Les algorithmes de tris

Chargée de cours: **Lélia Blin**

Email: *lelia.blin@lip6.fr*

Transparents : http://www-npa.lip6.fr/~blin/Enseignements.html

Idée fondamentale

- On considère une collection d'entier placés dans un tableau
- Un opérateur de comparaison (≤,≥, >, <, ...)
- <u>But :</u>
  - Ré-ordonner les valeurs de la façon suivante
  - $T[i] \leq T[i+1] \forall i \in [1..Taillemax]$

Quelques algorithmes de tris

Tris élémentaires

- Tri par insertion
- Tri par sélection
- Tri par permutation

Tris avancés

- Tri Fusion
- Tri rapide

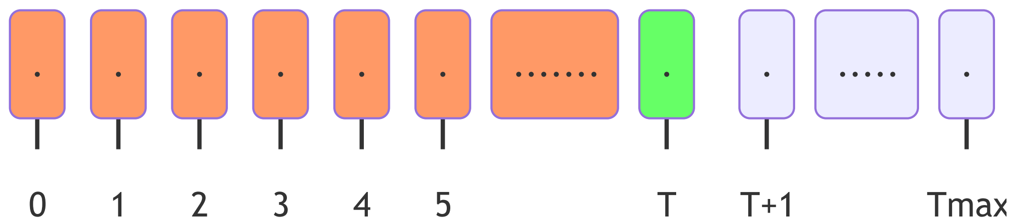
Tris élémentaires

Tri interne, sur place et par comparaison

Principe :

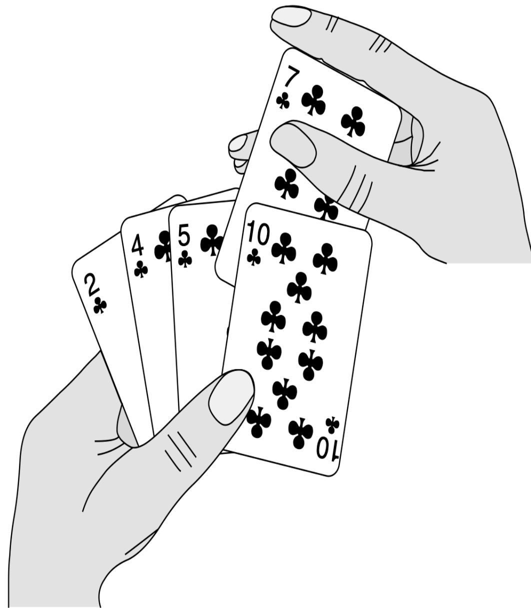
A tout moment, le tableau à trier sera en 2 parties :

1. Une partie triée $[1..T]$: T est la taille courante
2. Une partie non triée $[T+1..TMax]$



Tri par insertion

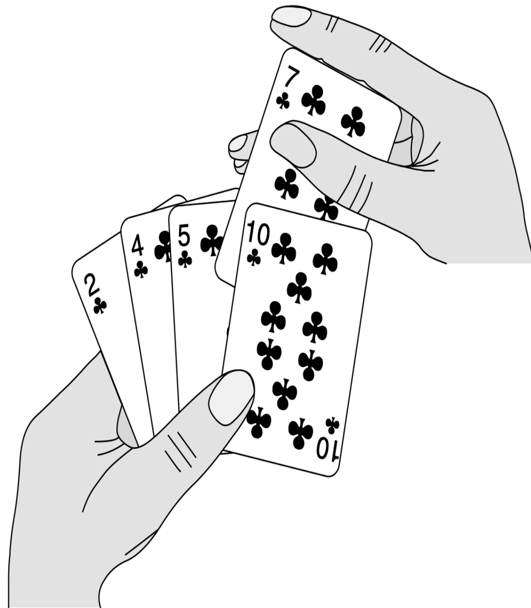
Tri insertion



- C' est un algorithme efficace quand il s'agit de trier un petit nombre d'éléments.
- Le tri par insertion s'inspire de la manière dont la plupart des gens tiennent des cartes à jouer.

Tri insertion

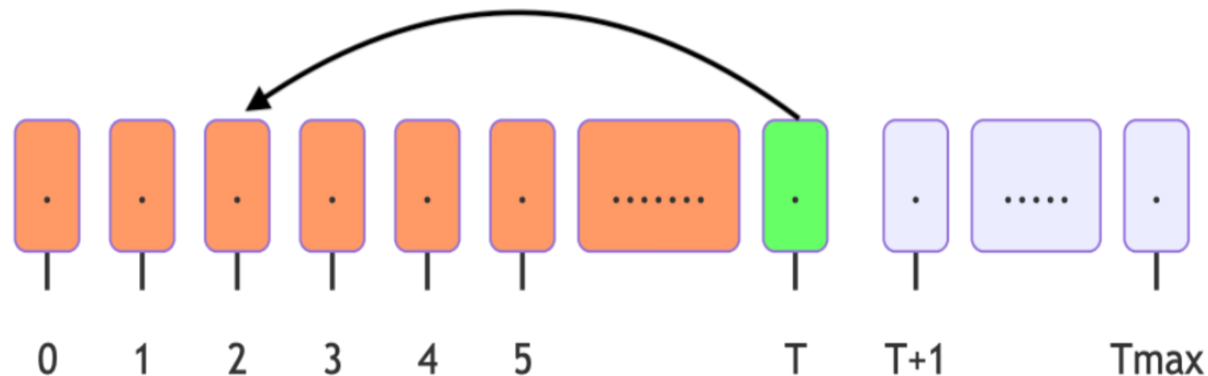
Tri des cartes à jouer:



- Au début, la main gauche du joueur est vide et ses cartes sont posées sur la table.
- Le joueur prend alors sur la table les cartes, une par une, pour les placer dans sa main gauche.
- Pour savoir où placer une carte dans son jeu, le joueur la compare avec chacune des cartes déjà présentes dans sa main gauche, en examinant les cartes de la droite vers la gauche
- A tout moment, les cartes tenues par la main gauche sont triées ;

Tri insertion principe

- Prendre un élément non encore trié
- L'insérer à sa place dans l'ensemble des éléments triés



Algorithme et exemple

Algorithme

```
def tri_insertion(L):
    n =len(L)
    for i in range(1,n):
        cle = L[i]
        j = i-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j]
            j = j-1
        L[j+1] = cle
    return(L)
```

Exemple

Liste initiale

cle: 4 , j: 1

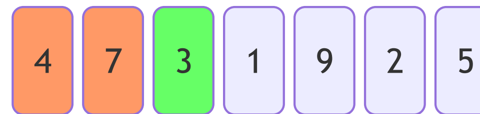
Algorithme et exemple

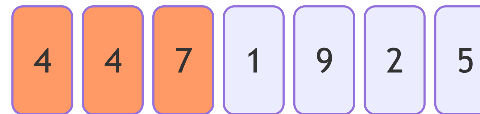
Algorithme

```python
def tri_insertion(L):
    n =len(L)
    for i in range(1,n):
        cle = L[i]
        j = i-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j]
            j = j-1
        L[j+1] = cle
    return(L)
```

Exemple

cle: 3 , j: 2





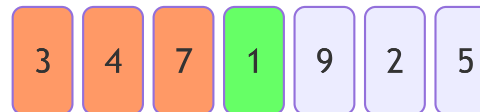
Algorithme et exemple

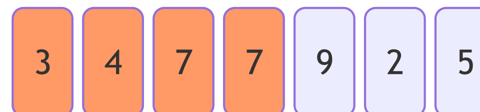
Algorithme

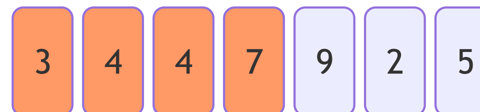
```
def tri_insertion(L):
    n =len(L)
    for i in range(1,n):
        cle = L[i]
        j = i-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j]
            j = j-1
        L[j+1] = cle
    return(L)
```

Exemple

cle: 1 , j: 2







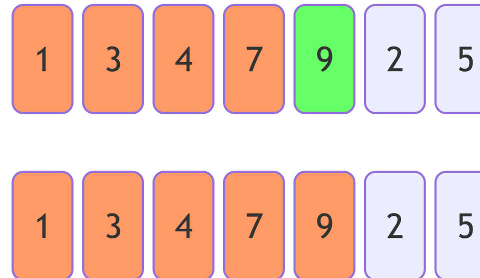
Algorithme et exemple

Algorithme

```
def tri_insertion(L):
    n =len(L)
    for i in range(1,n):
        cle = L[i]
        j = i-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j]
            j = j-1
        L[j+1] = cle
    return(L)
```

