

Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon

`dominique.poulalhon@irif.fr`

Université de Paris

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

Année universitaire 2021-2022

ALGORITHMES POUR LES ENSEMBLES

RECHERCHES DANS UNE LISTE

`recherche(x, L)`

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

RECHERCHES DANS UNE LISTE

`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)`)

RECHERCHES DANS UNE LISTE

`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
```

RECHERCHES DANS UNE LISTE

`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)`)

RECHERCHES DANS UNE LISTE

`occurrences(x, L)`

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

RECHERCHES DANS UNE LISTE

`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)`)

RECHERCHES DANS UNE LISTE

`max(L)`

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

RECHERCHES DANS UNE LISTE

`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

COMPLEXITÉ DE CES RECHERCHES

opération(s) élémentaire(s)

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

toutes effectuées en nombre équivalent

⇒ pour simplifier, on ne compte que les *comparaisons*

COMPLEXITÉ DE CES RECHERCHES

opération(s) élémentaire(s)

- *comparaisons d'éléments*

$\max(L)$

$\Rightarrow n - 1 = \Theta(n)$ comparaisons

COMPLEXITÉ DE CES RECHERCHES

opération(s) élémentaire(s)

- *comparaisons d'éléments*

`max(L)`

$\Rightarrow n - 1 = \Theta(n)$ comparaisons

`occurrences(x, L)`

$\Rightarrow n = \Theta(n)$ comparaisons

COMPLEXITÉ DE CES RECHERCHES

opération(s) élémentaire(s)

- *comparaisons d'éléments*

`max(L)`

$\Rightarrow n - 1 = \Theta(n)$ comparaisons

`occurrences(x, L)`

$\Rightarrow n = \Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

\Rightarrow selon les cas, entre 1 et n comparaisons

COMPLEXITÉ DE CES RECHERCHES

opération(s) élémentaire(s)

- *comparaisons d'éléments*

`max(L)`

$\Rightarrow n - 1 = \Theta(n)$ comparaisons

`occurrences(x, L)`

$\Rightarrow n = \Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

\Rightarrow selon les cas, entre 1 et n comparaisons

\Rightarrow on ne peut plus parler de « la » complexité

COMPLEXITÉ DE CES RECHERCHES

opération(s) élémentaire(s)

- *comparaisons d'éléments*

`max(L)`

$\Rightarrow n - 1 = \Theta(n)$ comparaisons

`occurrences(x, L)`

$\Rightarrow n = \Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

\Rightarrow selon les cas, entre 1 et n comparaisons

\Rightarrow 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)`

$\Rightarrow n - 1 = \Theta(n)$ comparaisons

`occurrences(x, L)`

$\Rightarrow n = \Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

\Rightarrow selon les cas, entre 1 et n comparaisons

\Rightarrow 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)`

$\Rightarrow n - 1 = \Theta(n)$ comparaisons

`occurrences(x, L)`

$\Rightarrow n = \Theta(n)$ comparaisons

`recherche_sequentielle(x, L)`

\Rightarrow selon les cas, entre 1 et n comparaisons

\Rightarrow 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 $\Theta(n)$?

COMPLEXITÉ DE CES RECHERCHES

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

Demi-réponse : sans hypothèse supplémentaire, non (au moins dans le pire cas) puisque l'élément cherché peut se trouver à n'importe quelle position

RECHERCHE DANS UN tableau trié

$\text{max}(T)$

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]
```

⇒ $\Theta(1)$ 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 n°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 n° 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
```

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

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 n°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

$\Rightarrow \Theta(\log n)$ 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

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

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

RECHERCHE DANS UN tableau trié

pour éviter les recopies, il faut toujours passer le même tableau en paramètre, ainsi que des indices indiquant quelle est la portion à traiter

convention : `debut` inclus, `fin` exclu (comme `range` par exemple)

```
def recherche_dicho(x, T, debut=0, fin=None) :  
    if fin == None : fin = len(T)  
    if fin-debut == 0 : return False  
    milieu = (debut+fin) // 2  
    return True if x == T[milieu]\  
        else recherche_dicho(x, T, debut, milieu) if x < T[milieu]\  
        else recherche_dicho(x, T, milieu+1, fin)
```

(les \ en fin de ligne permettent de poursuivre sur la ligne suivante)

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)

$\Rightarrow \Theta(n^2)$ 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
```

$\Rightarrow \Theta(n)$ 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 !

