Chapitre 8
Tri Fusion

Le **tri fusion** (en anglais *merge sort*) est un algorithme de tri par comparaison efficace. Il se base sur le principe du « **diviser pour régner** » (*divide and conquer* en anglais) : on décompose un problème en sous-problèmes plus simples, on résout ces sous-problèmes, puis on combine leurs solutions.

I. La méthode « diviser pour régner »

Définition 8.1.
La méthode « **diviser pour régner** » consiste à :
- **diviser** : découper le problème initial en sous-problèmes plus petits, de même nature ;
- **résoudre** : résoudre chacun des sous-problèmes (directement s'il est trivial, ou récursivement sinon) ;
- **combiner** : assembler les solutions des sous-problèmes pour obtenir la solution du problème initial.

■ **Exemple 8.1.**
Chercher un mot dans un dictionnaire : on ouvre le dictionnaire au milieu, on compare le mot cherché avec le mot de la page, et l'on ne garde que la moitié du dictionnaire où le mot peut se trouver. On répète l'opération sur cette moitié, jusqu'à trouver le mot. On a « divisé » le problème (rechercher dans tout le dictionnaire devient rechercher dans une moitié), puis on recommence sur le sous-problème restant.
■

■ **Exemple 8.2.**
Ranger une pièce en désordre : on peut diviser le travail en deux moitiés de pièce, ranger chaque moitié séparément (éventuellement en divisant encore), puis vérifier que l'ensemble est rangé. Diviser le travail ne le rend pas plus long ici, mais permet de le répartir. ■

Le tri fusion applique cette méthode au tri d'un tableau. L'algorithme se compose de deux parties distinctes :

Définition 8.2.
Le **tri fusion** d'une liste T repose sur deux phases :
- **Diviser** : on divise récursivement le tableau, c'est-à-dire qu'on le coupe en deux sous-tableaux et l'on recommence cette opération sur ces derniers, jusqu'à ce que chaque sous-tableau ne contienne plus qu'un seul élément.
- **Fusionner** : une fois le tableau divisé en N sous-tableaux (N étant le nombre d'éléments), on fusionne deux à deux les tableaux en réorganisant les éléments dans l'ordre du tri (croissant ou décroissant).

R L'intérêt de diviser pour ensuite fusionner est que créer un tableau trié à partir de deux sous-tableaux *déjà triés* peut s'effectuer en temps linéaire : il suffit de comparer les têtes des deux sous-tableaux et de placer la plus petite, encore et encore. C'est ce point en particulier qui fait la rapidité du tri fusion.

II. Un exemple pas à pas

Exemple 8.3.
1re étape : diviser. Prenons comme exemple la suite de nombres $5, 1, 3, 8, 9, 6$ que l'on veut trier avec le tri fusion dans l'ordre croissant. On divise le tableau en deux, puis on divise encore les sous-tableaux obtenus :
- $5, 1, 3 \mid 8, 9, 6$: on divise le tableau en deux ;
- $5 \mid 1, 3 \mid 8 \mid 9, 6$: on divise en deux les sous-tableaux ;
- $5 \mid 1 \mid 3 \mid 8 \mid 9 \mid 6$: chaque sous-tableau est de nouveau divisé pour n'avoir plus qu'un seul élément.

■

Exemple 8.4.
2e étape : fusionner. On repart des six sous-tableaux à un seul élément et l'on fusionne deux à deux les sous-tableaux adjacents, en réorganisant les éléments dans l'ordre croissant :
- $1, 5 \mid 3, 8 \mid 6, 9$: on prend deux sous-tableaux adjacents que l'on fusionne en les ordonnant ;
- $1, 3, 5, 8 \mid 6, 9$: on continue la fusion des sous-tableaux ;
- $1, 3, 5, 6, 8, 9$: le tableau ne contient plus de sous-tableaux, il est donc trié.

■

R Sur la figure d'origine de cet exemple, les éléments en bleu correspondent à l'état du tableau après la première étape (la division), et les éléments en vert après la deuxième étape (la fusion). On peut vérifier que, à chaque fusion, les deux sous-tableaux fusionnés sont déjà triés : c'est ce qui rend la fusion rapide.

III. Fusionner deux listes triées

R Rappel de vocabulaire : une liste est **triée** (par ordre croissant) lorsque chaque élément est inférieur ou égal au suivant, c'est-à-dire que pour tout i, $T[i] \leqslant T[i+1]$. La **tête** d'une liste est son premier élément.