Exemple

cle: 9 , j: 3

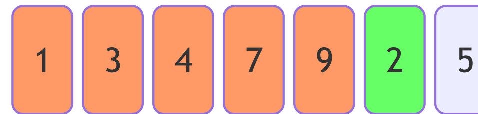


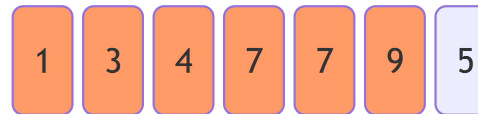
Algorithme et exemple

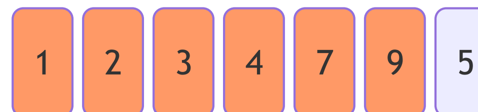
Algorithme

```
def tri_insertion(L):
    n =len(L)
    for i in range(1,n):
        cle = L[i]
        j = i-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j]
            j = j-1
        L[j+1] = cle
    return(L)
```

cle: 2 , j: 4





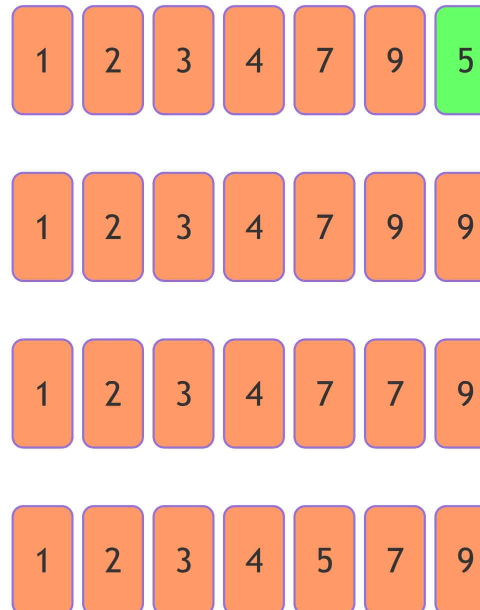


Algorithme et exemple

Algorithme

```
def tri_insertion(L):
    n =len(L)
    for i in range(1,n):
        cle = L[i]
        j = i-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j]
            j = j-1
        L[j+1] = cle
    return(L)
```

cle: 5 , j: 5



Preuve de l'algorithme

Boucle principale: constat

- Au début de chaque itération de la boucle `for`
  - le sous-tableau $L[1..j-1]$ se compose des éléments qui occupaient initialement les positions $L[1..j-1]$.
- Maintenant les éléments de $L[1..j-1]$ sont triés.
- On veut maintenir cet: invariant de boucle

Invariant de boucle

Nous devons montrer trois choses, concernant l'invariant de boucle :

1. Initialisation :
    - Il est vrai avant la première itération de la boucle.
2. Conservation :
    - S'il est vrai avant une itération de la boucle,
    - il le reste avant l'itération suivante.
3. Terminaison :
    - Une fois terminée la boucle, l'invariant fournit une propriété utile qui aide à montrer la validité de l'algorithme.

Invariant de boucle

Nous devons montrer trois choses, concernant l'invariant de boucle :

1. Initialisation :
    - Il est vrai avant la première itération de la boucle.
2. Conservation :
    - S'il est vrai avant une itération de la boucle,
    - il le reste avant l'itération suivante.
3. Terminaison :
    - Une fois terminée la boucle, l'invariant fournit une propriété utile qui aide à montrer la validité de l'algorithme.

- Si les deux premières propriétés sont vérifiées, alors l'invariant est vrai avant chaque itération de la boucle.
- La troisième propriété est utilisé pour prouver la validité de l'algorithme.

Invariant de boucle: Initialisation

Montrons que l'invariant est vérifié avant la première itération de la boucle

- Autrement dit quand $j = 2$.
- Le sous-tableau $L[1..j-1]$ se compose donc uniquement de l'élément $L[1]$
    - En outre, ce sous-tableau est trié (c'est une trivialité),
    - Cela montre que l'invariant est vérifié avant la première itération de la boucle.

Invariant de boucle: Conservation

Montrons que chaque itération conserve l'invariant

- De manière informelle le corps de la boucle `for` fonctionne:
    - en déplaçant $L[j-1], L[j-2], L[j-3]$, etc.
    - d'une position vers la droite
    - jusqu'à ce qu'on trouve la bonne position pour $L[j]$,
    - auquel cas on insère la valeur de $L[j]$.

Invariant de boucle : Terminaison

Examinon se qui se passe à la terminaison de la boucle

- la boucle `for` prend fin quand j dépasse n (c'est-à-dire quand $j = n + 1$).
- Substituons $n + 1$ à j dans la formulation de l'invariant de boucle.
- On obtient le sous-tableau $L[1..n]$ composait des éléments qui appartenaient originellement à $L[1..n]$ mais qui ont été triés depuis lors.
- Or, le sous-tableau $L[1..n]$ n'est autre que le tableau complet !
- Par conséquent, le tableau tout entier est trié, donc l'algorithme est correct.

Complexité temporelle

Algorithme

```
def tri_insertion(L):
    1:n =len(L)
    2:for i in range(1,n):
    3:    cle = L[i]
    4:    j = i-1
    5:    while j>=0 and L[j] > cle:
    6:        L[j+1] = L[j]
    7:        j = j-1
    8:    L[j+1] = cle
    9: return(L)
```

$cout$

$$= l_1 + nl_2(l_3 + l_4 + nl_5(l_6 + l_7) + l_8) + l_9$$

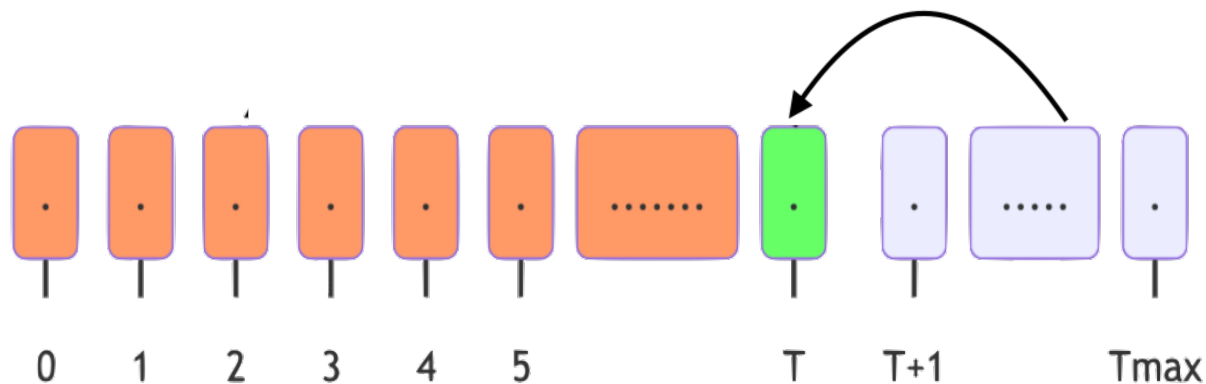
$$= (l_1 + l_9) + n(l_3 + l_4 + l_8) + n^2(l_6 + l_7) \rightarrow O(n^2)$$

Tri par selection

Tri par selection

Principe général :

- Tableau toujours divisé en 2 parties
- A chaque étape,
  - **Choisir le plus petit élément de la partie non triée**
  - **Mettre cet élément à la fin de la partie triée**



Algorithme et exemple

Algorithme

```
def tri_selection(tab):
    for i in range(len(tab)):
        min = i
        for j in range(i+1, len(tab)):
            if tab[min] > tab[j]:
                min = j
        tmp = tab[i]
        tab[i] = tab[min]
        tab[min] = tmp
    return tab
```

7	4	3	1	9	2	5

```
--- i: 0
j: 1 min: 0 tab[ 0 ]= 7 - tab[ 1 ]= 4
min change min= 1
j: 2 min: 1 tab[ 1 ]= 4 - tab[ 2 ]= 3
min change min= 2
j: 3 min: 2 tab[ 2 ]= 3 - tab[ 3 ]= 1
min change min= 3
j: 4 min: 3 tab[ 3 ]= 1 - tab[ 4 ]= 9
j: 5 min: 3 tab[ 3 ]= 1 - tab[ 5 ]= 2
j: 6 min: 3 tab[ 3 ]= 1 - tab[ 6 ]= 5
```

