Chapitre 10
Tris

Simon Dauguet
simon.dauguet@gmail.com

jeudi 28 mai, 2026

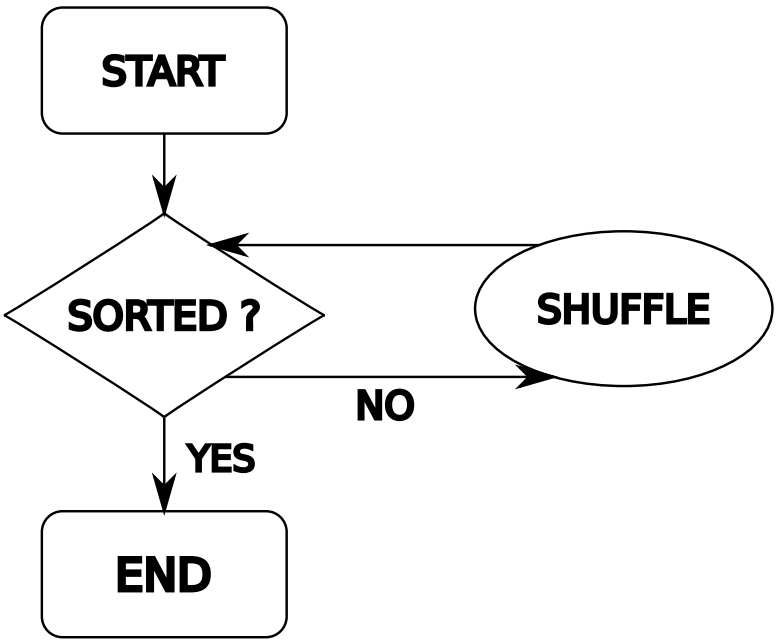


Table des matières

1 Tri comptage

1.1 Présentation

Définition 1.1 (Tri comptage) :
Lorsqu'un tableau `T` possède des entiers naturels (ou des valeurs transformables en entiers naturels) toutes strictement inférieures à m, on compte à l'indice i d'un tableau de comptage (de taille m) le nombre de valeurs i du tableau initial. On crée alors le tableau trié à partir de ce tableau de comptage.

Exemple 1.1 :
On considère le tableau `tab = [1,0,5,5,1,5,8,1,1]`.

Le tableau possède 1 zéro, 4 un, 3 cinq et 1 huit, ainsi `1*[0]+4*[1]+3*[5]+1*[8]=[0,1,1,1,1,5,5,5,8]` est le tableau trié.

Exercice 1 :
Après avoir compté chaque valeur, trier le tableau `[-2,1,0,1,-2,8,7,1,0,8]`.

On peut implémenter un tri comptage ainsi :

```python
def triComptage(tab: list) -> list:
    """Tri comptage pour un tableau d'entiers."""
    m = max( tab ) + 1
    tcomptage = m * [0]
    i = 0
    for i in range(len(tab)) :
        # variant : len(tab)-i
        # invariant : tcomptage[j] vaut le nombre de j dans le sous-tableau tab[:i]
        tcomptage[ tab[i] ] += 1
    ttrie = []
    i = 0
    for i in range(m) :
        # variant : m-i
        # invariant : ttrie est le tableau trié de tab comportant toutes les valeurs inférieures à i
        ttrie += tcomptage[i] * [ i ]    # on ajoute tcomptage[i] cases contenant i
    return(ttrie)
```

1.2 Étude

Terminaison

Les deux boucles sont ici des boucles `for`. Il n'y a donc pas de problème d'arrêt. L'algorithme se termine bien.

Néanmoins, les deux variants de boucles sont respectivement `len(tab)-i` et `m-i`, qui sont bien deux quantités strictement décroissantes à cause de l'incrémentation automatique de `i` dans les boucles `for`.

Correction

Les invariants de boucles sont indiqués. Il faut donc montrer, par récurrence finie, que ces deux invariants sont effectivement des invariants. Et donc que l'algorithme est correct.

Ce n'est pas très dur, mais un peu fastidieux à écrire.

