



Développement Efficace

3. Algorithmes de Tri

BUT INFO 2A

David Auger

IUT de Vélizy - UVSQ



3.1- Introduction générale aux tris

- ▶ Objectif : réorganiser une suite de valeurs (typiquement dans un tableau) en ordre croissant (ou décroissant).
- ▶ Applications : recherche, bases de données, algorithmique avancée.
- ▶ Problématiques :
 - ▶ efficacité (nombre de comparaisons, échanges, ou complexité globale),
 - ▶ simplicité d'implémentation,
 - ▶ stabilité et mémoire extérieure (tri en place ou non).
- ▶ : pour visualiser : <https://visualgo.net/en/sorting>

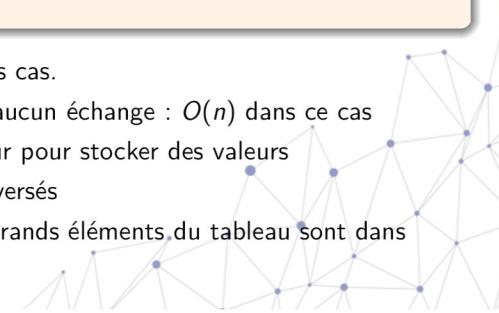


3.2- Tri à bulles

Idée : faire remonter les grands éléments en comparant par paires.

Code Python

```
def bubble_sort(T):
    n = len(T)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if T[j] > T[j+1]:
                T[j], T[j+1] = T[j+1], T[j]
```



- ▶ Complexité et comparaisons : $O(n^2)$ dans le pire des cas.
- ▶ Amélioration : arrêt du tri si la boucle sur j ne fait aucun échange : $O(n)$ dans ce cas
- ▶ le tri est **en place** : n'utilise pas un tableau extérieur pour stocker des valeurs
- ▶ le tri est **stable** : les éléments égaux ne sont pas inversés
- ▶ **invariant de boucle** : après la boucle i , les i plus grands éléments du tableau sont dans l'ordre en fin de tableau.

3.3- Tri par sélection

Idée : sélectionner le plus petit élément restant et le mettre à sa place définitive.

Code Python

```
def selection_sort(T):
    n = len(T)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if T[j] < T[min_idx]:
                min_idx = j
        T[i], T[min_idx] = T[min_idx], T[i]
```

- ▶ Comparaisons : $\frac{n(n-1)}{2}$ exactement soit une complexité $O(n^2)$.
- ▶ en place
- ▶ **non stable** : considérer [2,2,1]
- ▶ invariant de boucle : après la boucle i , les i plus petits éléments sont triés et placés en début de tableau.



3.4- Tri par insertion

Idée : insérer chaque élément à sa place dans le tableau trié précédent.

Code Python

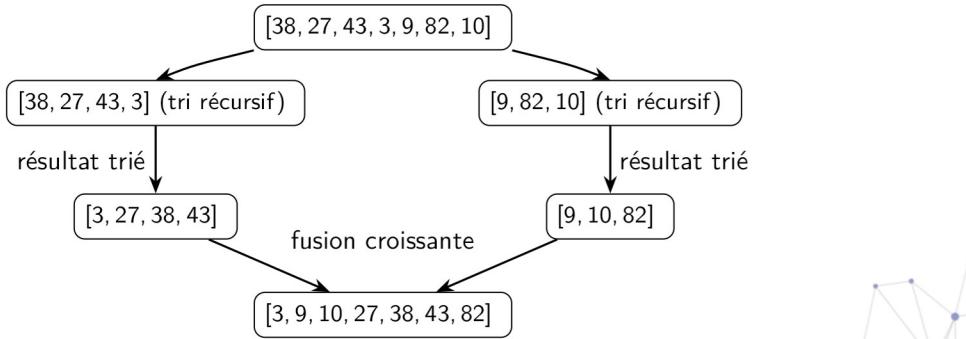
```
def insertion_sort(T):
    for i in range(1, len(T)):
        x = T[i]
        j = i-1
        while j >= 0 and T[j] > x:
            T[j+1] = T[j]
            j -= 1
        T[j+1] = x
```

- ▶ Comparaisons : $O(n^2)$ (meilleur cas $O(n)$).
- ▶ Tri stable et en place.
- ▶ Invariant de boucle : après la boucle i , les i premiers éléments du tableau sont triés