7	4	3	1	9	2	5

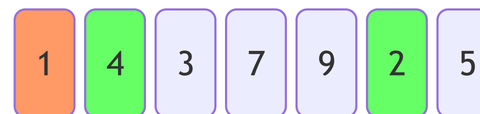
1	4	3	7	9	2	5

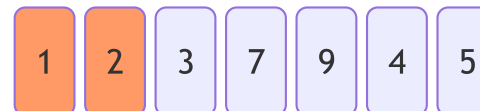
Algorithme et exemple

Algorithme

```
def tri_selection(tab):
   for i in range(len(tab)):
       min = i
       for j in range(i+1, len(tab)):
           if tab[min] > tab[j]:
               min = j
       tmp = tab[i]
       tab[i] = tab[min]
       tab[min] = tmp
   return tab
```

```
--- i: 1
j: 2 min: 1 tab[ 1 ]= 4 - tab[ 2 ]= 3
min change min= 2
j: 3 min: 2 tab[ 2 ]= 3 - tab[ 3 ]= 7
j: 4 min: 2 tab[ 2 ]= 3 - tab[ 4 ]= 9
j: 5 min: 2 tab[ 2 ]= 3 - tab[ 5 ]= 2
min change min= 5
j: 6 min: 5 tab[ 5 ]= 2 - tab[ 6 ]= 5
```

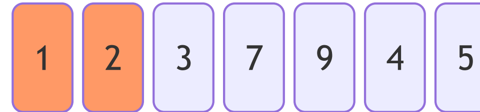




Algorithme et exemple

Algorithme

```
def tri_selection(tab):
    for i in range(len(tab)):
        min = i
        for j in range(i+1, len(tab)):
            if tab[min] > tab[j]:
                min = j
        tmp = tab[i]
        tab[i] = tab[min]
        tab[min] = tmp
    return tab
```

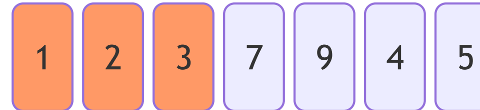



```
--- i: 2
j: 3 min: 2 tab[ 2 ]= 3 - tab[ 3 ]= 7
j: 4 min: 2 tab[ 2 ]= 3 - tab[ 4 ]= 9
j: 5 min: 2 tab[ 2 ]= 3 - tab[ 5 ]= 4
j: 6 min: 2 tab[ 2 ]= 3 - tab[ 6 ]= 5
```

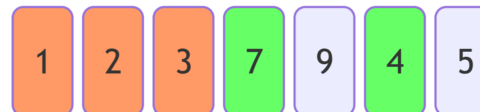
Algorithme et exemple

Algorithme

```
def tri_selection(tab):
   for i in range(len(tab)):
       min = i
       for j in range(i+1, len(tab)):
           if tab[min] > tab[j]:
               min = j
       tmp = tab[i]
       tab[i] = tab[min]
       tab[min] = tmp
   return tab
```

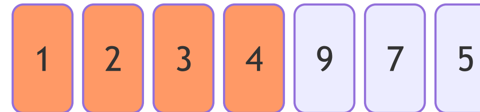



```
--- i: 3
j: 4 min: 3 tab[ 3 ]= 7 - tab[ 4 ]= 9
j: 5 min: 3 tab[ 3 ]= 7 - tab[ 5 ]= 4
min change min= 5
j: 6 min: 5 tab[ 5 ]= 4 - tab[ 6 ]= 5
```



Algorithme et exemple

Algorithme



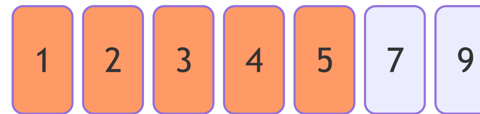

```
def tri_selection(tab):
    for i in range(len(tab)):
        min = i
        for j in range(i+1, len(tab)):
            if tab[min] > tab[j]:
                min = j
        tmp = tab[i]
        tab[i] = tab[min]
        tab[min] = tmp
    return tab
```

```
--- i: 4
j: 5 min: 4 tab[ 4 ]= 9 - tab[ 5 ]= 7
min change min= 5
j: 6 min: 5 tab[ 5 ]= 7 - tab[ 6 ]= 5
min change min= 6
```

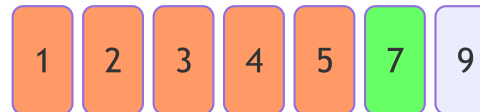
Algorithme et exemple

Algorithme

```
def tri_selection(tab):
    for i in range(len(tab)):
        min = i
        for j in range(i+1, len(tab)):
            if tab[min] > tab[j]:
                min = j
        tmp = tab[i]
        tab[i] = tab[min]
        tab[min] = tmp
    return tab
```



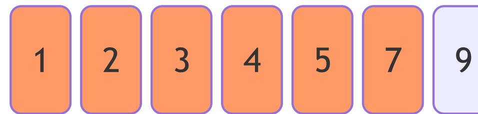

```
--- i: 5
j: 6 min: 5 tab[ 5 ]= 7 - tab[ 6 ]= 9
```



Algorithme et exemple

Algorithme

```
def tri_selection(tab):
    for i in range(len(tab)):
        min = i
        for j in range(i+1, len(tab)):
            if tab[min] > tab[j]:
                min = j
        tmp = tab[i]
        tab[i] = tab[min]
        tab[min] = tmp
    return tab
```




```
--- i: 6
```

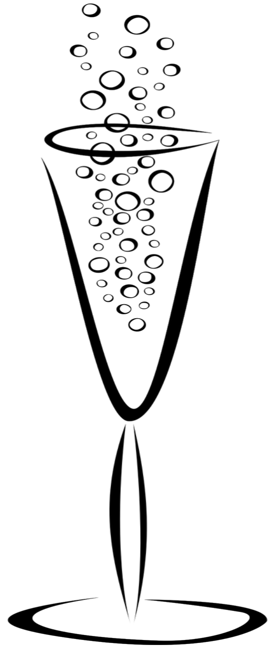
Complexité du tri par selection

- Nombre d'itérations : $n - 1$
- A chaque itération :
  - **Recherche du minimum : $n - T$ (T taille courante)**
  - **Mettre l'élément à sa place : 3**
- Au total : $3n + n(n-1)/2$
- Complexité : $O(n^2)$

Tri par permutation

Tri à bulles

Tri par permutation



Pincipe:

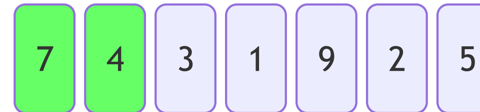
- Si 2 éléments voisins ne sont pas ordonnés on les échange
- Deux parties dans le tableau :
  - Les éléments de la partie triée
  - sont inférieurs
  - aux éléments de la partie non triée.

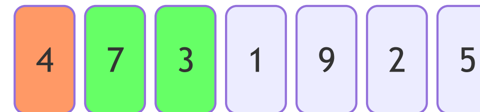
Algorithme et exemple

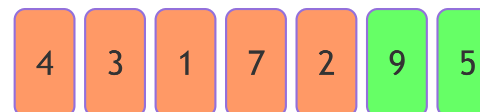
Algorithme

```
def tri_bulle(tab):
    n = len(tab)
    for i in range(0,n):
        for j in range(n-i-1):
            if tab[j] > tab[j+1] :
                tmp=tab[j+1]
                tab[j+1]=tab[j]
                tab[j] = tmp
    return(tab)
```

```
---i: 0
j: 0 tab[ 0 ]= 7  - tab[ 1 ]= 4 <->
j: 1 tab[ 1 ]= 7  - tab[ 2 ]= 3 <->
j: 2 tab[ 2 ]= 7  - tab[ 3 ]= 1 <->
j: 3 tab[ 3 ]= 7  - tab[ 4 ]= 9
j: 4 tab[ 4 ]= 9  - tab[ 5 ]= 2 <->
j: 5 tab[ 5 ]= 9  - tab[ 6 ]= 5 <->
```





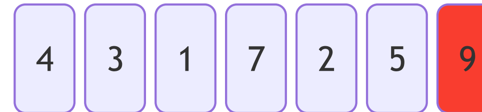


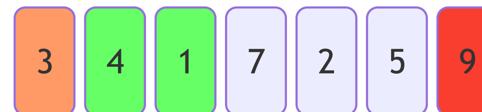
Algorithme et exemple

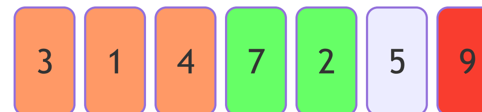
Algorithme

```
def tri_bulle(tab):
    n = len(tab)
    for i in range(0,n):
        for j in range(n-i-1):
            if tab[j] > tab[j+1] :
                tmp=tab[j+1]
                tab[j+1]=tab[j]
                tab[j] = tmp
    return(tab)
```

```
---i: 1
j: 0 tab[ 0 ]= 4  - tab[ 1 ]= 3 <->
j: 1 tab[ 1 ]= 4  - tab[ 2 ]= 1 <->
j: 2 tab[ 2 ]= 4  - tab[ 3 ]= 7
j: 3 tab[ 3 ]= 7  - tab[ 4 ]= 2 <->
j: 4 tab[ 4 ]= 7  - tab[ 5 ]= 5 <->
```