Complexité temporelle

Posons n la longueur de la liste (*i.e.* `n=len(tab)`).

Remarquons d'abord que la somme des éléments de `tcomptage` est égale à `len(tab)=n` (*i.e.* $\sum_{i=0}^{m-1} tcomptage[i] = n$).

La fonction `max` peut se coder facilement (voir les chapitres précédents). Elle ne nécessite qu'un seul parcours de la liste. Donc la fonction `max` a une complexité en $O(n)$.

La première boucle `for` ne pose pas de problème particulier pour calculer sa complexité. C'est la deuxième qui pose un problème, à cause du nombre d'opérations qui change en fonction de l'avancement dans la boucle.

La dernière ligne de la deuxième boucle `for` effectue $tcomptage[i]$ produit, une concaténation et une affectation. Donc le corps de la deuxième boucle `for` effectue $tcomptage[i]+2$ opération. On passe m fois dans cette boucle `for`. Donc le nombre d'opérations dans la deuxième boucle `for` est

$$\sum_{i=0}^{m-1} (tcomptagee[i] + 2) = 2m + n$$

en utilisant la remarque précédente.

Finalement, on peut en déduire que la complexité de cet algorithme est $O(n, m)$. En admettant que la taille de la liste est de l'ordre de son maximum (*i.e.* si $O(n) = O(m)$), alors la complexité de `triComptage` est $O(n)$ (donc linéaire).

Exercice 2 :
On considère un tableau contenant des entiers compris entre -100 et 100. Proposer une fonction `triComptageZ(tab: l` qui implémente un tri comptage sur des tableaux à valeurs entières.

Préciser les variants et invariants de boucles ainsi que la complexité temporelle qui en résulte. Enfin proposer plusieurs appels qui testent votre algorithme.

2 Tri à bulles

2.1 Présentation

Le tri à bulles consiste à parcourir le tableau et intervertir deux éléments mitoyens s'ils sont dans le mauvais ordre. On fait ainsi "remonter" la plus grande des valeurs à chaque fois, un peu comme une bulle. Lors du premier parcours, c'est la plus grosse bulle qui se retrouve tout en haut (à la fin). On re-parcourt alors de nouveau le tableau autant de fois que nécessaire pour positionner toutes les bulles correctement.

Donc à chaque nouveau parcours, c'est la "nouvelle plus grosse bulle" qui sera placée correctement. Il suffit donc de s'arrêter un peu avant la fin, avant que les plus grosses bulles soient bien placées.

Exemple 2.1 :
On considère le tableau `tab = [6,1,0,5,8,1]`.

étape	indices comparés	état du tableau
0	-	[6,1,0,5,8,1]
1	0-1	[1,6,0,5,8,1]
2	1-2	[1,0,6,5,8,1]
3	2-3	[1,0,5,6,8,1]
4	3-4	[1,0,5,6,8,1]
5	4-5	[1,0,5,6,1,8]

Une fois le tableau parcouru une fois, le plus grand élément est forcément bien positionné (les plus grandes valeurs sont assimilables à des bulles qui montent vite).

On recommence le parcours du tableau en s'arrêtant à la pénultième valeur. Puis on recommence en s'arrêtant à l'anté-pénultième, etc. jusqu'à ce que le tableau soit bien trié.

étapes	indices comparés	état du tableau
6-9	de 0-1 à 3-4	[0,1,5,1,6,8]
10-12	de 0-1 à 2-3	[0,1,1,5,6,8]
13-14	0-1,1-2	[0,1,1,5,6,8]

Puisqu'il n'y a eu aucune interversion au dernier parcours, le tableau est trié dans l'ordre croissant et on peut s'arrêter.

