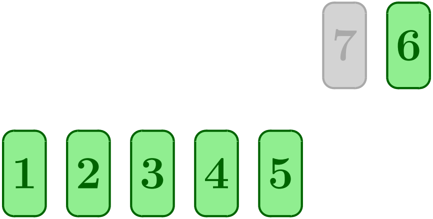


Exemple :



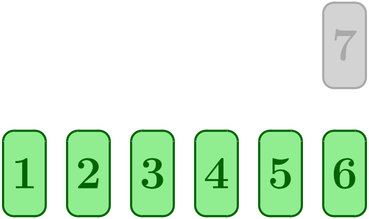
Tri par sélection

Exemple :

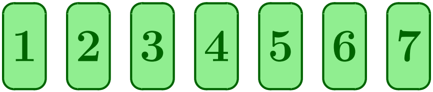


Tri par sélection

Exemple :



Exemple :



tri(L)

Étant donné une liste L d'éléments comparables, construire la liste des éléments de L classés en ordre croissant

```
def tri_selection(L) :
  res = []
  while(L != []) :
    m = minimum(L)
    L.remove(m)
    res.append(m)
  return res
```

tri_en_place(L)

Étant donné une liste L d'éléments comparables, réordonner les éléments de L en ordre croissant

```
def tri_selection(T) :
  for i in range(len(T)) :
    min = indice_minimum(T, i)
      # indice du plus petit élément de T[i:]
    T[i], T[min] = T[min], T[i]
  return T
```

`tri(L)`

Étant donné une liste `L` d'éléments comparables, construire la liste des éléments de `L` classés en ordre croissant

Taille de l'entrée

= longueur de la liste

Opérations élémentaires prises en compte

- comparaisons entre éléments de la liste
- échanges d'éléments de la liste

Tri par insertion

Exemple :

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

```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

Exemple :

3 5 1 7 4 6 2

```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

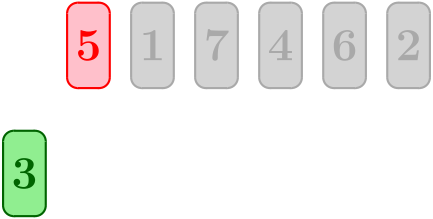
Tri par insertion

Exemple :


```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

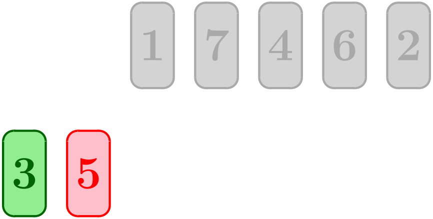
Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

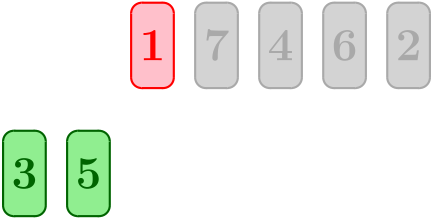
Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

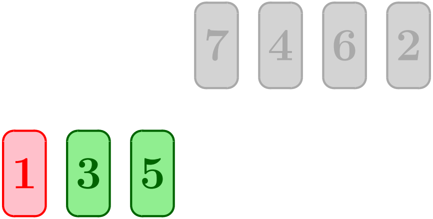
Exemple :




```python
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

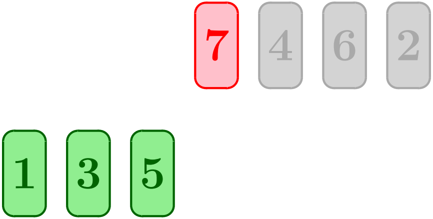
Tri par insertion

Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

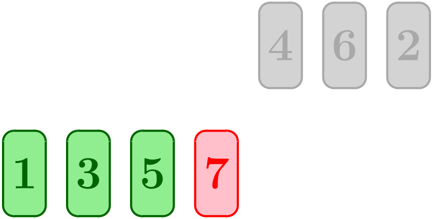
Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

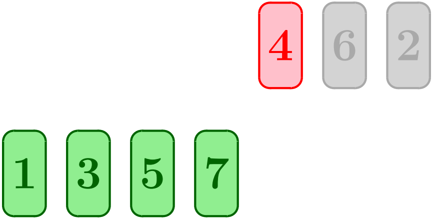
Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