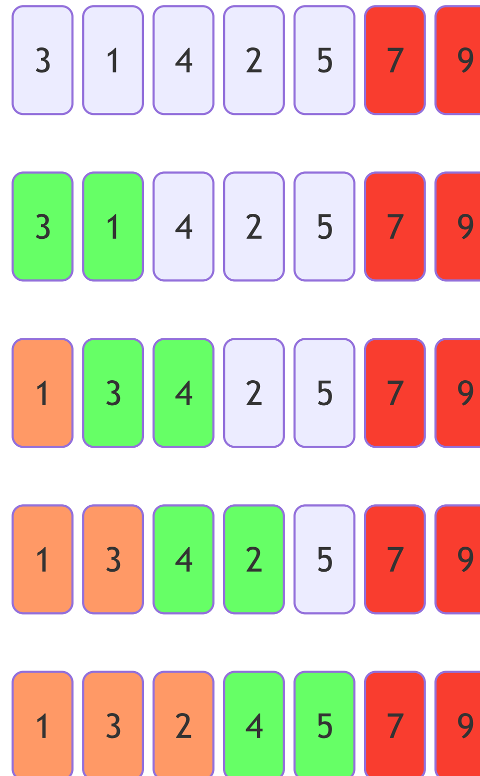


Algorithme et exemple

Algorithme

```
def tri_bulle(tab):
    n = len(tab)
    for i in range(0,n):
        for j in range(n-i-1):
            if tab[j] > tab[j+1] :
                tmp=tab[j+1]
                tab[j+1]=tab[j]
                tab[j] = tmp
    return(tab)
```

```
---i: 2
j: 0 tab[ 0 ]= 3  - tab[ 1 ]= 1 <->
j: 1 tab[ 1 ]= 3  - tab[ 2 ]= 4
j: 2 tab[ 2 ]= 4  - tab[ 3 ]= 2 <->
j: 3 tab[ 3 ]= 4  - tab[ 4 ]= 5
```

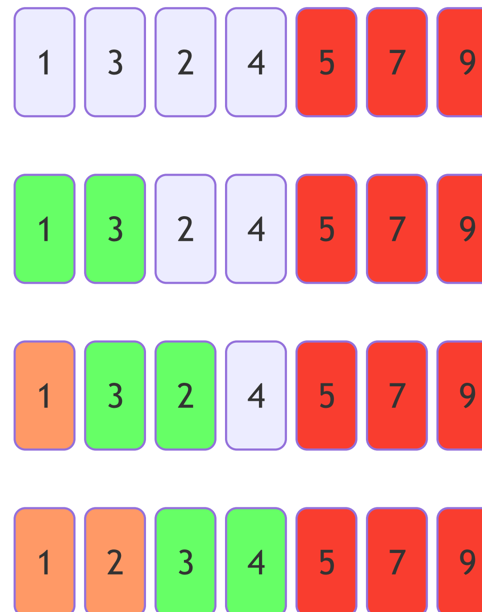


Algorithme et exemple

Algorithme

```
def tri_bulle(tab):
    n = len(tab)
    for i in range(0,n):
        for j in range(n-i-1):
            if tab[j] > tab[j+1] :
                tmp=tab[j+1]
                tab[j+1]=tab[j]
                tab[j] = tmp
    return(tab)
```

```
---i: 3
j: 0 tab[ 0 ]= 1  - tab[ 1 ]= 3
j: 1 tab[ 1 ]= 3  - tab[ 2 ]= 2 <->
j: 2 tab[ 2 ]= 3  - tab[ 3 ]= 4
```

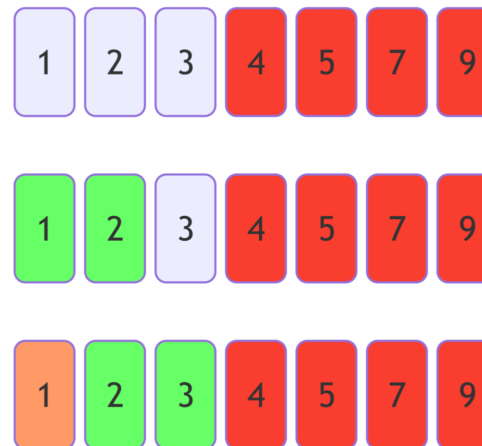


Algorithme et exemple

Algorithme

```
def tri_bulle(tab):
    n = len(tab)
    for i in range(0,n):
        for j in range(n-i-1):
            if tab[j] > tab[j+1] :
                tmp=tab[j+1]
                tab[j+1]=tab[j]
                tab[j] = tmp
    return(tab)
```

```
---i: 4
j: 0 tab[ 0 ]= 1  - tab[ 1 ]= 2
j: 1 tab[ 1 ]= 2  - tab[ 2 ]= 3
```

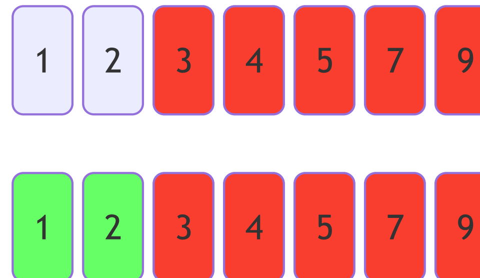


Algorithme et exemple

Algorithme

```
def tri_bulle(tab):
    n = len(tab)
    for i in range(0,n):
        for j in range(n-i-1):
            if tab[j] > tab[j+1] :
                tmp=tab[j+1]
                tab[j+1]=tab[j]
                tab[j] = tmp
    return(tab)
```

```
---i: 5
j: 0 tab[ 0 ]= 1  - tab[ 1 ]= 2
```



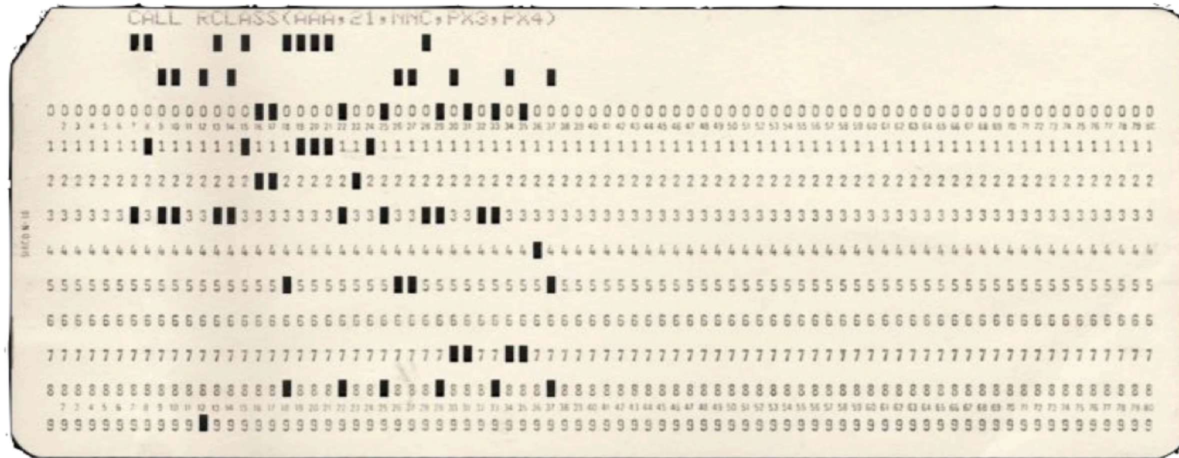
Complexité tri permutation

- Boucle externe : $n-2$ fois
- Boucle interne : $n-T$ fois
- Total : $(n-1)(n-2)/2$
- Complexité $O(n^2)$

Les tris avancés

Le tri fusion

- Machine à trier des cartes perforées en 1938
- 1er algo de tri fusion écrit par Von Neumann pour l'EDVAC en 1945
- Basé sur le paradigme « Diviser pour Régner »



John von Neumann (1903-1957)

Mathématicien et physicien américano-hongrois.