Ainsi l'algorithme du tri à bulles peut s'écrire en pseudo-code (sans diminution du tableau à trier) :

Tant que le tableau n'est pas trié,
    Pour toutes les valeurs du tableau
        échanger les deux valeurs mitoyennes si celle de droite est plus petite

On propose l'implémentation suivante (on peut retirer le booléen **bDejaTrie** pour avoir une complexité toujours égale au pire cas, ou retirer la **copy()** initiale pour obtenir un tri en place).

```
1  def triBulles(tab: list) -> list:
2      ttrie = tab[:]    # copie du tableau. A supprimer pour modifier le tableau initial.
3      for i in range( len(ttrie)-1 ):
4          # variant : len(ttrie)-1-i
5          # invariant : les i plus grandes valeurs de ttrie sont triées dans le sous-tableau
   ttrie[-1-i:]
6          bDejaTrie = True            # Permet un arrêt anticiper dans le cas où la liste
   est déjà trié
7          for j in range( len(ttrie)-1-i ):
8              # variant : len(ttrie)-1-i-j
9              # invariant : ttrie[j] est supérieure à toute valeur du sous-tableau ttrie[:j]
10             if ttrie[ j+1 ] < ttrie[ j ]:    # Mettre > pour avoir l'autre sens.
11                 tmp = ttrie[ j ]     # variable tampon pour faire l'interversion. Ou
   : ttri[j],ttri[j+1] = ttri[j+1],ttri[j]
12                 ttrie[ j ] = ttrie[ j+1 ]
13                 ttrie[ j+1 ] = tmp
14                 bDejaTrie = False    # Une interversion à eu lieu. Donc la liste n'est
   pas encore trié.
15         if bDejaTri: return(ttrie)   # Dans le cas où il n'y a pas eu d'interversion,
   la liste est alors bien trié et on peut s'arrêter
16     return(ttrie)
```

2.2 Étude

Terminaison

Il n'y a que des boucles **for** là encore. Le problème de la terminaison de l'algorithme ne se pose donc pas. Les variants de boucles sont clairement identifiés et sont automatiquement strictement décroissants.

Correction

Là encore, les invariants sont donnés. Il reste seulement à montrer par récurrence que ce sont bien des invariants. Il n'y a pas de problèmes majeurs sur les récurrences. Uniquement sur la rédaction compte tenu du fait que ce sont des objets informatiques et pas mathématiques.

Complexité temporelle

On pose **n=len(tab)**. Il y a deux boucles **for** imbriquées l'une dans l'autre. La première est parcourue n fois. Pour tout $i \in \{0, \dots, n-1\}$, indice de la première boucle **for**, la seconde boucle **for** est parcourue $n-i-1$. Le corps de la deuxième boucle **for** a un nombre fixe d'opérations (14 opérations).

Donc le nombre d'opérations totales dans la première boucle `for` est alors

$$\sum_{i=0}^{n-2}\left(1+\sum_{j=0}^{n-i-2}14\right)=2(n-1)+14\sum_{i=0}^{n-2}(n-i-1)=7n^2-5n-2.$$

La première ligne n'est pas obligatoire et a de toute façon une complexité linéaire ($O(n)$).
Donc la complexité de `triBulles` est $O(n^2)$ quadratique.

Exercice 3 :
Écrire une fonction itérative `triBullesC(tab: list) -> list`, qui trie un tableau contenant des couples `(lettre, nombre)` selon leur nombre.

Déterminer les variants et invariants de boucles ainsi que la complexité temporelle.

3 Tri par insertion

3.1 Présentation

Le tri par insertion consiste à considérer successivement les sous-tableau de gauche du tableau et insérer correctement chaque nouvelle variable correctement dans le sous-tableau déjà trié.

Autrement dit, à l'étape i, on suppose le tableau `tab[0:i-1]` bien trié. On considère la nouvelle valeur `tab[i]`. On l'insère à la bonne place dans `tab[0:i-1]`. Alors `tab[0:i]` est bien trié. Et on passe à l'élément suivant.

Exemple 3.1 :
On considère `tab=[6,1,0,5,8,1]`.

