6.2 Le tri-fusion

NSI terminale - JB Duthoit

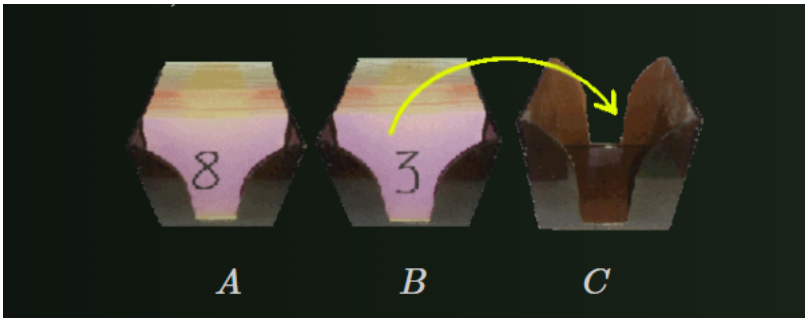
6.2.1 Trier la concaténation de deux listes triées

Principe

On se donne deux listes triées A et B et on souhaite obtenir une liste triée C qui contient l'intégralité des termes présents dans ces deux listes. Une première idée naïve consisterait à trier la concaténation A.B de ces deux listes avec un algorithme de tri de son choix. Cette méthode ne faisant aucun cas de la croissance des deux listes initiales, la complexité de ce tri serait celle de l'algorithme de tri choisi.
On peut faire certainement faire mieux en exploitant cette croissance des deux listes

L'algorithme de fusionnement efficace est celui qui tombe sous le sens dès que l'on réfléchit quelques instants à un moyen économique de constituer une seule liste triée à partir de deux listes triées. C'est la réponse à un problème que tout le monde a été amené à résoudre dans le monde physique. On dispose de deux boites contenant des fiches triées dans l'ordre croissant dont on ne voit que la première fiche :

Fusion de deux listes triées A et B dans une liste C



Pour trier l'ensemble des valeurs, il suffit de comparer le premier terme de chacune des deux listes A et B, d'extraire le plus petit des deux de sa boite/liste et de l'insérer à la fin d'une liste auxiliaire C, puis répéter cette opération jusqu'à ce que tous les éléments des deux boites/listes aient été traités.

Il n'est pas difficile de démontrer que la complexité de cet algorithme est linéaire en le nombre d'éléments contenus dans ces deux listes.

Un exemple

La fusion des 2 tableaux déjà triés est assez simple, prenons comme exemple la fusion entre le tableau [4, 12, 23, 56] et le tableau [3, 32, 35, 42, 57] :

Soit C le tableau issu de la fusion du tableau A = [4, 12, 23, 56] et du tableau B = [3, 32, 35, 42, 57].

- On considère le premier élément du tableau A (4) et le premier élément du tableau B (3) : 3 est inférieur à 4, on place 3 dans le tableau C et on le supprime du tableau B. Nous avons donc alors C = [3], A = [4, 12, 23, 56] et B = [32, 35, 42, 57].
- On recommence ensuite à comparer le premier élément du tableau A (4) et le premier élément du tableau B (32) : 4 est inférieur à 32, on place 4 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4], A = [12, 23, 56] et B = [32, 35, 42, 57].
- On compare le premier élément du tableau A (12) et le premier élément du tableau B (32) : 12 est inférieur à 32, on place 12 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4, 12], A = [23, 56] et B = [32, 35, 42, 57].
- On compare le premier élément du tableau A (23) et le premier élément du tableau B (32) : 23 est inférieur à 32, on place 23 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4, 12, 23], A = [56] et B = [32, 35, 42, 57].
- On compare le premier élément du tableau A (56) et le premier élément du tableau B (32) : 32 est inférieur à 56, on place 32 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4, 12, 23, 32], A = [56] et B = [35, 42, 57].
- On compare le premier élément du tableau A (56) et le premier élément du tableau B (35) : 35 est inférieur à 56, on place 35 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4, 12, 23, 32, 35], A = [56] et B = [42, 57].
- On compare le premier élément du tableau A (56) et le premier élément du tableau B (35) : 35 est inférieur à 56, on place 35 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4, 12, 23, 32, 35], A = [56] et B = [42, 57].
- On compare le premier élément du tableau A (56) et le premier élément du tableau B (42) : 42 est inférieur à 56, on place 42 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4, 12, 23, 32, 35, 42], A = [56] et B = [57].
- On compare le premier élément du tableau A (56) et le premier élément du tableau B (57) : 56 est inférieur à 57, on place 56 dans le tableau C et on le supprime du tableau A. Nous avons donc alors C = [3, 4, 12, 23, 32, 35, 42, 56], A = [] et B = [57].
- Le tableau A est vide, il nous reste juste à placer le seul élément qui reste dans B (57) dans C : C = [3, 4, 12, 23, 32, 35, 42, 56, 57], A = [] et B = [].