- Importantes contributions:
  - mécanique quantique, analyse fonctionnelle,sciences économiques,
  - théorie des ensembles, informatique,
- Il a de plus participé aux programmes militaires américains.
  - Architecture de Von Neuman:
    - possède une unique mémoire qui sert à conserver les logiciels et les données.
    - utilisée dans la quasi totalité des ordinateurs modernes.

Alan Mathison Turing (1912-1954)

Mathématicien britannique,

- Auteur de l'article fondateur de la science informatique:
  - La machine de Turing et
  - les concepts modernes de programmation et de programme
  - Création des calculateurs universels programmables: les ordinateurs.
- Pere de l'informatique : Il est également à l'origine:
  - de la formalisation des concepts d'algorithme et
  - de calculabilité qui ont profondément marqué cette discipline.
  - Thèse de Church-Turing : Son modèle a contribué à établir définitivement la thèse Church-Turing qui donne une définition mathématique au concept intuitif de fonction calculable.

Alan Mathison Turing (1912-1954)

- Durant la Seconde Guerre mondiale:
  - Il joue un rôle majeur dans les recherches sur les cryptographies générées par la machine Enigma utilisée par les nazis.
- Après la guerre:
  - il travaille sur un des tout premiers ordinateurs puis contribue au débat déjà houleux à cette période sur la capacité des machines à penser en établissant le test de Turing.
  - En 1952 un fait divers indirectement lié à son homosexualité lui vaut des poursuites judiciaires. Pour éviter la prison, il choisit la castration chimique par prise d'œstrogènes. Il se suicide par empoisonnement au cyanure le 7 juin 1954.

Alan Mathison Turing (1912-1954)

- Réabilitation en 2009
  - Pétition: « Nous soussignés demandons au premier ministre de s'excuser pour les poursuites engagées contre Alan Turing qui ont abouti à sa mort prématurée», dressée à l'initiative de l'informaticien John Graham-Cumming a été envoyée à Gordon Brown.
  - En septembre 2009, le Premier ministre britannique a présenté des regrets au nom du gouvernement britannique pour le traitement qui lui a été infligé

PRIX TURING

- Depuis 1966 le prix Turing est annuellement décerné par l'Association for Computing Machinery à des personnes ayant apporté une contribution scientifique significative à la science de l'informatique.

- Cette récompense est souvent considérée comme l'équivalent du prix Nobel de l'informatique.

Diviser pour Régner

Séparer le problème en plusieurs sous-problèmes similaires au problème initial

1. Diviser : le pb en un certain nombre de sous-pb
2. Régner : sur les sous-pbs en les résolvant
3. Combiner : les solutions des sous-pbs en une solution unique.

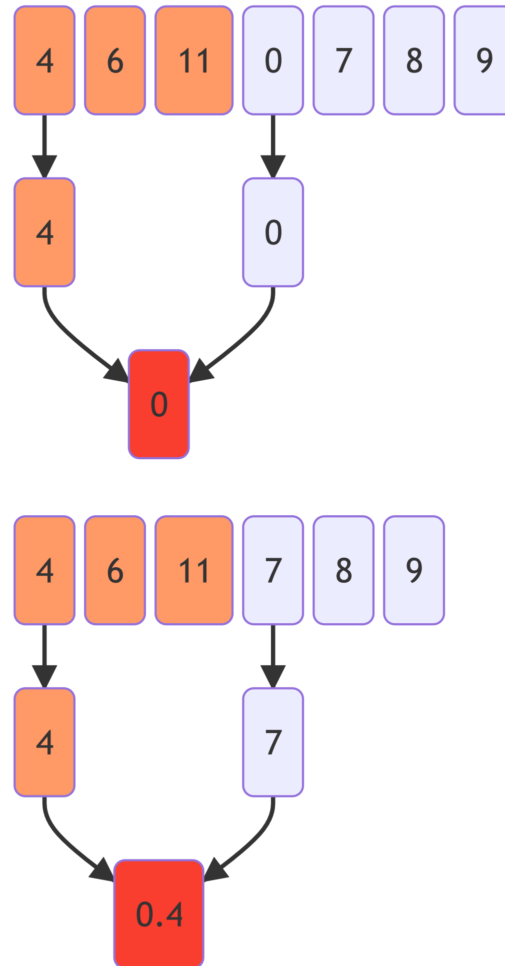
Le tri par fusion : Diviser pour Régner

1. Diviser :
   - la séquence de n éléments à trier en 2 sous-séquences de $n/2$ éléments, jusqu'à que chaque sous-séquence soit de taille 1
2. Régner :
   - Trier les 2 sous-séquences récursivement à l'aide du tri fusion
3. Combiner :
   - Fusionner les 2 sous-séquences triées pour produire la séquence triée.

Fusion

Algorithme

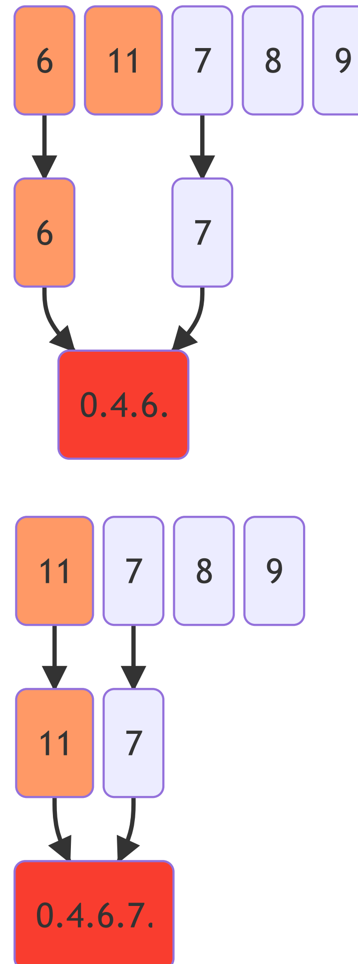
```
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0; i2 = 0; i = 0
    while i1<n1 and i2<n2:
        if L1[i1] < L2[i2]:
            L12[i] = L1[i1]
            i1 += 1
        else:
            L12[i] = L2[i2]
            i2 += 1
        i += 1
    while i1<n1:
        L12[i] = L1[i1]
        i1 += 1;i += 1
    while i2<n2:
        L12[i] = L2[i2]
        i2 += 1; i += 1
    return(L12)
```



Fusion

Algorithme

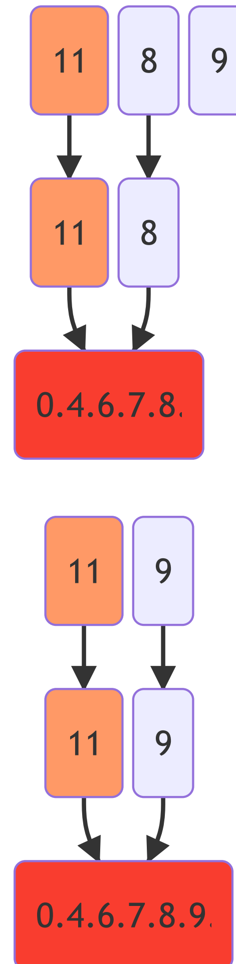
```
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0; i2 = 0; i = 0
    while i1<n1 and i2<n2:
        if L1[i1] < L2[i2]:
            L12[i] = L1[i1]
            i1 += 1
        else:
            L12[i] = L2[i2]
            i2 += 1
        i += 1
    while i1<n1:
        L12[i] = L1[i1]
        i1 += 1;i += 1
    while i2<n2:
        L12[i] = L2[i2]
        i2 += 1; i += 1
    return(L12)
```



Fusion

Algorithme

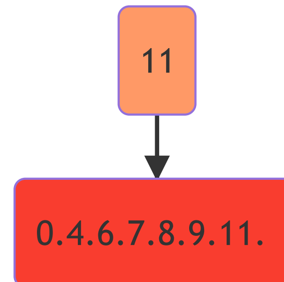
```
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0; i2 = 0; i = 0
    while i1<n1 and i2<n2:
        if L1[i1] < L2[i2]:
            L12[i] = L1[i1]
            i1 += 1
        else:
            L12[i] = L2[i2]
            i2 += 1
        i += 1
    while i1<n1:
        L12[i] = L1[i1]
        i1 += 1;i += 1
    while i2<n2:
        L12[i] = L2[i2]
        i2 += 1; i += 1
    return(L12)
```