Étape	Nouvel indice	Valeur du nouvel élément	Variable tampon d'inversion	État du tableau
0				[6,1,0,5,8,1]
1	1	1	1	[1,6,0,5,8,1]
2	2	0	0	[1,6,6,5,8,1]
3			0	[0,1,6,5,8,1]
4	3	5	5	[0,1,6,6,8,1]
5			5	[0,1,5,6,8,1]
6	4	8	8	[0,1,5,6,8,1]
7	5	1	1	[0,1,5,6,8,8]
8			1	[0,1,5,6,6,8]
9			1	[0,1,5,5,6,8]
10			1	[0,1,1,5,6,8]

On peut donc résumer l'algorithme avec le pseudo-code suivant :

Pour tous les éléments du tableau,
    sauvegarder cette valeur dans une variable
    Tant que la valeur à gauche existe et est plus grande que cette valeur,
        décaler la valeur de gauche à droite
    mettre la valeur sauvegardée dans la dernière case décalée

On propose l'implémentation suivante (il suffit de retirer la `copy()` initiale pour obtenir un tri en place) :

```
1  def triInsertion(tab: list) -> list:
2      ttrie = tab[:]    # copie du tableau. A enlever pour modifier le tableau initial
3      for i in range(1, len(ttrie)):
4          # variant : len(ttrie)-i
5          # invariant : le sous-tableau ttrie[:i] est trié dans l'ordre croissant
6          val = ttrie[ i ]     # variable tampon de la valeur à replacer
7          j = i-1              # on parcours le tableau déjà à trié de la position actuelle
8          while j >= 0 and val < ttrie[j]: # on s'arrête dès qu'on arrive dans un partie
   trié "trop petite"
9              # variant : j
10             # invariant : val est inférieure à toutes valeurs de ttrie[j:i+1]
11             ttrie[ j+1 ] = ttrie[ j ]    # on décale les valeurs vers la droite dans
   le tableau trié à gauche
12             j = j-1
13         ttrie[ j+1 ] = val
14     return(ttrie)
```

3.2 Étude

Terminaison

La première boucle est une boucle `for`, donc pas de problème pour la terminaison.

Dans la boucle `while`, c'est la variable `j` qui joue le rôle de variant de boucle. Elle est décrémentée à chaque passage. C'est donc bien une suite d'entier strictement décroissante. C'est donc bien un variant de boucle. Et donc la boucle s'arrêtera.

Donc l'algorithme se termine bien.

Correction

Les invariants de boucles sont indiqués. Là encore, il faut démontrer que ce sont bien des invariants de boucles par récurrence.

Complexité temporelle

On pose encore `n=len(tab)`. La boucle `for` est parcourue $n-1$ fois. Le nombre de fois où la boucle `while` est parcourue est moins claire. La deuxième condition `val < ttrie[j]` écourte le processus. Dans le pire des cas, cette condition n'est jamais vérifiée (autrement dit, la liste est trié à l'envers) et donc chaque boucle `while` sera parcourue $i-1$ fois.

Chaque boucle `while` contient un nombre fixe d'opérations (précisément 10 opérations en comptant les conditions de la boucle). Ainsi, le nombre d'opérations dans la boucle `for` est :

$$\sum_{i=1}^{n-1}(4+10(i-1)+3) = 5n^2 - 8n + 3.$$

La première ligne est optionnelle et a une complexité linéaire $O(n)$.
Finalement, `triInsertion` a une complexité quadratique $O(n^2)$.

Exercice 4 :
On associe à un tableau `tab` un tableau de référence `ref`. Proposer une fonction itérative `triInsertionSelonRef(tab: list, ref: list) -> list`, qui trie le tableau `tab` dans l'ordre croissant selon les valeurs `ref`. Par exemple, `triInsertionSelonRef( ['A','B','C'], [4,9,1])`, renverra `['C','A','B']`.
Préciser les variants et invariants de boucles ainsi que la complexité temporelle.

4 Tri par sélection

4.1 Présentation

Le tri par sélection est un peu un miroir du tri par insertion. On se déplace dans le tableau à trié. Pour chaque nouvel indice, on suppose que le sous-tableau sur sa gauche est déjà trié. On sélectionne alors le minimum du sous-tableau de droite. Et on le place correctement dans le tableau bien trié de gauche en faisant un échange avec l'élément dont il prend la place à gauche. Puis, on passe à l'indice suivant.