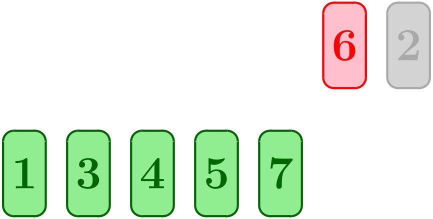
Tri par insertion

Exemple :


```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

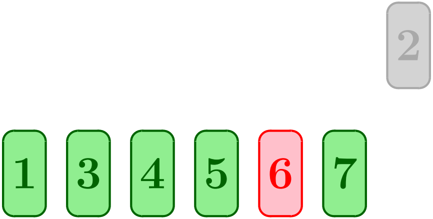
Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

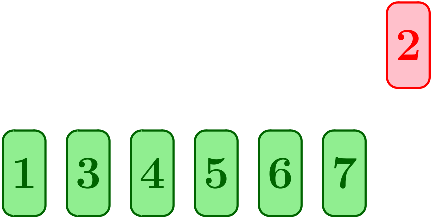
Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

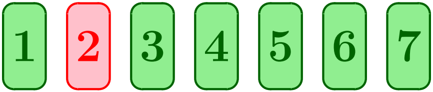
Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Tri par insertion

Exemple :




```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

Exemple :

1 2 3 4 5 6 7

```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res
```

```
def tri_insertion(L) :
  res = []
  for elt in L : insertion_triee(elt, res)
  return res

def insertion_triee(x, L) :
  for elt in L :
    if x < elt : break
  ## insertion de x avant elt dans L
  return res
```

```
def insertion_triee(x, L) :
  for elt in L :
    if x < elt : break
  ## insertion de x avant elt dans L
  return res
```

Cas d'une liste chaînée

insertion par modification du chaînage

Cas d'un tableau

insertion par déplacements multiples

```
def insertion_triee(x, L) :
  for elt in L :
    if x < elt : break
  ## insertion de x avant elt dans L
  return res
```

Cas d'une liste chaînée

insertion par modification du chaînage $\Longrightarrow$ coût constant

Cas d'un tableau

insertion par déplacements multiples $\Longrightarrow$ coût linéaire

Tri par insertion dans un tableau

(ajout par rapport aux slides projetés en cours)

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

(ajout par rapport aux slides projetés en cours)

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

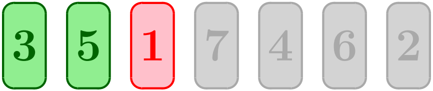
(ajout par rapport aux slides projetés en cours)

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

(ajout par rapport aux slides projetés en cours)




```
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

(ajout par rapport aux slides projetés en cours)


```
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

(ajout par rapport aux slides projetés en cours)


```
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
```

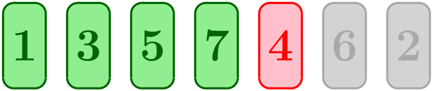
Tri par insertion dans un tableau

(ajout par rapport aux slides projetés en cours)


```
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

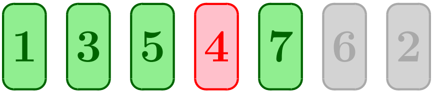
(ajout par rapport aux slides projetés en cours)




```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

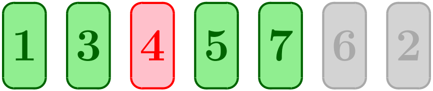
(ajout par rapport aux slides projetés en cours)




```
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

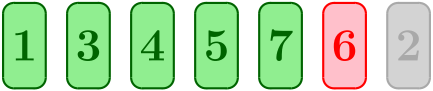
(ajout par rapport aux slides projetés en cours)




```
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

(ajout par rapport aux slides projetés en cours)




```
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

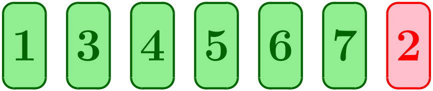
(ajout par rapport aux slides projetés en cours)

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

(ajout par rapport aux slides projetés en cours)




```
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

(ajout par rapport aux slides projetés en cours)

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

(ajout par rapport aux slides projetés en cours)

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

```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

(ajout par rapport aux slides projetés en cours)

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

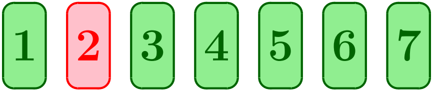
(ajout par rapport aux slides projetés en cours)

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

```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

(ajout par rapport aux slides projetés en cours)




```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

(ajout par rapport aux slides projetés en cours)

1	2	3	4	5	6	7

```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
```

Remarque : pour avoir un « meilleur cas » en $\Theta(n)$, il est important d'effectuer le parcours de droite à gauche – sinon la complexité serait $\Theta(n^2)$ dans tous les cas.

Tri par sélection $\Theta(n^2)$ comparaisons dans tous les cas

Tri par insertion $\Theta(n^2)$ comparaisons au pire

Questions

- peut-on être plus précis pour le tri par insertion ?
- peut-on faire mieux que $\Theta(n^2)$ au pire ?