Fusion

Algorithme

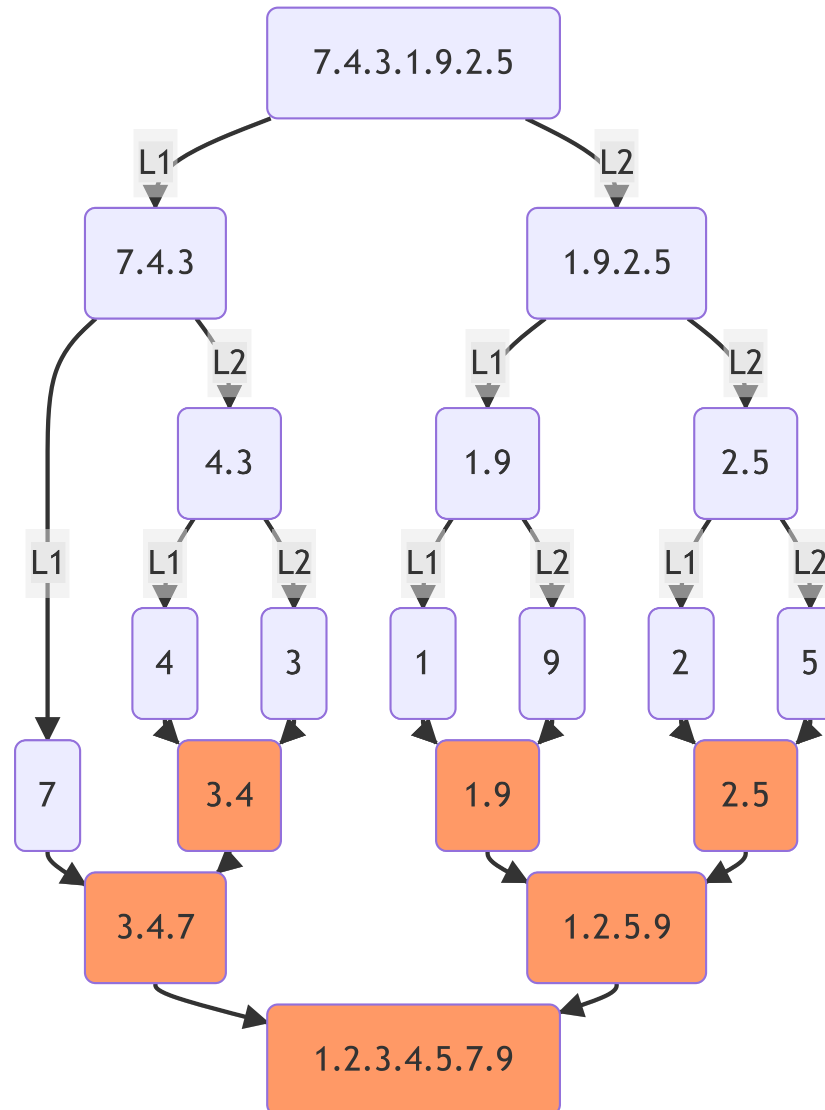
```
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0; i2 = 0; i = 0
    while i1<n1 and i2<n2:
        if L1[i1] < L2[i2]:
            L12[i] = L1[i1]
            i1 += 1
        else:
            L12[i] = L2[i2]
            i2 += 1
        i += 1
    while i1<n1:
        L12[i] = L1[i1]
        i1 += 1;i += 1
    while i2<n2:
        L12[i] = L2[i2]
        i2 += 1; i += 1
    return(L12)
```



Tri fusion

Algorithme

```
def tri_fusion(L):
    n = len(L)
    if n > 1:
        p = n // 2
        L1 = L[:p]
        L2 = L[p:n]
        tri_fusion(L1)
        tri_fusion(L2)
        L[:] = fusion(L1,L2)
    return(L)
```

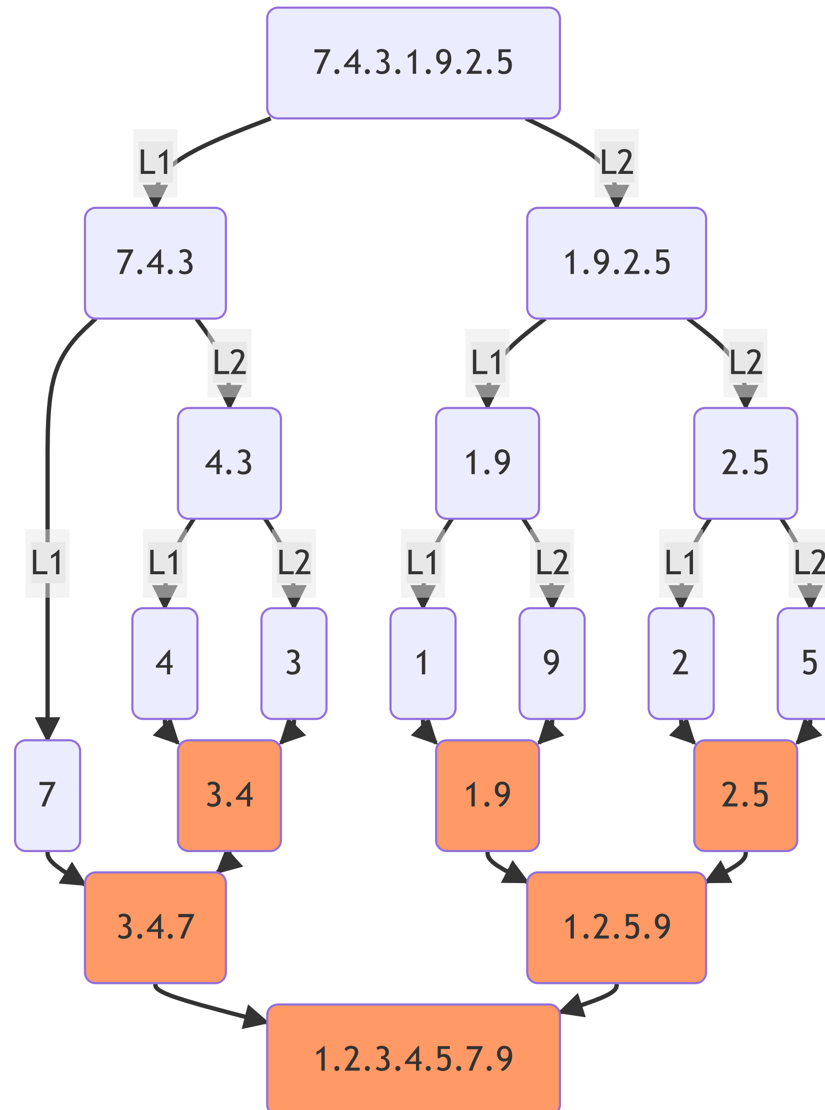


Tri fusion

Algorithme

```
def tri_fusion(L):
    n = len(L)
    if n > 1:
        p = n // 2
        L1 = L[:p]
        L2 = L[p:n]
        tri_fusion(L1)
        tri_fusion(L2)
        L[:] = fusion(L1,L2)
    return(L)
```

Dans quel ordre sont fait les calculs par l'ordinateur?



Complexité du tri par fusion

La preuve est technique mais intuitivement il faut résoudre :

- $Tri(n) = 2 * Tri(n/2) + \Theta(n)$
- **Complexité finale** : $O(n \log_2 n)$

Le tri rapide : Diviser pour Régner

Proposé par Hoare en 1962

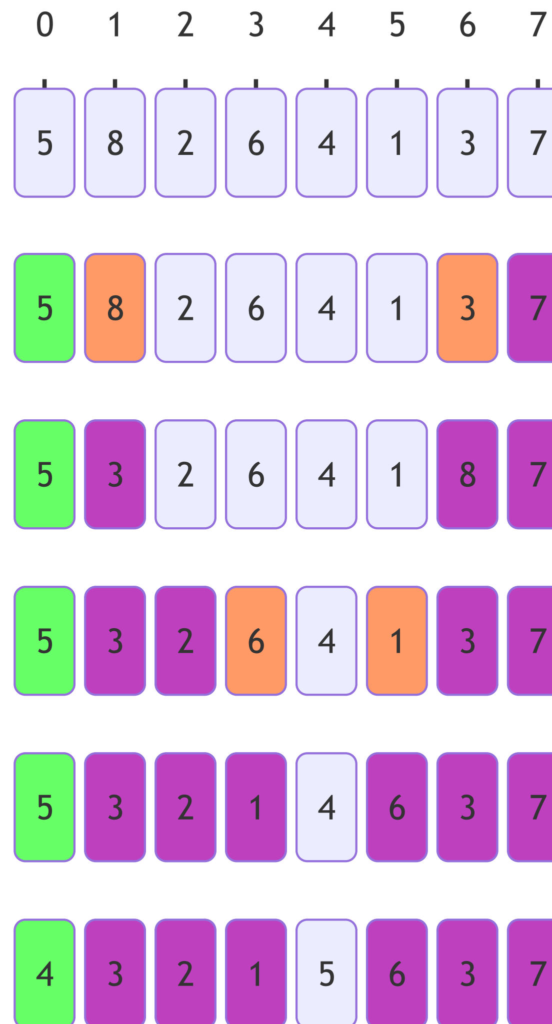
1. **Diviser** :
    - $T[p..r]$ est divisé en 2 sous-tableaux non vide $T[p..q]$ et $T[q+1..r]$
    - $\forall i \in T[p..q]$ et $\forall j \in T[q+1..r]$ on a $i < j$
    - fonction Partitionner
2. **Régner** :
    - 2 sous-tableaux triés grâce à la récursivité
    - fonction TriRapide
3. **Combiner** :
    - 2 sous-tableaux triés sur place : Rien à faire