■ **Exemple 8.5.**
On veut fusionner les deux listes triées $[1, 3, 5]$ et $[2, 4, 6]$. On compare à chaque fois les deux têtes, on place la plus petite dans la liste résultat, puis on avance dans la liste d'où vient l'élément placé :

Comparaison	Élément placé	Liste résultat
$1 \leqslant 2$	1	$[1]$
$3 > 2$	2	$[1, 2]$
$3 \leqslant 4$	3	$[1, 2, 3]$
$5 > 4$	4	$[1, 2, 3, 4]$
$5 \leqslant 6$	5	$[1, 2, 3, 4, 5]$
fin de la deuxième liste	6	$[1, 2, 3, 4, 5, 6]$

La liste résultat contient exactement les éléments des deux listes de départ, dans l'ordre croissant. ■

■ **Exemple 8.6.**
Si l'une des deux listes est vide, la fusion est immédiate : fusionner [] et $[2, 4]$ donne $[2, 4]$, et fusionner $[1, 3]$ et [] donne $[1, 3]$. C'est ce cas de base qui arrête la récursivité de la fonction de fusion. ■

Voici le pseudo-code de la fusion de deux listes triées A et B :

```
entree : deux tableaux tries A et B
sortie : un tableau trie contenant exactement les elements de A et de B

fonction fusion(A[1..a], B[1..b])
    si A est vide :
        renvoyer B
    si B est 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])
```

■ **Exemple 8.7.**
Le pseudo-code de la fusion se traduit directement en Python par une fonction récursive : à chaque appel, on compare les deux têtes, on place la plus petite, puis on rappelle la fonction sur les listes restantes. ■

```
def fusion(A, B):
    """Renvoie la liste triee contenant les elements de A et de 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(A, B[1:])
```

R La fusion de deux listes de tailles a et b a un coût **linéaire** : à chaque étape on place un élément dans la liste résultat, donc au plus $a+b$ étapes au total. Le coût est proportionnel à $a+b$, ce que l'on note $O(a+b)$.

R La version récursive ci-dessus découpe les listes avec l'écriture $A[1:]$, ce qui recopie la liste restante à chaque appel. C'est simple à écrire et à lire, mais pour de très grandes listes, on préférera une version itérative qui parcourt les deux listes avec des indices i et j sans faire de recopie (voir les exercices).

IV. L'algorithme de tri fusion

Définition 8.3.
Le **tri fusion** est un algorithme récursif : pour trier une liste T de taille n,
- si $n \leqslant 1$, la liste est déjà triée : on la renvoie telle quelle ;
- sinon, on divise la liste en deux moitiés (de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$), on trie chacune des deux moitiés en appelant récursivement le tri fusion, puis on fusionne les deux moitiés triées avec la fonction `fusion`.

■ **Exemple 8.8.**
On applique le tri fusion à la liste $[4,2,5,1]$:
- Division : $[4,2,5,1] \to [4,2]$ et $[5,1] \to [4]$, $[2]$, $[5]$, $[1]$;
- Fusion : $[4]$ et $[2]$ donnent $[2,4]$; $[5]$ et $[1]$ donnent $[1,5]$;
- Fusion finale : $[2,4]$ et $[1,5]$ donnent $[1,2,4,5]$.

La liste est triée après deux niveaux de division, car $\log_2 4 = 2$. ■

Voici le pseudo-code du tri fusion :

```
entree : un tableau T
sortie : une permutation triee de T

fonction triFusion(T[1..n])
    si n <= 1 :
        renvoyer T
    sinon :
        renvoyer fusion(triFusion(T[1..n//2]),
                        triFusion(T[n//2 + 1..n]))
```

■ **Exemple 8.9.**
Le pseudo-code du tri fusion se traduit en Python en une fonction récursive qui utilise la fonction `fusion` : on coupe la liste en deux, on trie chaque moitié par un appel récursif, puis on fusionne les deux moitiés triées. ■

```
def tri_fusion(T):
    """Renvoie une copie triee de la liste T."""
    n = len(T)
    if n <= 1:
        return T
    milieu = n // 2
    gauche = tri_fusion(T[:milieu])
    droite = tri_fusion(T[milieu:])
    return fusion(gauche, droite)
```

R Le tri fusion est un algorithme **stable** : deux éléments égaux conservent leur ordre relatif après le tri. C'est une conséquence du choix $A[1] \leqslant B[1]$ dans la fusion : en cas d'égalité, on place d'abord l'élément de A, c'est-à-dire celui qui était le plus à gauche.