3.5- Tri par fusion : vue d'ensemble

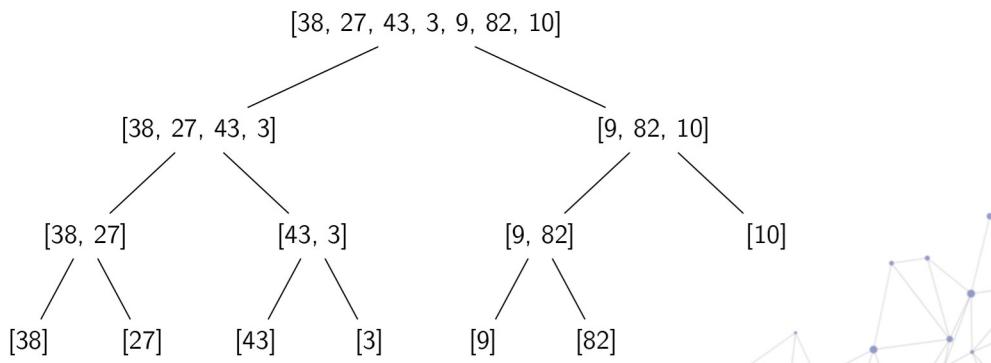
Le tri par fusion est un tri naturellement **récursif** qui utilise la technique **diviser pour régner**.



principe du tri fusion

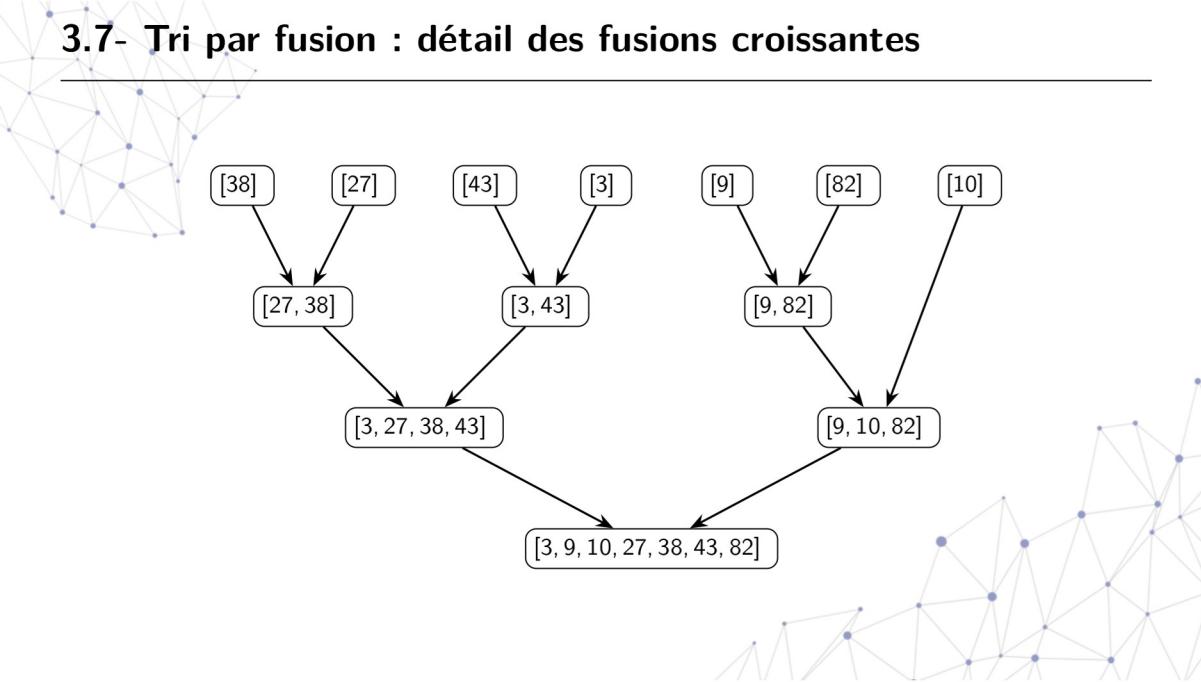
- On divise le tableau en deux sous-tableaux, chacun est trié récursivement.
- Puis on procède à la **fusion croissante** des deux résultats triés.

3.6- Tri par fusion - détail des appels récursifs



On divise par deux récursivement jusqu'à obtenir des listes d'un élément.

3.7- Tri par fusion : détail des fusions croissantes



3.8- Fusion croissante : code python

Code Python

```
def fusion_croissante(tab1, tab2):  
  
    n1, n2 = len(tab1), len(tab2)  
    fusion = [None]*(n1+n2)  
    i1, i2 = 0, 0  
  
    for i in range(n1+n2):  
        if i2==n2 or (i1<n1 and tab1[i1] <= tab2[i2] ):  
            fusion[i] = tab1[i1]  
            i1 += 1  
        else:  
            fusion[i] = tab2[i2]  
            i2 += 1  
    return fusion
```

La fusion croissante a une complexité algorithmique en $\mathcal{O}(n)$ où $n = \text{len}(\text{tab1}) + \text{len}(\text{tab2})$ est le nombre total d'éléments fusionnés.