Tri rapide

```
def partition(L,p,q):
    pivot = L[p];i = p;j = q
    while j>i:
        if L[j] < pivot:
            if L[i]> pivot:
                L[i],L[j] = L[j],L[i]
            i += 1
        else:
            j -= 1
    L[p],L[j] = L[j],L[p]
    return(i,L)
```

```
def tri_partition(L,debut,fin):
    if debut < fin:
        R= partition(L,debut,fin)
        i=R[0];L=R[1]
        tri_partition(L,debut,i-1)
        tri_partition(L,i+1,fin)
```

```
tri_partition(L,0,len(L)-1)
```



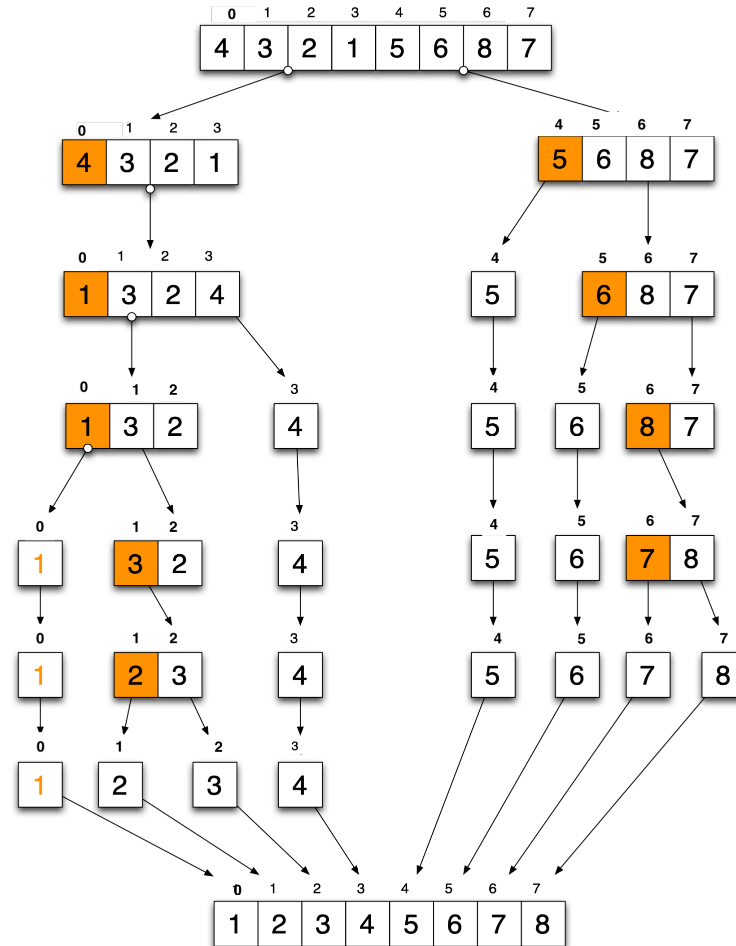

```
partition [5, 8, 2, 6, 4, 1, 3, 7] 0 7
```

Tri rapide

```
def partition(L,p,q):
    pivot = L[p];i = p;j = q
    while j>i:
        if L[j] < pivot:
            if L[i]> pivot:
                L[i],L[j] = L[j],L[i]
            i += 1
        else:
            j -= 1
    L[p],L[j] = L[j],L[p]
    return(i,L)
```

```
def tri_partition(L,debut,fin):
    if debut < fin:
        R= partition(L,debut,fin)
        i=R[0];L=R[1]
        tri_partition(L,debut,i-1)
        tri_partition(L,i+1,fin)
```

```
tri_partition(L,0,len(L)-1)
```



Tri Rapide : Complexité

- Dépend de l'équilibre ou non du partitionnement.
- Si le partitionnement est équilibré :
  - aussi rapide que le tri fusion
- Si le partitionnement est déséquilibré :
  - aussi lent que le tri par insertion

Partitionnement dans le pire cas

- 2 sous-tableaux de :
  - 1 élément
  - $n-1$ éléments

- Supposons que ce partitionnement intervienne à chaque étape.

  - Le partitionnement coûte $\Theta(n)$
  - La récurrence :
    - $T(n) = T(n-1) + \Theta(n)$
    - $T(1) = \Theta(1)$
    - $T(n) = \Sigma O(k) = O(\Sigma k) = O(n^2)$

- Ce partitionnement apparaît quand le tableau est trié !!!!

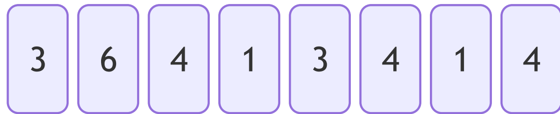
- Pire dans ce cas là le tri par insertion est linéaire !!

Tris par comparaisons

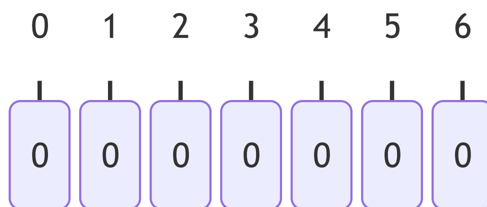
- Tous les tris vu dans ce cours sont tris par comparaisons
- un tri par comparaison est un tri dans lequel on compare une paire d'éléments.
- Existe-t-il des tris qui ne sont pas par comparaisons?

Tri linéaire: tri par dénombrement

- **soit A un tableau d'entier inférieur ou égal à 6**



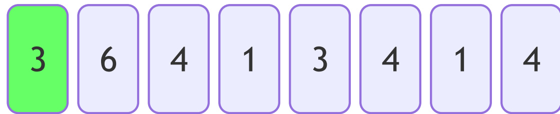
- **soit C un tableau qui compte le nombre de fois qu'apparaît un entier**



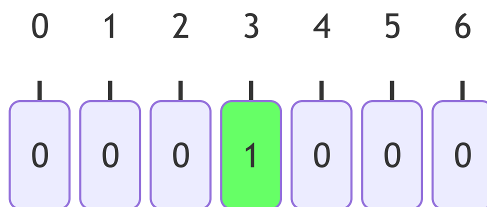
- **soit B le tableau trié**

Tri linéaire: tri par dénombrement

- **soit A un tableau d'entier inférieur ou égal à 6**



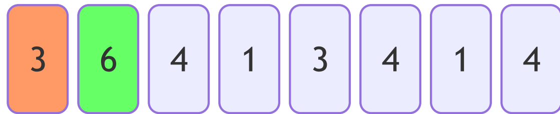
- **soit C un tableau qui compte le nombre de fois qu'apparaît un entier**



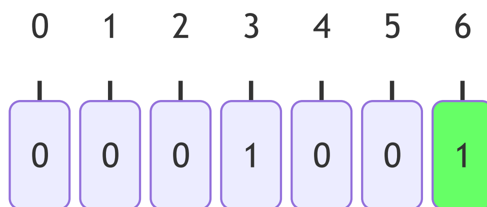
- **soit B le tableau trié**

Tri linéaire: tri par dénombrement

- **soit A un tableau d'entier inférieur ou égal à 6**



- **soit C un tableau qui compte le nombre de fois qu'apparaît un entier**

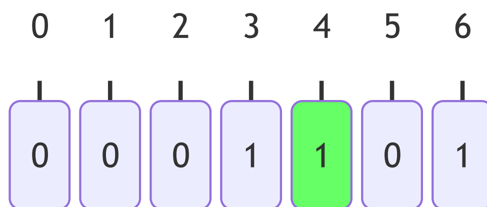


- **soit B le tableau trié**

Tri linéaire: tri par dénombrement

- **soit A un tableau d'entier inférieur ou égal à 6**

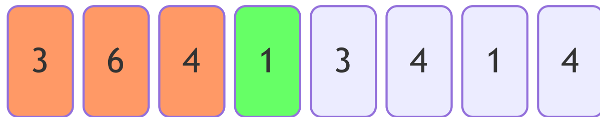
- **soit C un tableau qui compte le nombre de fois qu'apparaît un entier**