Exemple 4.1 :
On considère la liste `tab=[6,1,0,5,8,1]`.

Étape	Indice	Tableau gauche	Tableau droite	Min à droite	État du tableau entier
0					[6,1,0,5,8,1]
1	0	[]	[6,1,0,5,8,1]	0	[0,1,6,5,8,1]
2	1	[0]	[1,6,5,8,1]	1	[0,1,6,5,8,1]
3	2	[0,1]	[6,5,8,1]	1	[0,1 1,5,8,6]
4	3	[0,1,1]	[5,8,6]	5	[0,1,1,5,8,6]
5	4	[0,1,1,5]	[8,6]	6	[0,1,1,5,6,8]

On peut donc résumer l'algorithme de tri par sélection en pseudo-code de la manière suivante :

Pour toutes les cases i du tableau,
  chercher la valeur minimale du sous-tableau de droite (case courante comprise)
  échanger ce minimum avec la valeur de la case courante

On propose l'implémentation suivante (il suffit de retirer la `copy()` initiale pour obtenir un tri en place) :

```
1  def triSelection(tab: list) -> list:
2      ttrie = tab[:]    # À enlever pour modifier le tableau initial
3      for i in range( len(ttrie)-1 ): # On s'arrête un cran avant la fin pour avoir
   un tableau d'au moins deux valeurs à droite.
4          # variant : len(ttrie)-1-i
5          # invariant : le sous-tableau ttrie[:i] est trié dans l'ordre croissant
6          jmin = i                  # Position du minimum de droite
7          vmin = ttrie[ jmin ]      # Minimum de droite
8          for j in range( i+1, len(ttrie) ):   # recherche du minimum à droite
9              # variant : len(ttrie)-j
10             # invariant : vmin est la valeur minimale du sous-tableau ttrie[i:j] (imin
   son indice)
11             if vmin > ttrie[ j ]:    # Mettre < pour avoir l'autre sens
12                 jmin = j
13                 vmin = ttrie[ jmin ]
14         ttrie[ jmin ] = ttrie[ i ]   # Échange du terme de la position courante et
   du minimum de droite
15         ttrie[ i ] = vmin            # Échange
16     return ttrie
```

4.2 Étude

Terminaison

Il n'y a que des boucles `for`. Donc la terminaison est assurée. Les variants de boucles sont automatiques.

Correction

Les invariants sont indiqués. Il n'y a plus qu'à faire les récurrences.

Complexité temporelle

Soit `n=len(tab)`. La première boucle `for` est parcourue $n-1$ fois. Il y a 3 opérations avant la deuxième boucle `for` et 5 après. Soit 8 opérations dans la première boucle `for` et en dehors de la deuxième boucle `for`.

La deuxième boucle `for` contient 5 opérations et elle est parcourue $n-1-i-1+1 = n-i-1$ fois, où i est l'indice de la première boucle `for`.

Par conséquent, le nombre d'opération dans la première boucle `for` est

$$\sum_{i=0}^{n-1}(8+5(n-i-1)) = \frac{(5n+11)n}{2}.$$

La première ligne est optionnelle et en complexité linéaire.

Donc l'algorithme `triSelection` a une complexité quadratique $O(n^2)$.

Exercice 5 :

On associe à un tableau `tab` un tableau de référence `ref`. Proposer une fonction itérative

`triSelectionSelonRef(tab: list, ref: list) -> list`, qui trie le tableau `tab` dans l'ordre croissant selon les valeurs `ref`. Par exemple, `triSelectionSelonRef( ['A','B','C'], [4,9,1])`, renverra `['C','A','B']`. Préciser les variants et invariants de boucles ainsi que la complexité temporelle.