TRIER UNE LISTE

`tri(L)`

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

TRIÉ 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 :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



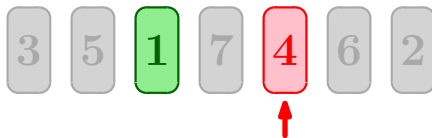
TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



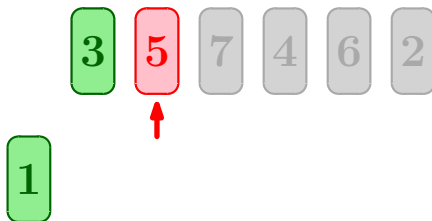
TRI PAR SÉLECTION

Exemple :



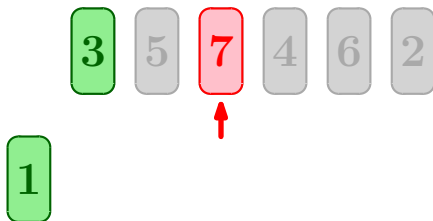
TRI PAR SÉLECTION

Exemple :



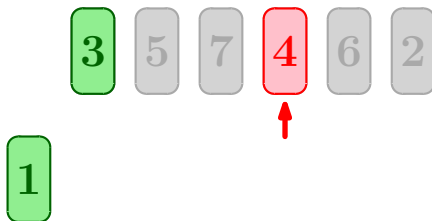
TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



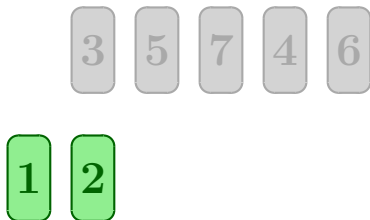
TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



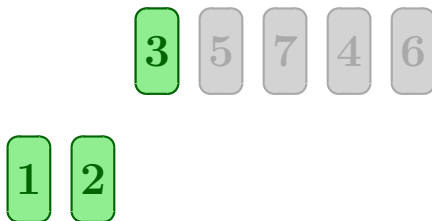
TRI PAR SÉLECTION

Exemple :



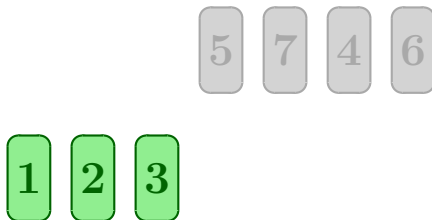
TRI PAR SÉLECTION

Exemple :



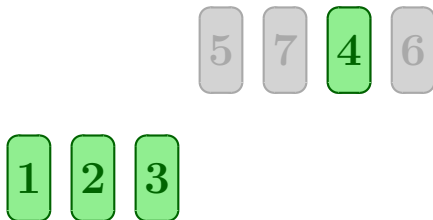
TRI PAR SÉLECTION

Exemple :



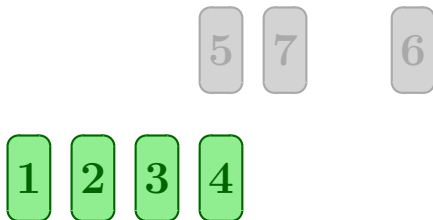
TRI PAR SÉLECTION

Exemple :



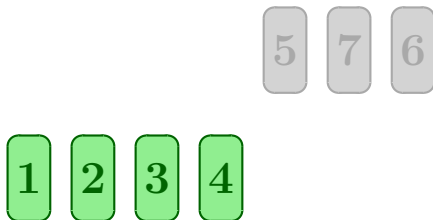
TRI PAR SÉLECTION

Exemple :