V. Complexité du tri fusion

Pour évaluer le coût du tri fusion, on compte les opérations (essentiellement les comparaisons) en fonction de la taille n de la liste.

1. **Coût d'une fusion** : fusionner deux listes de tailles totales n coûte $O(n)$, comme on l'a vu.
2. **Nombre de niveaux de division** : à chaque niveau, chaque liste est coupée en deux ; après k niveaux, les listes ont une taille d'environ $n/2^k$. On atteint des listes d'un seul élément quand $2^k \approx n$, c'est-à-dire après environ $\log_2 n$ niveaux.
3. **Coût total** : à chaque niveau, la somme des tailles de toutes les listes vaut n, donc chaque niveau coûte $O(n)$. Il y a $\log_2 n$ niveaux, donc le coût total est $O(n \log_2 n)$.

Propriété 8.1.
Le **tri fusion** a un coût en $O(n \log_2 n)$ comparaisons dans le pire des cas, pour une liste de n éléments. C'est un coût bien meilleur que le coût quadratique $O(n^2)$ des tris par insertion et par sélection vus en Première.

■ **Exemple 8.10.**
Pour $n = 8$ éléments, la division produit $\log_2 8 = 3$ niveaux (8, puis 4, puis 2, puis 1 éléments par liste). Chaque niveau coûte environ 8 opérations : le tri fusion effectue de l'ordre de $8 \times 3 = 24$ opérations, là où un tri quadratique en ferait de l'ordre de $8^2 = 64$. ■

■ **Exemple 8.11.**
L'écart devient spectaculaire quand n grandit. Le tableau suivant compare n^2 et $n \log_2 n$ pour différentes tailles :

Taille n	n^2	$n \log_2 n$
10	100	≈ 33
100	10 000	≈ 664
1 000	1 000 000	≈ 9966
10 000	100 000 000	≈ 132877
100 000	10 000 000 000	≈ 1660964

Pour 100 000 éléments, un tri quadratique demande environ dix milliards d'opérations, contre moins de deux millions pour le tri fusion : c'est l'intérêt du passage de n^2 à $n \log_2 n$ mentionné au programme. ∎

Algorithme	Méthode	Coût dans le pire des cas
Tri par insertion	vu en Première	$O(n^2)$
Tri par sélection	vu en Première	$O(n^2)$
Tri fusion	diviser pour régner	$O(n \log_2 n)$

R Contrairement au tri par insertion, qui est très rapide sur une liste presque triée, le tri fusion effectue toujours environ le même nombre d'opérations : le pire cas et le meilleur cas sont tous les deux en $O(n \log_2 n)$. Une liste déjà triée n'est pas traitée plus vite qu'une liste aléatoire.

R La version du tri fusion présentée ici crée de nouvelles listes à chaque découpe et à chaque fusion : son coût **en mémoire** est en $O(n)$ (on conserve au plus deux moitiés de tailles totales n à chaque niveau). C'est le prix à payer pour sa rapidité en temps.

VI. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à écrire des programmes propres et à tester vos fonctions sur de petits exemples.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : vocabulaire du tri fusion, lecture de code court, ordres de grandeur. Chaque question doit pouvoir être traitée en moins de trente secondes.

Exercice 8.1.
Dire si les affirmations suivantes sont vraies ou fausses.

1. Le tri fusion est un tri par comparaison.
2. La méthode « diviser pour régner » consiste à couper le problème en sous-problèmes, à les résoudre, puis à combiner leurs solutions.
3. Fusionner deux listes triées peut se faire en temps linéaire, proportionnel à la somme de leurs tailles.
4. Le tri fusion a un coût quadratique $O(n^2)$ dans le pire des cas.
5. Le tri fusion est un algorithme récursif.

■

Exercice 8.2.
On suppose que la fonction `fusion` du cours est correctement implémentée. Que renvoient les appels suivants ?

```
fusion([1, 2], [3, 4])
fusion([3, 4], [1, 2])
fusion([], [5, 6])
fusion([7], [7])
```

■

Exercice 8.3.
Rappel de Première : dire si les affirmations suivantes sont vraies ou fausses.

1. Le tri par insertion insère chaque élément à sa place dans la partie déjà triée de la liste.
2. Le tri par sélection cherche le minimum restant et le place au début de la partie non triée.
3. Le coût du tri par insertion dans le pire des cas est quadratique.
4. Le tri fusion est plus rapide que le tri par insertion pour de grandes listes.