La fusion est terminée.

Exercice 6.3

Reprenez tout le raisonnement qui vient d'être fait sur les tableaux $A = [1, 2, 33, 444]$ et $B = [5, 6, 77, 78, 79, 80]$.

Exercice 6.4

- Écrire un algorithme itératif qui définit une fonction *fusion_ iteratif* qui prend deux listes triées A et B en entrée, et qui renvoie la liste C, résultant de la concaténation des deux listes A et B, triée. On utilisera un procédé itératif.
- Implémenter cet algorithme en Python.

***

Exercice 6.5

Écrire un algorithme récursif qui définit une fonction *fusion* qui prend deux listes triées A et B en entrée, et qui renvoie la liste C, résultant de la concaténation des deux listes A et B, triée. Implémenter cet algorithme en Python.
***

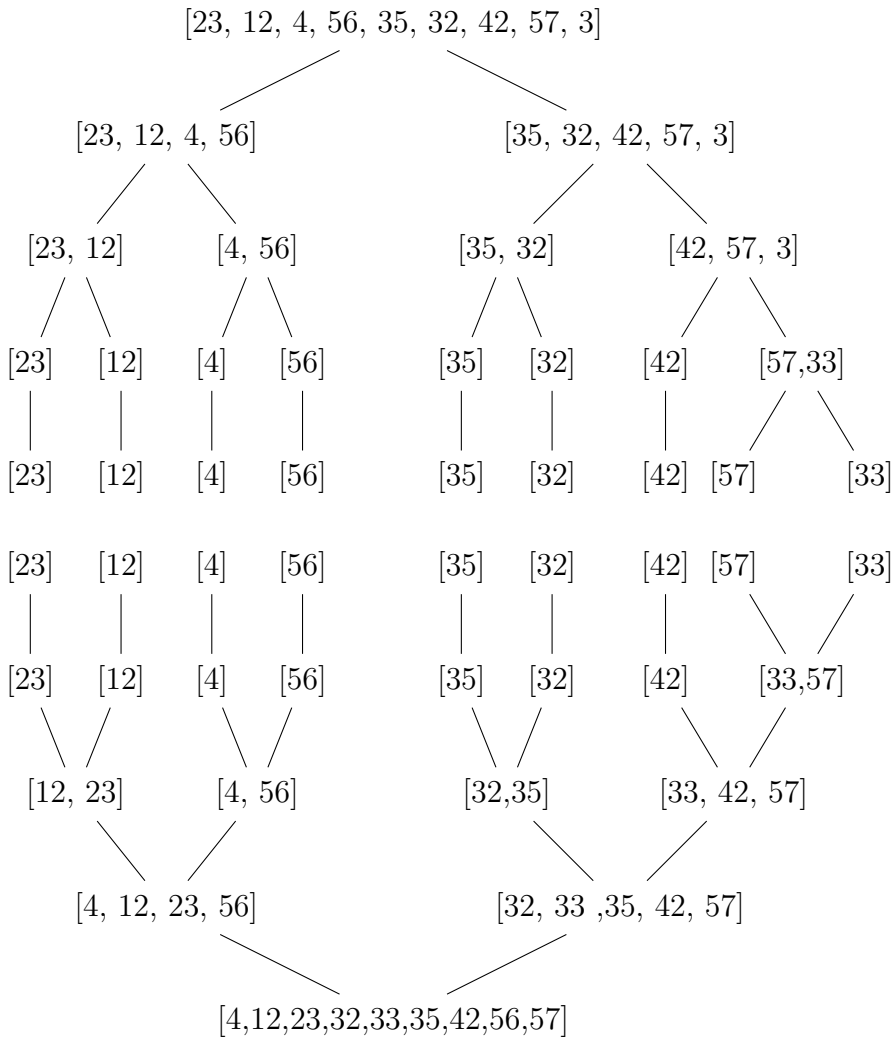
Remarque

Cet exercice est utile uniquement afin de s'entraîner sur la récursivité. On constatera que l'écriture récursive n'apporte aucun bénéfice, même pas sur la forme du code.
Dans la suite du tri-fusion, c'est bien la version itérative de la fusion que nous allons utiliser

6.2.2 Le tri fusion

Exemple

Prenons comme exemple :
C = [23, 12, 4, 56, 35, 32, 42, 57, 3] que l'on veut trier par un tri-fusion.



