# Tri-fusion

## 1. Définition et idée générale

Le **tri-fusion** est un algorithme de tri qui divise une liste en deux, trie les deux moitiés, puis fusionne les résultats. C'est un bon exemple de l'approche **diviser pour régner** (au programme de terminale NSI).

Sa complexité est en $O(n \log n)$ : c'est efficace !

## 2. Illustration

Cela ressemble à une situation où deux personnes décident de collaborer pour classer chacune une moitié d'une pile de feuilles, avant de réunir les deux piles déjà ordonnées.

Voici une illustration provenant de Wikipedia, illustrant le processus avec des nombres :

Cliquez ici pour une visualisation dynamique de la version récursive (https://www.recursionvisualizer.com/?function_definition=def%20fusion%28gauche%2C%20droite%29%3A%20%20%20%20%0A%20%20%20%20resultat%20%3D%20%5B%5D%0A%20%20%20%20%0A%20%20%20%20resultat%20%3D%20resultat%20%2B%20gauche%5Bindex_gauche%3A%5D%20%2B%20droite%5Bindex_droite%3A%5D%0A%

## 3. Algorithmes

### 3.1. Tri-fusion récursif

L'algorithme ci-dessous, tiré de Wikipedia, définit le fonctionnement d'une fonction récursive `tri_fusion` (une version itérative est possible). Elle effectue le tri-fusion en faisant appel à la fonction auxiliaire `fusion` qui, elle, ne s'occupe que de la fusion.

```
entrée : un tableau T
sortie : une permutation triée de T
fonction tri_fusion(T[1, …, n])
    si n ≤ 1
        renvoyer T
    sinon
        renvoyer fusion(tri_fusion(T[1, …, n/2]), tri_fusion(T[n/2 + 1, …, n]))
```

💭 **Remarque : précisions diverses ▼**

- Le pseudo-code ci-dessus montre le recours à une logique de *« slicing »*. Il est toujours possible de l'éviter (grâce à une compréhension de liste en Python, par exemple).
- Cela peut avoir de l'importance : en Python, le *slicing* effectue une **copie** de la liste initiale, ce qui a un *coût* (en temps et en mémoire).
- La numérotation des cases du tableau est faite à partir de `1` : en Python, on compte à partir de `0`.
- Lorsqu'on trouve `n/2` dans le pseudo-code, il faut comprendre *« partie **entière** de `n/2` »*.
  En Python, on écrirait `n // 2`.

### 3.2. Fusion récursive de deux listes

La fonction suivante est également définie récursivement (là encore une version itérative existe). Le symbole `⊕` désigne la *concaténation* de tableaux.

```
entrée : deux tableaux TRIÉS, nommés A et B
sortie : un tableau trié qui contient exactement les éléments des tableaux A et B
fonction fusion(A[1, …, a], B[1, …, b])
    si A est le tableau vide
        renvoyer B
    si B est le tableau vide
        renvoyer A
    si A[1] ≤ B[1]
        renvoyer A[1] ⊕ fusion(A[2, …, a], B)
    sinon
        renvoyer B[1] ⊕ fusion(A, B[2, …, b])
```

💭 **Remarque : concaténation de tableaux en pratique ? ▼**

En pratique, concaténer des listes Python se fait simplement avec l'opérateur `+`. Mais le faire un grand nombre de fois a aussi un *coût*, car cette fusion *crée un **nouvel objet** en mémoire*.

## 4. Codes Python

### 4.1. Fusion

La fonction de fusion compare les premières valeurs de deux listes triées et construit progressivement une nouvelle liste triée.

**Version récursive**

```python
def fusion(A, B):
    if A == [] :
        return B
    if B == [] :
        return A
    if A[0] <= B[0]:
        return [A[0]] + fusion(A[1:], B)
    else:
        return [B[0]] + fusion(B[1:], A)
```

Cette version utilise intensivement le *slicing* et la concaténation de listes Python : cela se paie *cher* en temps d'exécution !

**Version itérative**

```python
def fusion(gauche, droite):
    len_g, len_d = len(gauche), len(droite)
    resultat = [None] * (len_g + len_d)   # On crée d'emblée UNE liste de la bonne taille, qui
    # sera peuplée par les valeurs de `gauche` et `droite`, dans le bon ordre, puis renvoyée.
    i = j = 0
    while i < len_g and j < len_d:        # Tant qu'il reste des éléments dans les 2 listes
        if gauche[i] <= droite[j]:        # on compare les valeurs en cours : la plus petite
            resultat[i+j] = gauche[i]     # va dans la liste fusionnée, qu'elle vienne de gauche
            i += 1                        # ...
        else:                             # ...
            resultat[i+j] = droite[j]     # ou de la liste de droite.
            j += 1                        # On actualise la position associée, pour poursuivre.
    if j == len_d:                        # Si on a épuisé la liste de droite
        while i < len_g:                  # on complète, avec les valeurs restant dans la liste
            resultat[i+j] = gauche[i]     # de gauche, la liste qui sera renvoyée à la fin.
            i += 1
    else:                                 # Sinon,
        while j < len_d:                  # on fait de même avec ce qui reste
            resultat[i+j] = droite[j]     # dans la liste de droite.
            j += 1
    return resultat
```

On cherche ici à éviter absolument le *slicing* et la concaténation des `list` Python provoquée par le `+`. Le code est moins lisible, l'idée générale est sensiblement moins perceptible… mais le programme en résultat est plus performant.

