Algorithmes de tri

1. Définition et idée générale

Un algorithme de **tri** réorganise une liste pour mettre ses éléments dans un ordre donné, généralement croissant.

> ⚠️ **Attention**
>
> Un **tri correct** doit **conserver les valeurs inchangées** et **produire un ordre final *cohérent*.**
>
> Cela signifie que le tri ne doit ni *perdre*, ni *modifier*, ni *ajouter* de valeurs : on souhaite uniquement *permuter* les éléments initialement présents dans la liste, de sorte que ceux-ci soient rangés dans un ordre *défini* (croissant ou décroissant).

2. Illustration immédiate

Trier, c'est ordonner. Ainsi, on souhaite trouver un moyen pour que la liste Python `[3, 1, 4, 1, 5]` devienne, après traitement, `[1, 1, 3, 4, 5]`.

La question algorithmique est : comment faire cela efficacement ?

3. Les tris au programme

En première NSI, les tris rencontrés sont le **tri par insertion** (https://fr.wikipedia.org/wiki/Tri_par_insertion) et le **tri par sélection** (https://fr.wikipedia.org/wiki/Tri_par_s%C3%A9lection). Ces tris sont simples à comprendre, mais peu efficaces sur de grandes listes.

> ✅ **Astuce mnémotechnique**
>
> - Le tri par **insertion** est le tri du joueur de cartes : lorsqu'on veut trier une « main de cartes », on observe une carte et on l'**insère** directement à « la bonne place ».
> - Le tri par sélection consiste en :
>
> - sélectionner la plus petite valeur d'un tableau et la mettre à la première place de ce tableau ;

- sélectionner la *deuxième* plus petite valeur du tableau et la mettre à la **deuxième** place de ce tableau (c'est *la* plus petite parmi les valeurs *restant à trier*) ;
- et ainsi de suite.

En terminale, **le tri-fusion** (https://fr.wikipedia.org/wiki/Tri_fusion) est l'exemple important à connaître, car il illustre la notion de récursivité, et le principe « diviser pour régner » (DPR). En outre, il est efficace en raison d'une bonne complexité.

💭 **Remarque : pourquoi des tris inefficaces sont-ils néanmoins utilisés ?** ▼

Le tri par **insertion** présente un intérêt *réel* lorsque les données sont *presque triées* (exemple typique : on veut ranger « au bon endroit » un score dans un tableau des « high scores », par définition déjà trié). Dans cette situation particulière, le tri par insertion peut être très rapide, *plus efficace* que des tris plus rapides en général.

3.1. Tri par insertion

Considérons un tableau d'entiers `tab` de longueur `n`. Pour ranger ces entiers par ordre croissant, la logique du tri par insertion consiste, pour chaque position `i` allant de `1` à `n-1` (incluses), à « faire glisser » `tab[i]` vers la gauche « autant que possible ».

Pour cela, il faut :

- mémoriser la valeur `tab[i]` dans une variable `x` ;
- faire remonter la valeur `tab[i-1]` à la position `i` *si possible* : c'est le cas si `tab[i-1] > tab[i]` ;
- faire remonter la valeur `tab[i-2]` à la position `i-1` *si possible* (critère similaire) ;
- etc. ;
- lorsque le critère de déplacement n'est plus rempli, on a trouvé la position `k` où placer `x`.

Code Python

```
def tri_insertion(tab):
    n = len(tab)
    if n > 1:                               # Cas limite explicite
        for i in range(1, n):
            x = tab[i]                      # Valeur à « faire glisser »
            k = i
            while k > 0 and tab[k - 1] > x: # Important de vérifier k > 0 !
                tab[k] = tab[k - 1]         # Remontée de la valeur précédente
                k = k - 1                   # Passage à la valeur précédente
            tab[k] = x                      # Placer x à sa bonne position
```

Code Python avec affichage

```
def tri_insertion(tab):
    n = len(tab)
    if n > 1:
        for i in range(1, n):
            x = tab[i]
            print(f"On regarde {x} en position {i}")
            k = i
            while k > 0 and tab[k - 1] > x:
                print(f"  ...on remonte {tab[k - 1]}")
                tab[k] = tab[k - 1]
                print(tab)
                k = k - 1
            print(f"   > On positionne {x} en {k}")
            tab[k] = x
            print(tab)
```

✅ **À retenir** ▼

- Pour une liste de taille n, le coût moyen de son tri est fonction de n^2.
- Ce tri, bien que peu performant en moyenne, est très performant lorsque la liste est « presque triée ».

✏️ **Exemple : hall of fame** ▼

Imaginez un jeu qui conserve la liste ordonnée des meilleurs scores. Insérer un nouveau score amène à insérer une nouvelle valeur dans une liste déjà triée. Ajouter ce nouveau score à la fin de cette liste conduit à créer une liste « presque triée ». Dans cette situation, l'efficacité du tri par insertion est *très bonne* : il suffit de faire glisser la nouvelle valeur à sa bonne place. La complexité est alors en $O(n)$.

3.2. Tri par sélection

On considère toujours un tableau d'entiers `tab` de longueur `n`. La logique du tri par sélection consiste à : - chercher entre les positions `0` et `n-1` celle de la plus petite valeur : notons-la `k`. On échange ensuite les valeurs en position `0` et `k` :

```
tab[0], tab[k] = tab[k], tab[0]
```

- chercher entre les positions `1` et `n-1` celle de la (2e) plus petite valeur. On échange ensuite les valeurs en position `1` et `k` ; - et ainsi de suite jusqu'à ce que la position de départ de la recherche vaille `n-2`.

Pour cela, il faut parcourir les positions `0` à `n-1` avec une variable `i` et :

- parcourir les positions `i` à `n-1` avec une variable `j` ;
- trouver la plus petite valeur du tableau entre les positions `i` et `n-1` : notons `k` sa position ;
- échanger `tab[i]` et `tab[k]` (seulement si `i != k`).

Code Python

```python
def tri_selection(tab):
    n = len(tab)
    if n > 1:                               # Cas limite explicite
        for i in range(n - 1):
            m = tab[i]                      # Minimum initialisé
            pos_m = i                       # Position de ce min. initialisée
            for j in range(i, n):           # Recherche du minimum "au-delà"
                if tab[j] < m:
                    m = tab[j]              # "Meilleur" minimum trouvé
                    pos_m = j               # Sa position
            if pos_m != i:
                tab[i], tab[pos_m] = tab[pos_m], tab[i]     # Échange
```

Code Python avec affichage

```python
def tri_selection(tab):
    n = len(tab)
    if n > 1:
        for i in range(n - 1):
            print(tab)
            m = tab[i]
            pos_m = i
            for j in range(i, n):
                if tab[j] < m:
                    m = tab[j]
                    pos_m = j
            print(f"    - Min : {tab[pos_m]} en {pos_m} ∈ [{i};{n-1}]")
            if pos_m != i:
                tab[i], tab[pos_m] = tab[pos_m], tab[i]
                print(f"    - On échange les valeurs aux pos {pos_m} et {i}")
```

3.3. Tri-fusion

Voir la fiche dédiée.

4. Liens avec d'autres notions

- La **récursivité** est au cœur du tri-fusion.
- La méthode **diviser pour régner** décrit parfaitement sa structure.
- La **complexité** du tri fusion est en $O(n \log n)$, celle des tris abordés en classe de première est, en moyenne, en $O(n^2)$.
- Le **logarithme base 2** aide à comprendre cette complexité.