Exercice 6.6
Faire un tel schéma avec la liste [5,6,1,3,2,7]

Diviser pour régner appliqué au tri fusion

- DIVISER : Si milieu est le point milieu entre debut et fin, alors nous pouvons diviser le sous-tableau T[debut..fin] en deux tableaux T[debut..milieu] et T[milieu + 1, fin].

- RÉGNER : Dans l'étape Régner, nous essayons de trier les sous-réseaux T[debut..milieu] et T[milieu + 1, fin]. Si nous n'avons pas encore atteint le cas de base (le sous-tableau contient un seul élément), nous divisons à nouveau ces deux sous-réseaux et essayons de les trier.

- COMBINER : Lorsque l'étape de conquête atteint l'étape de base et que nous obtenons deux sous-tableaux triés T[debut..milieu] et T[milieu + 1, fin] pour le tableau T[debut..milieu], nous combinons les résultats en créant un tableau trié T[debut..milieu] à partir de deux sous-réseaux triés T[debut..milieu] et T[milieu + 1, fin].

Un algorithme du tri-fusion

```
1  VARIABLE
2  T : liste d'entiers
3  long(T) : longueur de T
4  DEBUT
5  Fonction tri_fusion(T)
6  |  SI long(T) <= 1 ALORS
7  |  |  Retourner T
8  |  milieu = long(T) // 2
9  |  T1 = T[0 :milieu] #slicing
10 |  T2 = T[milieu :] #slicing
11 |  Retourner fusion_iteratif (tri_fusion(T1),tri_fusion(T2))
12 end
```

⚠ Le slicing, bien que très pratique, n'est pas explicitement dans les programmes de NSI. Il peut être du coup très intéressant de trouver une solution alternative aux lignes 10 et 11.

```
1  VARIABLE
2  T : liste d'entiers
3  long(T) : longueur de T
4  DEBUT
5  Fonction tri_fusion(T)
6  |  SI long(T) <= 1 ALORS
7  |  |  Retourner T
8  |  n = len(T)
9  |  milieu = long(T) // 2
10 |  q = n - milieu
11 |  T1 = [0]*milieu
12 |  T2 = [0]*q
13 |  POUR i DE 0 À (milieu-1) FAIRE
14 |  |  T1[i] ← T[i]
15 |  POUR i DE 0 À (q-1) FAIRE
16 |  |  T2[i] ← T[i+q]
17 |  Retourner fusion_iteratif (tri_fusion(T1),tri_fusion(T2))
18 end
```

ou bien encore :

```
1  VARIABLE
2  T : liste d'entiers
3  long(T) : longueur de T
4  DEBUT
5  Fonction tri_fusion(T)
6  |  SI long(T) <= 1 ALORS
7  |  |  Retourner T
8  |  n = len(T)
9  |  milieu = long(T) // 2
10 |  T1 = []
11 |  T2 = []
12 |  POUR i DE 0 À n-1 FAIRE
13 |  |  SI i < milieu ALORS
14 |  |  |  T1.append(T[i])
15 |  |  SINON
16 |  |  |  T2.append(T[i])
17 |  Retourner fusion_iteratif (tri_fusion(T1),tri_fusion(T2))
18 end
```

Exercice 6.7

Implémenter cet algorithme en Python.

***

Remarque

Ce n'est pas, ici, un tri en place : on a utilisé une copie (partielle) pour scinder la liste en 2.

6.2.3 Tri fusion pour une liste chainée

On suppose qu'on dispose de l'objet Maillon qui permet de créer des listes chaînée.

Exercice 6.8

1. une fonction `vers_liste(lst)` qui prend en paramètre une liste chainée `lst` et qui renvoie un tableau dynamique python pour faciliter les tests.
2. Créer une fonction **coupe(lst)** de paramètre `lst` une liste chainée et qui renvoie deux listes issues de `lst` ayant le même nombre d'éléments, à un près.
3. Créer une fonction **fusion(l1,l2)** de paramètres `l1` et `l2` deux listes chaînées triées et qui renvoie la fusion triée issue de la concaténation des deux listes `l1` et `l2`
4. Créer la fonction **tri_ fusion(lst)** de paramètre une liste chaînée `lst` et qui renvoie la liste chaînée triée, en utilisant les deux fonctions précédentes.