💭 **Remarque : mais pourquoi ce code est-il plus performant ?** ▼

Python est un langage très expressif : il permet simplement de « faire des choses » qui demanderaient plus d'effort dans un autre langage. Mais il faut parfois aller au-delà de la « magie de Python » pour comprendre ce qui se cache derrière… et les conséquences qui en découlent. Ainsi, le recours au *slicing* (ou à une compréhension de liste) ainsi que l'emploi de la concaténation de listes amènent Python à *créer de **nouveaux** objets* en mémoire : cela a un coût ! De même l'utilisation des méthodes `append()` ou `extend()` entraîne régulièrement des surcoûts. Tout cela finit par dégrader le niveau de performance d'un code !

## 4.2. Tri-fusion récursif

```python
def tri_fusion(tab):
    if len(tab) <= 1:
        return tab
    milieu = len(tab) // 2
    gauche = tri_fusion(tab[:milieu])
    droite = tri_fusion(tab[milieu:])
    return fusion(gauche, droite)
```

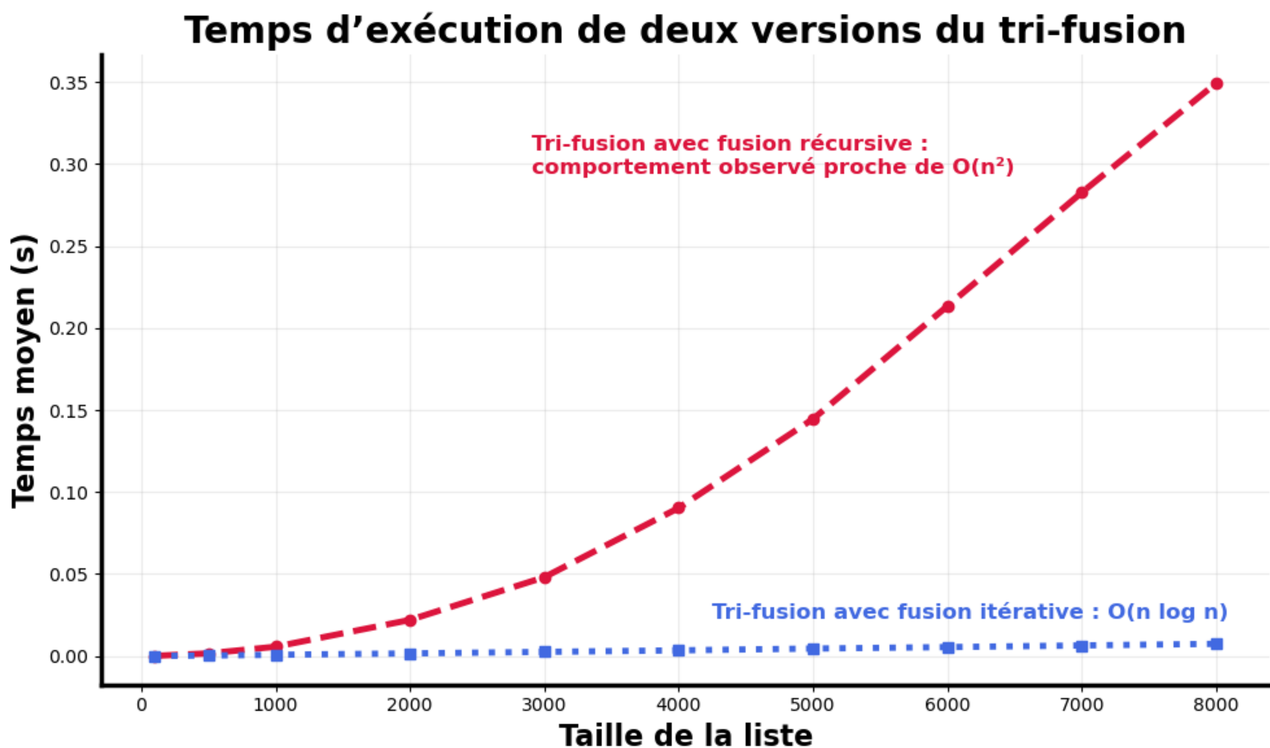
- On voit bien la récursivité du tri-fusion dans ce code, et son lien direct avec la méthode « diviser pour régner ».
- Le cas de base est une liste de taille 0 ou 1 (qui ne nécessite donc aucun traitement).

💭 **Remarque : il y a *slicing* !** ▼

Certes, et on peut à nouveau le remplacer par des compréhensions de listes.

💭 **Remarque : *quid* des performances ?** ▼

Voici un graphique illustrant la différence entre une implémentation *itérative* ***efficace*** de la partie fusion du tri-fusion, et la *fusion récursive* ***naïve*** de cette même fusion. Le *coût* ***pratique*** de la version *récursive* ***naïve*** ne correspond pas à la complexité *théorique* de l'idée derrière le tri-fusion. C'est bien la version Python récursive avec *slicing* et concaténations qui dégrade *fortement* le comportement observé, au point qu'on observe un comportement proche de $O(n^2)$.




## 5. Liens avec d'autres notions

- La **récursivité** est au cœur de cet algorithme.
- Ce tri relève de l'approche **diviser pour régner**.
- La **complexité** du tri-fusion est en $O(n \log n)$.
- La **recherche dichotomique** partage l'idée de couper le problème en deux.