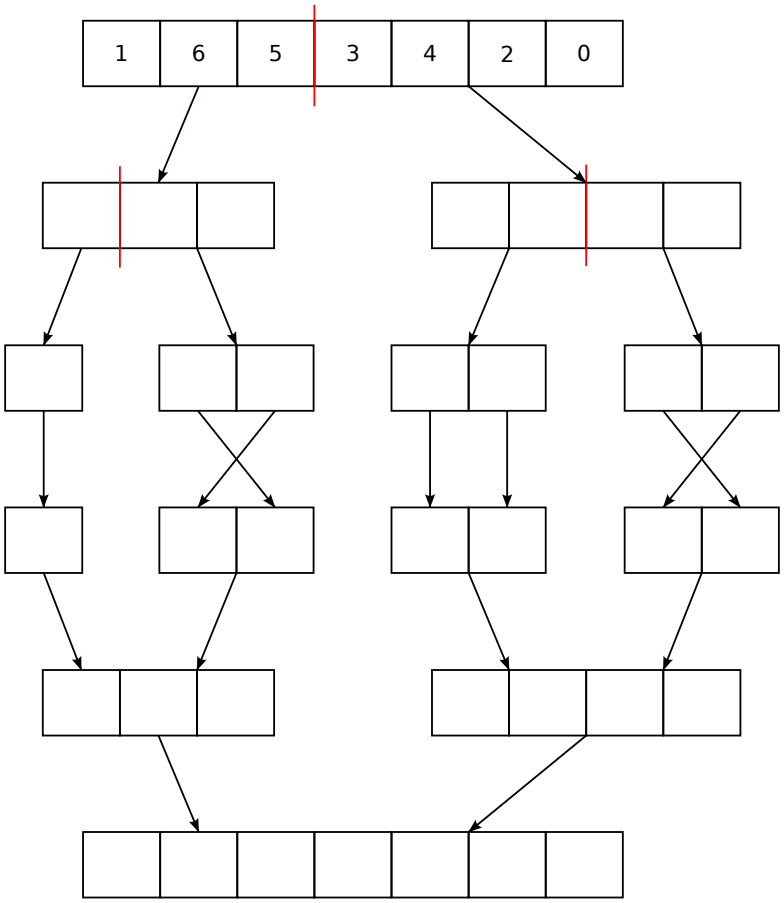
5 Tri fusion (ou tri dichotomique)

5.1 Présentation

Le tri fusion est un tri naturellement récursif, qui utilise la méthode "diviser pour régner". Il consiste à découper le tableau en deux par son milieu, trier chacun des deux sous-tableaux puis réassembler les deux sous-tableaux qui ont été triés en fusionnant des morceaux des deux tableaux pour garder une suite croissante (on prend les éléments du tableau de gauche tant qu'ils sont plus petits que ceux de droite, puis ceux de droite tant qu'ils sont plus petit que les éléments restant à gauche etc).

- Règne : lorsque le tableau possède moins d'un élément, on le renvoie tel quel. Par commodité, on ajoute la terminaison facultative suivante : si le tableau est de taille 2, on le trie.
- Division : on découpe le tableau en deux sous-tableaux de même taille (± 1), puis on fait un appel récursif pour trier chaque sous-tableau
- Assemblage : on assemble les deux sous-tableaux triés en un seul tableau trié en sélectionnant les éléments à droite et à gauche dans l'ordre croissant.

Exemple 5.1 :
Avec `tab=[1,6,5,3,4,2,0]`



Le tableau obtenu `tab=[0,1,2,3,4,5,6]`, est bien trié dans l'ordre croissant.

On propose l'implémentation en python suivant (on rappelle que c'est une fonction naturellement récursive et qu'une copie du tableau est obligatoire).

```python
def triFusion(tab: list) -> list:
    if len(tab) <= 1:
        return tab[:]     # <=> tab.copy()
    m = len(tab)//2
    tabG = triFusion( tab[:m] )
    tabD = triFusion( tab[m:] )
    return assemblage(tabG, tabD)

def assemblage(tab1: list, tab2: list) -> list:
    ntab = []    # nouveau tableau trié
    n1, n2 = len(tab1), len(tab2)
    i1, i2 = 0, 0
    while i1 < n1 and i2 < n2:
        # variant : min(n1-i1, n2-i2)
        # invariant : ntab possède les valeurs des sous-tableaux tab1[:i1] et tab2[:i2] triées
        if tab1[i1] < tab2[i2]:
            ntab.append( tab1[i1] )
            i1 += 1
        else:
            ntab.append( tab2[i2] )
            i2 += 1
    if i1 < n1:              # on rajoute les éléments de tab1 restant éventuels
        ntab += tab1[i1:]
    elif i2 < n2:            # on rajoute les éléments de tab2 restant éventuels
        ntab += tab2[i2:]
    return ntab
```