3.9- Tri par fusion : code python

Code Python

```
def tri_fusion(tab):  
  
    n = len(tab)  
  
    if  n<= 1:  
        return tab  
  
    m = n //2  
    tab1 = tri_fusion(tab[:m])  
    tab2 = tri_fusion(tab[m:])  
    return fusion_croissante(tab1,tab2)
```

3.10- Complexité du tri par fusion

Considérez le code :

dichotomie

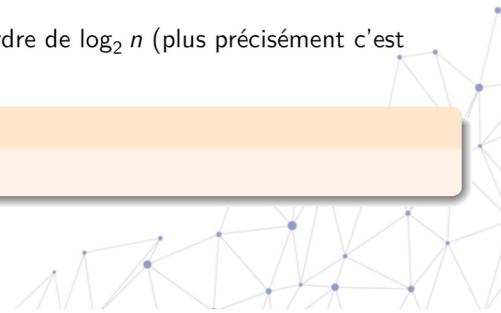
```
while n > 1:  
    n = n // 2
```

Si $n = 2^k$, on a exactement k passages dans la boucle.

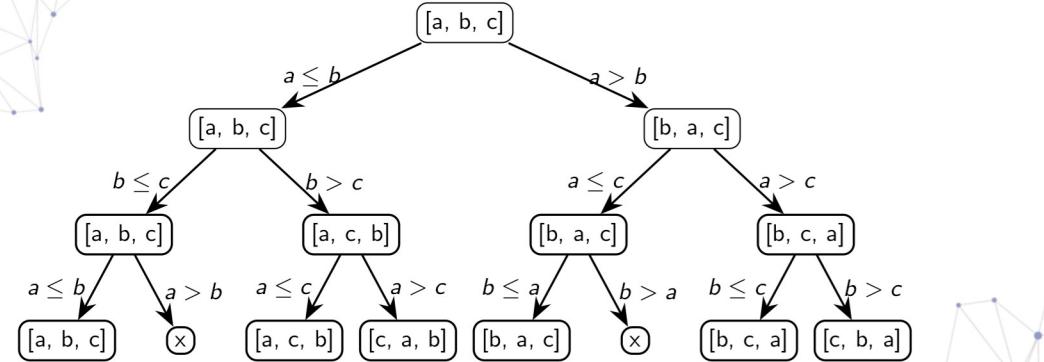
On peut en déduire que la profondeur de récursion de l'ordre de $\log_2 n$ (plus précisément c'est $k = \lceil \log_2(n) \rceil$). On peut alors prouver :

Complexité du tri fusion

Le tri par fusion a une complexité en $\mathcal{O}(n \log n)$.



3.11- Arbre de décision du tri à bulles (3 éléments)



- ▶ Chaque nœud montre l'état du tableau courant.
- ▶ Les feuilles couvrent toutes les ordres possibles.
- ▶ Les 'x' sont des cas impossibles.

3.12- Optimalité du tri par fusion

- On peut faire un arbre de décision similaire pour n'importe quel algorithme de tri par comparaisons quand il est exécuté sur un tableau de taille n .
- Les feuilles de l'arbre contiennent les $n!$ ordres possibles du tableau (éventuellement plusieurs fois).
- Donc le nombre de feuilles de l'arbre est au moins $n!$.
- Un arbre binaire de hauteur h possède au plus 2^h feuilles.
- Donc $2^h \geq n!$ soit $h \geq \log_2(n!) \approx n \log n$.

Résultat

Tout algorithme de tri par comparaisons effectue dans le pire des cas au moins (approximativement) $n \log_2 n$ comparaisons sur un tableau de taille n . La complexité de cet algorithme est donc au moins $O(n \log n)$.

Le tri par fusion a donc une complexité optimale parmi les tris par comparaison.

3.13- Tris linéaires

Idée : On peut faire des tris plus rapides que $O(n \log n)$ en sortant du cadre des tris par comparaison : il faut alors exploiter des informations supplémentaires sur les données.

- ▶ **Tri par comptage (Counting Sort)** : $O(n + k)$.
- ▶ **Radix Sort** (tri par base) : $O(d \cdot (n + k))$.
- ▶ **Bucket Sort** : $O(n)$ attendu sous hypothèses de distribution uniforme.

Exemple : Counting Sort

```
def counting_sort(T, k):  
    C = [0]*(k+1)  
    for x in T:  
        C[x] += 1  
    i = 0  
    for val in range(k+1):  
        for _ in range(C[val]):  
            T[i] = val; i += 1
```