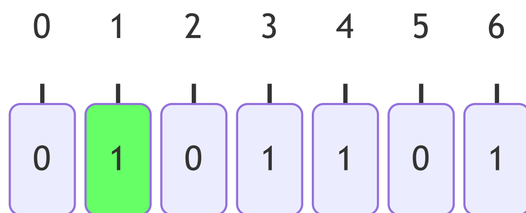
- **soit B le tableau trié**

Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6



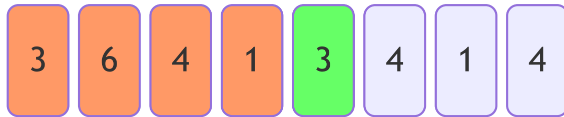
- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



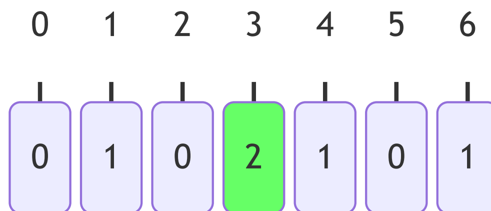
- soit B le tableau trié

Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6



- soit C un tableau qui compte le nombre de fois qu'apparaît un entier

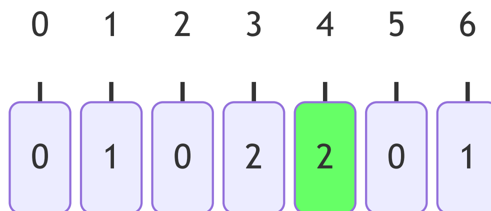


- soit B le tableau trié

Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6

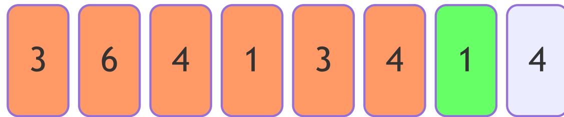
- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



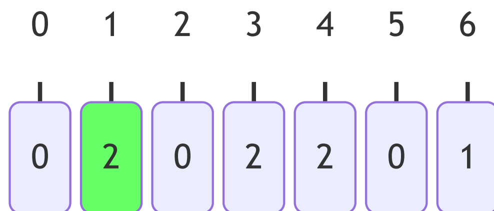
- soit B le tableau trié

Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6



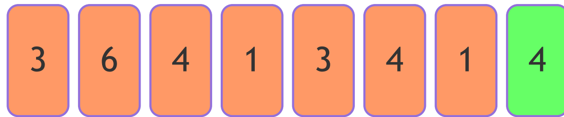
- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



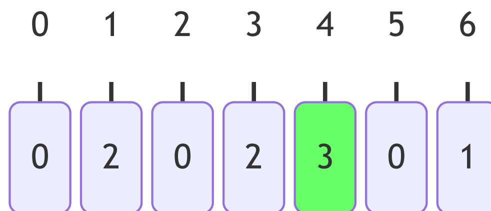
- soit B le tableau trié

Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6



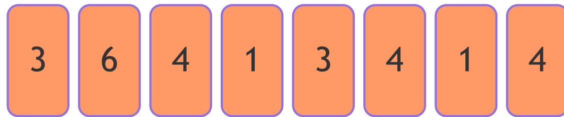
- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



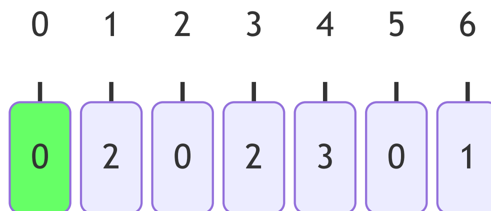
- soit B le tableau trié

Tri linéaire: tri par dénombrement

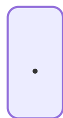
- soit A un tableau d'entier inférieur ou égal à 6



- soit C un tableau qui compte le nombre de fois qu'apparaît un entier

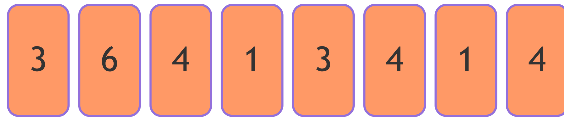


- soit B le tableau trié

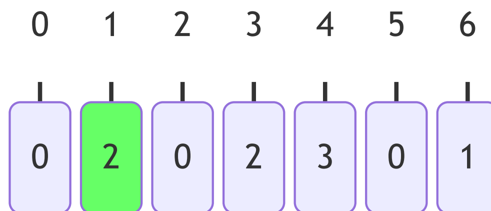


Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6



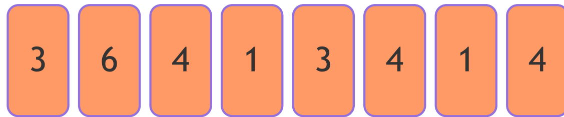
- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



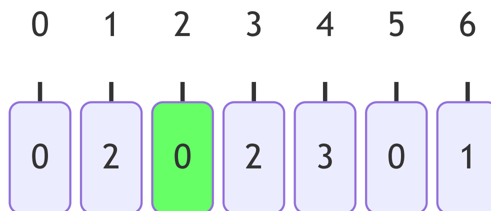
- soit B le tableau trié

Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6



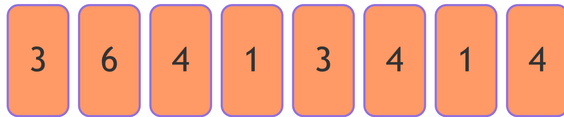
- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



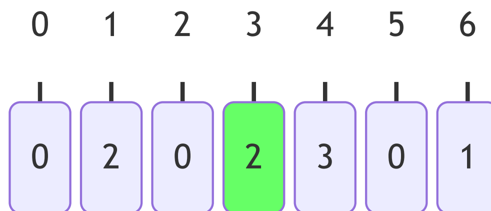
- soit B le tableau trié

Tri linéaire: tri par dénombrement

- soit A un tableau d'entier inférieur ou égal à 6



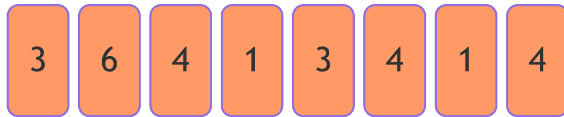
- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



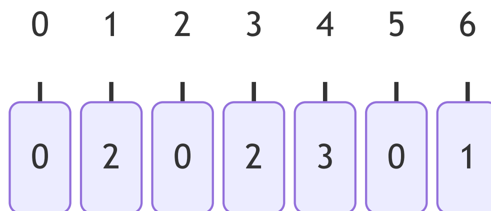
- soit B le tableau trié

Tri linéaire: tri par dénombrement

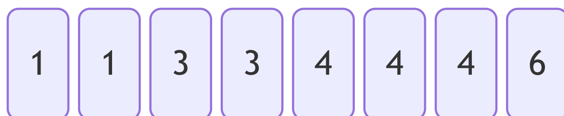
- soit A un tableau d'entier inférieur ou égal à 6



- soit C un tableau qui compte le nombre de fois qu'apparaît un entier



- soit B le tableau trié



Optimalité des tris vus en cours

- La borne inférieure pour un tri par comparaison est: $\Omega(n \log_2 n)$
- Donc seul les tris qui ont une complexité en $O(n \log_2 n)$ sont optimaux.
- Le tri fusion est optimal mais pas la version du tri rapide présenté dans ce cours!

Récapitulatif

Algorithme	O	Ω	Principe
Insertion	$O(n^2)$	$\Omega(n)$	Prendre un élément non encore trié, l'insérer à sa place dans les éléments triés
Sélection	$O(n^2)$		Choisir le plus petit élément de la partie non triée, le mettre à la fin de la partie triée
Permutation	$O(n^2)$		Si 2 éléments voisins ne sont pas ordonnés on les échange
Fusion	$O(n \log_2 n)$		On divise le tableau en élément plus petit, les tableaux sont triés à la reconstruction
Rapide	$O(n^2)$	$\Omega(n \log_2 n)$	Utilisation d'un pivot, les éléments sont triés en séparants les listes

Autres tris

- Tri cocktail
- Tri par tas
- Smoothsort
- Tri de Shell
- Tri à peigne

Livre conseillé