5.2 Étude

Terminaison

Il n'y a qu'une seule boucle qui est la boucle **while** de la fonction **assemblage**. Le variant de boucle est alors $\min(n_1 - i_1, n_2 - i_2)$ qui est bien strictement décroissant à cause de l'incrémentation obligatoire de i_1 ou de i_2 à chaque étape (c'est le **if/else** qui permet de s'assurer que l'un des deux sera toujours incrémenter). L'algorithme **assemblage** se termine donc bien.

Et donc par suite, **triFusion** se termine aussi.

Correction

Il n'y a de nouveau que la boucle **while** à étudier. L'invariant est indiqué. Ça se démontre par récurrence. Ce n'est ici pas très agréable à écrire à cause des deux indices n_1 et n_2 à gérer.

Complexité temporelle

Soit **n=len(tab)**. La complexité de **triFusion** dépend de celle de **assemblage**. On s'occupe donc dans un premier temps de la complexité de **assemblage**.

Le corps de la boucle **while** contient 9 opérations (3 dans la condition du **if** et 4 dans les deux situations). La condition du **while** contient 3 opérations. Or, on parcourt la boucle **while** au plus n_1+n_2-1 fois. Donc la complexité de **assemblage** est $O(n_1 + n_2)$.

Or, dans la fonction **triFusion**, la fonction **assemblage** est appliquée à deux moitié de tableau de **tab**. Donc $n_1+n_2=n$. Donc, en terme de taille de **tab**, la fonction **assemblage** est de complexité $O(n)$.

Notons $c(n)$ le nombre d'opération de la fonction **triFusion(tab)** avec toujours **n=len(tab)**. Il est facile de voir que $c(1)=c(0)=4$. De plus,

$$\forall p\in\mathbb{N}^*,\ \begin{cases} c(2p)=6+2c(p)+f(p,p)\\ c(2p+1)=6+c(p)+c(p+1)+f(p,p+1)\end{cases}$$

où $f(p,q)$ est le nombre d'opération de **assemblage** sur des listes de longueurs p et q.

Or, on a déjà vu que $f(p,q)=p+q-1$. Donc

$$\forall p\in\mathbb{N}^*,\ \begin{cases} c(2p)=5+2c(p)+2p\\ c(2p+1)=6+c(p)+c(p+1)+2p\end{cases}$$

Supposons que p soit une puissance de 2. Donc calculons $c(2^k)$. $c(2^0)=4$ comme au dessus. Et

$$\forall k\in\mathbb{N},\ c(2^{k+1})=5+2^{k+1}+2c(2^k)\iff \qquad \forall k\in\mathbb{N},\ \frac{c(2^{k+1})}{2^{k+1}}=1+\frac{5}{2^{k+1}}+\frac{c(2^k)}{2^k}.$$

Puis, par sommation et télescopage :

$$\begin{aligned}\forall k\in\mathbb{N}^*,\ \frac{c(2^k)}{2^k}-\frac{c(1)}{1}&=\sum_{i=0}^{k-1}\left(\frac{c(2^{i+1})}{2^{i+1}}-\frac{c(2^i)}{2^i}\right) && \text{télescopage}\\ &=\sum_{i=0}^{k-1}\left(1+\frac{5}{2^{i+1}}\right)\\ &=k+5\frac{1-\frac{1}{2^k}}{1-1/2}\\ &=k+5\left(2-\frac{1}{2^{k-1}}\right).\end{aligned}$$

Et donc

$$\forall k\in\mathbb{N},\ c(2^k)=2^k+k2^k+5(2^{k+1}-2)=(k+11)2^k-10.$$

Montrons maintenant que la suite $(c(n))_{n\in\mathbb{N}}$ est croissante. On a déjà calculé $c(0)=c(1)=4$. Donc $c(0)\leq c(1)$.