☛ Pour ces questions, vous pouvez utiliser le fichier annexe où vous trouverez quelques tests :

```
class Maillon:
    def __init__(self, valeur, suivante=None):
        self.valeur = valeur
        self.suivante = suivante



# Question 1 : Conversion vers une liste Python (pour
  faciliter les tests)

#def vers_liste(lst):
    """Prend en paramètre une liste chaînée lst et renvoie un
    tableau dynamique Python."""
    # A completer :
    pass


# Question 2 : Découpage d'une liste chaînée en deux sous-
  listes

def coupe(lst):
    """Renvoie deux listes issues de lst ayant le même nombre
    d'éléments, à un près."""
    l1, l2 = None, None
    # A completer :
    while ................:
        .....
        .....
        .....
    return l1, l2


# Question 3 : Fusion de deux listes chaînées triées

def fusion(l1, l2):
```

```
    """Renvoie la fusion triée issue des deux listes chaînées
    triées l1 et l2."""
    # A completer :
    if l1 is None:
        return ....
    if l2 is None:
        return ....
    if ......................:
        return ........................
    else:
        return ............


# Question 4 : Tri fusion d'une liste chaînée


def tri_fusion(lst):
    """Renvoie la liste chaînée lst triée par la méthode du
   tri fusion."""
    # A completer :
    if lst is None or lst.suivante is None:
        return ....
    else:
        l1, l2 = .........
        return fusion(............, ............)
```

***

6.2.4 Un autre algorithme qui permet de trier en place

Exercice 6.9

- Créer en pseudo langage une fonction *inserer_ element_ position(tab,element,p)* qui :
  - a pour paramètre :
    - `tab` : tableau dynamique
    - `element` : entier à insérer
    - `p` : indice sur lequel il faut insérer l'élément
  - modifie en place la liste
- Implémenter cet algorithme en Python

***

Exercice 6.10

- Créer en pseudo langage une fonction *suppression_ element_ position(tab,p* qui :
  - a pour paramètre :
    - `tab` : tableau dynamique
    - `p` : indice sur lequel il faut supprimer l'élément

  - modifie en place la liste

- Implémenter cet algorithme en Python

***

Exercice 6.11

Créer une fonction *fusion_ avec_ indice(l,p,q,r)* qui trie en place de l'indice `p` à l'indice `r` inclus. Voici le prototype de cette fonction :

```
def fusion_avec_indice(l,p,q,r):
    '''
    l est une liste
    les termes dt les indices sont entre p et q sont triées
    les termes dt les indices sont entre q+1 et r sont triées
    '''
```

Avec ces deux fonctions, il est maintenant facile de créer la fonction *tri_ fusion_ avec _ indice(l,p,r)* :

```
def tri_fusion_avec_indice(l,p,r):
    if p < r: # ce qui signifie que l liste comporte au moins 2 éléments
        milieu = (p + r) // 2
        tri_fusion_avec_indice(l,p,milieu)
        tri_fusion_avec_indice(l,milieu+1,r)
        fusion_avec_indice(l,p,milieu,r)
    return l

def tri_fusion2(l):
    if len(l) <= 1:
        return l
    milieu = len(l) // 2
    tri_fusion2(l[:milieu])
    tri_fusion2(l[milieu:])
    return fusion_iteratif(tri_fusion2(l[:milieu]),tri_fusion2(l[milieu:]))
```

6.2.5 Complexité du tri-fusion

Étudions la complexité pour *tri_fusion* :

Supposons d'abord que la taille du tableau initial est $N = 2^n$ une puissance de 2. On a donc alors (complexité en nombre de tests) :

$$C(N) = \begin{cases} 1 \; si \; N = 1 \; (n = 0) \\ 2C(N/2) + N \; sinon \end{cases}$$

(deux appels récursifs avec des tableaux de taille divisée par 2, plus le coût de la fusion).
Si on réécrit :

$$C(2^n) = \begin{cases} 1 \; si \; n = 0 \\ 2C(2^n/2) + 2^n \; sinon \end{cases}$$

Soit :

$$C(2^n) = \begin{cases} 1 \; si \; n = 0 \\ 2C(2^{n-1}) + 2^n \; sinon \end{cases}$$

Ou encore, si on divise par 2^n :

$$\frac{C(2^n)}{2^n} = \begin{cases} 1 \; si \; n = 0 \\ \frac{2C(2^{n-1})}{2^n} + \frac{2^n}{2^n} \; sinon \end{cases}$$

Soit :

$$\frac{C(2^n)}{2^n} = \begin{cases} 1 \; si \; n = 0 \\ \frac{C(2^{n-1})}{2^{n-1}} + \frac{2^n}{2^n} \; sinon \end{cases}$$

On pose $u_n = \frac{C(2^n)}{2^n}$:

$$u_n = \begin{cases} 1 \; si \; n = 0 \\ u_{n-1} + 1 \; sinon \end{cases}$$

(u_n) est une suite arithmétique de premier terme $u_0 = 1$ et de raison 1, donc $u_n = 1 + n$.