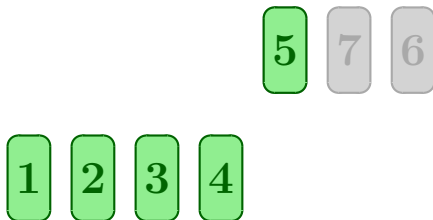
TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



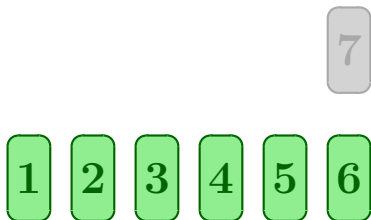
TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

Exemple :



TRI PAR SÉLECTION

`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 PAR SÉLECTION

`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
```

TRIÉ UNE LISTE

`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 :



```
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

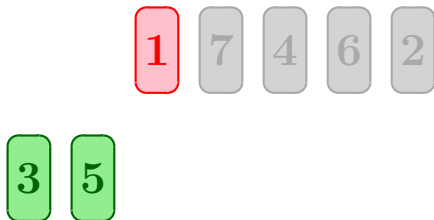
Exemple :



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


TRI PAR INSERTION

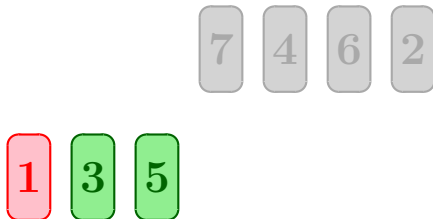
Exemple :



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

TRI PAR INSERTION

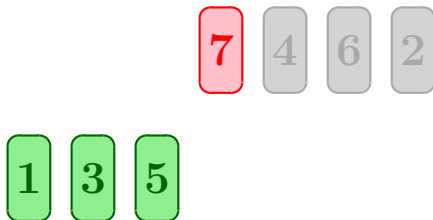
Exemple :



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

TRI PAR INSERTION

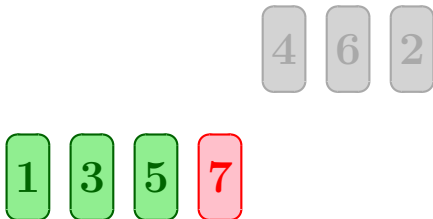
Exemple :



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

TRI PAR INSERTION

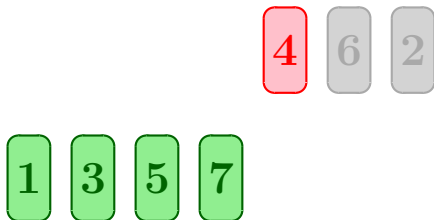
Exemple :



```
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

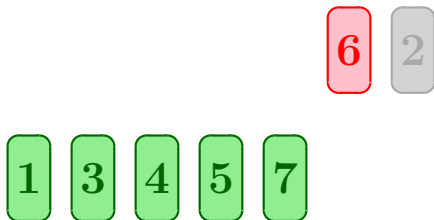
Exemple :



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

TRI PAR INSERTION

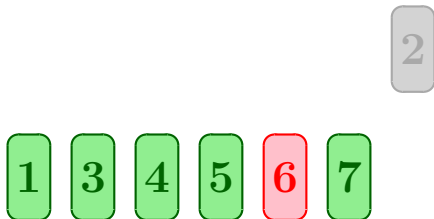
Exemple :



```
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
```

TRI PAR INSERTION

Exemple :



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

TRI PAR INSERTION

```
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
```

TRI PAR INSERTION

```
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

TRI PAR INSERTION

```
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

⇒ *coût constant*

Cas d'un tableau

insertion par déplacements multiples

⇒ *coût linéaire*

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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


TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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


TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



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

TRI PAR INSERTION DANS UN TABLEAU



```
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
```

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.

COMPLEXITÉ

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 ?