Soit $n\in\mathbb{N}$. Supposons $\forall k\in\{0,\dots,n\}$, $c(k)\leq c(k+1)$.

- Si $n+1\equiv 0\ [2]$. Alors $\exists p\in\mathbb{N}^*$ tel que $n+1=2p$. Alors

$$c(n+2)-c(n+1)=c(2p+1)-c(2p)=1+c(p+1)-c(p)$$

  d'après les calculs précédents.

- Si $n+1\equiv 1\ [2]$. Alors $\exists p\in\mathbb{N}$ tel que $n+1=2p+1$. Et donc

$$c(n+2)-c(n+1)=c(2p+2)-c(2p+1)=1+c(p+1)-c(p).$$

Donc, dans les deux cas, $\exists p \in \{0, \dots, n\}$, tel que $c(n+2) - c(n+1) = 1 + c(p+1) - c(p)$. Par hypothèse de récurrence forte, on a donc $c(n+2) - c(n+1) \geq 1$.

Et donc, par principe de récurrence forte, $\forall n \in \mathbb{N}$, $c(n+1) \geq c(n)$. Donc $(c(n))_{n\in\mathbb{N}}$ est croissante.

Comme la suite $(2^k)_{k\in\mathbb{N}}$ est une suite strictement croissante, alors $\forall n \in \mathbb{N}$, $\exists! k \in \mathbb{N}$ tel que $2^k \leq n < 2^{k+1}$. Et donc, $c(2^k) \leq c(n) \leq c(2^{k+1})$. De plus, $k = \lfloor \ln(n)/\ln(2) \rfloor = \lfloor \log_2(n) \rfloor$. Donc

$$\forall n \in \mathbb{N},\ (k+11)2^k - 10 \leq c(n) \leq (k+12)2^{k+1} - 10.$$

Et donc,

$$c(n) \underset{n\to+\infty}{=} O(n \log_2(n)).$$

Finalement, la complexité de `triFusion` est quasi-linéaire en $O(n \log_2(n))$.

Exercice 6 :
Proposer un déroulé du tri fusion pour le tableau `[4,2,2,8,1,5,4,8,3,0]`.

6 Récapitulatif

Chaque tri à ses avantages et ses inconvénients. La connaissance exacte des performances de chaque tris en fonction de la situation n'est pas nécessaire. Mais il faut garder en mémoire les domaines d'applications de chacun, ainsi que leur complexité.

Tris	Restrictions de types	Type d'algorithme	Complexité	Description
Comptage	Entiers ou s'y ramenant	Itératif	$O(n)$ Linéaire	Compte le nombre d'éléments de chaque valeur
Tri à bulles	Objets avec ordre (`str`, `float`, ...)	Itératif	$O(n^2)$ Quadratique	Faire remonter la plus grande valeur
Tri par insertion	Objets avec ordre (`str`, `float`, ...)	Itératif	$O(n^2)$ Quadratique	Insérer chaque nouvelle valeur scannée dans la partie de gauche déjà triée
Tri par sélection	Objets avec ordre (`str`, `float`, ...)	Itératif	$O(n^2)$ Quadratique	Pour chaque position, sélection du minimum de la droite et le placer à gauche
Tri fusion	Objets avec ordre (`str`, `float`, ...)	Récursif	$O(n \log_2(n))$ Quasi-linéaire	Découper le tableau en deux, puis les fusionne en choisissant l'ordre des éléments

7 Autres types de tris

Il existe beaucoup d'autres types de tris. Chacun ayant leurs avantages et leurs inconvénients. On peut citer notamment :

- Le tri à pierre : c'est la version "alourdi" du tri à bulles.
- Le tri rapide
- Le tri par tas
- Le tri Timsort
- Le tri cocktail
- ...