On a donc

$$\frac{C(2^n)}{2^n} = 1 + n$$

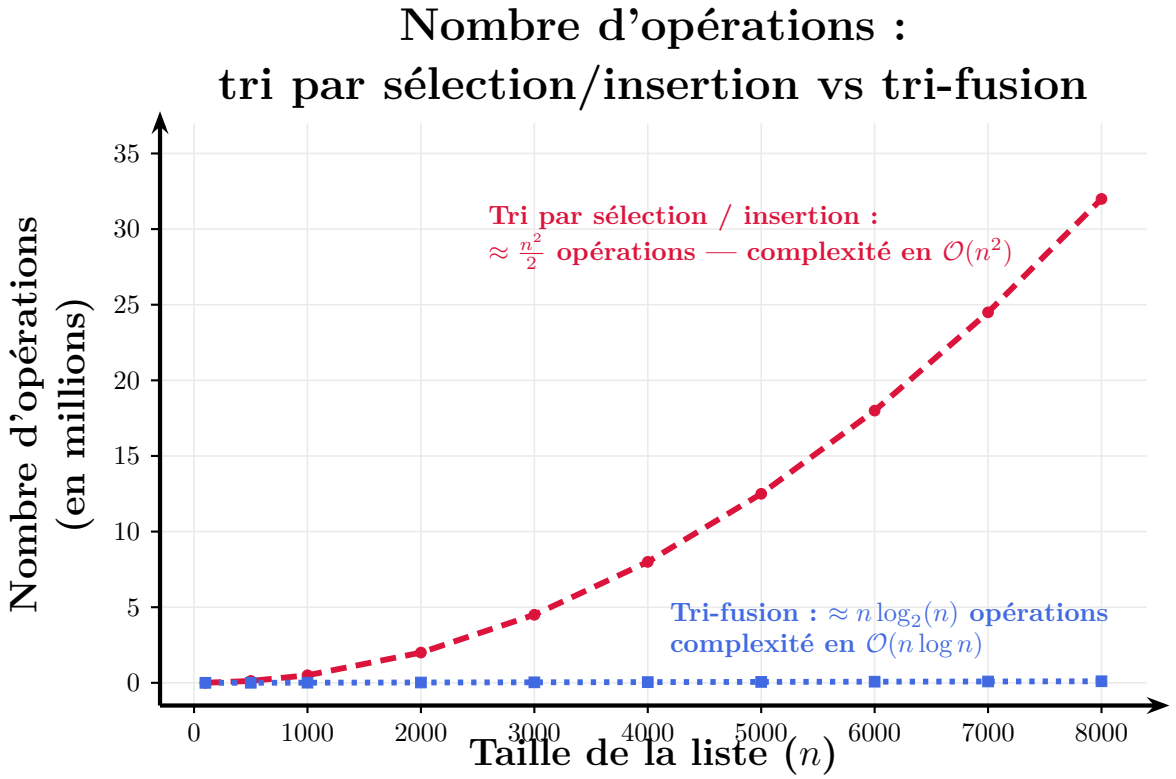
Soit

$$C(2^n) = 2^n + n \times 2^n$$

On repasse en "N" :

$$C(N) = N + log_2(N) \times N$$

On arrive finalement à une complexité en $O(Nlog_2(N))$.



Exercice 6.12

Vérifier l'intérêt d'avoir une complexité en $O(nlog_2(n))$ au lieu de $O(n^2)$, en remplissant le tableau suivant (les résultats seront exprimés en seconde) :

longueur de la liste	Tri par sélection	Tri par fusion
100		
1000		
10 000		
20 000		
40 000		

☞ Avec une complexité en $O(n^2)$, si on double la taille de la liste, on va multiplier la durée par 4 (2^2).
Avec une complexité en $O(log_2(n))$, si on double la taille de la liste, on va multiplier la durée par 2 ($2 \times log_2(2) = 2$).

Remarque

$log_2(1) = 0$, $log_2(2) = 1$, $log_2(4) = 2$, $log_2(8) = 3$, $log_2(16) = 4$ ...

Exercice 6.13

Extrapoler ces données pour compléter le tableau avec 10 240 000 éléments (convertir dans une unité facilement interprétable) :

longueur de la liste	Tri par sélection	Tri par fusion
10 240 000		

***