■

■ **Exercice 8.4.**
Compléter le pseudo-code de la fusion avec les morceaux manquants (indiqués par des pointillés).

```
fonction fusion(A, B)
    si A est vide : renvoyer ...
    si B est vide : renvoyer ...
    si A[0] <= B[0] :
        renvoyer A[0] + fusion(..., B)
    sinon :
        renvoyer B[0] + fusion(A, ...)
```

■

■ **Exercice 8.5.**
Pour une liste de 16 éléments, combien de niveaux de division successifs faut-il pour arriver à des sous-listes d'un seul élément ? Même question pour une liste de 1024 éléments. ■

Exercices d'application

Les exercices d'application demandent d'écrire des fonctions complètes et de les tester sur des exemples.

■ **Exercice 8.6.**
Implémenter en Python une fonction récursive `fusion` de fusion de deux listes triées : elle prend en entrée deux listes triées A et B et renvoie la liste triée contenant exactement leurs éléments. On suivra le pseudo-code du cours.

```
def fusion(A, B):
    if A == []:
        return B
    ...
```

■

■ **Exercice 8.7.**
Écrire une version **itérative** de la fusion, `fusion_iterative`, qui parcourt les deux listes avec deux indices i et j (sans recopier les listes avec des découpes $A[1:]$), et qui renvoie la liste triée contenant tous les éléments. On rappelle que `res.append(x)` ajoute x en fin de liste et que `res.extend(L)` ajoute tous les éléments de L.

```
def fusion_iterative(A, B):
    i, j = 0, 0
    res = []
    ...
```

■

■ **Exercice 8.8.**
Implémenter en Python une fonction récursive `tri_fusion` qui trie une liste par l'algorithme du tri fusion. Il faudra utiliser la fonction `fusion` écrite dans l'exercice précédent (ou la fonction du cours).

```
def tri_fusion(T):
    n = len(T)
    ...
```

■

■ **Exercice 8.9.**
Appliquer à la main l'algorithme du tri fusion à la liste $[3, 1, 4, 1, 5, 9, 2, 6]$: écrire toutes les étapes de division, puis toutes les étapes de fusion, comme dans l'exemple du cours. ■

■ **Exercice 8.10.**
Adapter la fonction `fusion` du cours pour trier par ordre **décroissant** : il suffit de modifier la comparaison. Vérifier sur la liste $[5, 1, 3, 8, 9, 6]$. Faut-il modifier autre chose dans `tri_fusion` ? ■

Exercices de réflexion

Les exercices de réflexion demandent d'analyser le comportement de l'algorithme et de le combiner avec d'autres notions du programme.

■ **Exercice 8.11.**
On reprend la programmation orientée objet du chapitre sur la POO (chapitre 0).

1. Faire une classe `Cadeau` avec comme attribut une valeur.
2. Faire une liste de 1000 cadeaux avec des valeurs aléatoires (on utilisera `random.randint`).

3. Trier cette liste en utilisant (et en adaptant) l'algorithme de tri fusion, par ordre croissant de valeur.

```
import random

class Cadeau:
    def __init__(self, valeur):
        self.valeur = valeur
```

■

■ **Exercice 8.12.**

Comparaison expérimentale des tris. On dispose de `tri_fusion` (ce chapitre) et d'un tri par insertion (vu en Première).

1. Écrire un programme qui mesure le temps d'exécution des deux tris sur des listes aléatoires de tailles 100, 1000 et 10 000, à l'aide de `time.perf_counter`.
2. Présenter les résultats dans un tableau et conclure : pour quelle taille l'écart devient-il visible ?

```
import random, time

for taille in [100, 1000, 10000]:
    liste = [random.randint(0, 1000) for _ in range(taille)]
    ...
```

■

■ **Exercice 8.13.**

Une liste déjà triée. Le tri fusion effectue-t-il nettement moins d'opérations sur une liste déjà triée que sur une liste aléatoire ?

1. Justifier la réponse en analysant le nombre de niveaux de division et le coût de chaque fusion, quelle que soit la liste de départ.
2. Vérifier expérimentalement en chronométrant `tri_fusion` sur une liste déjà triée puis sur une liste aléatoire de même taille.

■

■ **Exercice 8.14.**

Tri de mots. Adapter le tri fusion pour trier une liste de mots par ordre alphabétique (en Python, la comparaison `<` fonctionne sur les chaînes de caractères). Tester sur la liste `["kiwi", "pomme", "banane", "cerise", "abricot"]` et

donner les étapes de la fusion finale. ■