TP Chapitre 8 : Tri Fusion

Objectifs du TP

Objectifs du TP. À l'issue de cette séance, tu dois être capable de :

- expliquer la méthode « diviser pour régner » (diviser, résoudre, combiner) et la reconnaître dans le tri fusion ;
- fusionner deux listes triées en temps linéaire, en version récursive et en version itérative ;
- écrire l'algorithme récursif du tri fusion ;
- trier une liste par ordre croissant ou décroissant en adaptant la comparaison ;
- justifier la complexité du tri fusion en $O(n \log_2 n)$ et la comparer aux tris quadratiques vus en Première ;
- expliquer la stabilité du tri fusion et la vérifier sur un exemple.

Consignes générales. Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur ; si tu obtiens une `RecursionError`, vérifie que le cas de base de ta récursivité est bien atteint. Les fonctions de la partie A (`fusion`, `tri_fusion`, `fusion_iterative`) sont réutilisées dans la partie B : garde-les dans ton fichier de travail. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation.

Exercice A1 — La fusion de deux relevés de transactions (fonction `fusion`)

Deux relevés bancaires contiennent des montants de transactions en euros (arrondis à l'euro). Chaque relevé est déjà trié du plus petit au plus grand montant. On veut une liste unique, triée, contenant toutes les transactions : c'est l'opération de *fusion*. On suit le pseudo-code du cours : si une liste est vide, la fusion est l'autre liste ; sinon on compare les deux têtes et on place la plus petite.

```
def fusion(A, B):
    """Renvoie la liste triee des montants de A et de B."""
    if A == []:
        return B
    # A COMPLETER : cas ou B est vide (renvoyer A)
    if A[0] <= B[0]:
        # A COMPLETER : placer A[0] puis fusionner les restes
    # A COMPLETER : cas ou B[0] est plus petit (placer B[0] puis fusionner
       les restes)
```

Point de validation

Vérifie : `fusion([12, 30, 75], [20, 45, 60])` doit renvoyer `[12, 20, 30, 45, 60, 75]`.
Vérifie aussi le cas d'une liste vide : `fusion([], [7, 8])` doit renvoyer `[7, 8]`.

Exercice A2 — Le tri fusion du catalogue de prix (fonction `tri_fusion`)

Un catalogue de prix en euros (entiers) doit être trié du moins cher au plus cher. Le tri fusion applique la méthode « diviser pour régner » : on coupe la liste en deux moitiés, on trie chaque moitié en appelant récursivement `tri_fusion`, puis on fusionne les deux moitiés triées avec la fonction `fusion` de l'exercice A1. Si la liste a au plus un élément, elle est déjà triée.

```
def tri_fusion(T):
    """Renvoie une copie triee de la liste T (prix en euros)."""
    n = len(T)
    if n <= 1:
        return T
    milieu = n // 2
    # A COMPLETER : trier recursivement la moitie gauche (T[:milieu])
    # A COMPLETER : trier recursivement la moitie droite (T[milieu:])
    # A COMPLETER : renvoyer la fusion des deux moities triees
```

Point de validation

Vérifie : `tri_fusion([45, 12, 78, 3, 56])` doit renvoyer `[3, 12, 45, 56, 78]`. Après l'appel, la liste initiale doit être inchangée : le tri fusion renvoie une *copie* triée, il ne modifie pas la liste de départ.

Exercice A3 — La fusion itérative de deux relevés (indices i et j)

La version récursive de la fusion découpe les listes avec l'écriture $A[1:]$, ce qui *recopie* la liste restante à chaque appel. C'est simple à écrire, mais pour de très grandes listes, on préfère une version *itérative* qui parcourt les deux listes avec deux indices i et j, sans recopie. 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):
    """Fusion de deux listes triees sans recopie (version iterative)."""
    i, j = 0, 0
    res = []
    # A COMPLETER : tant que i < len(A) et j < len(B),
    #               comparer A[i] et B[j] et placer le plus petit dans res
    # A COMPLETER : ajouter les elements restants de A, puis ceux de B
    return res
```

Point de validation

Vérifie : `fusion_iterative([5, 15, 25], [10, 20, 30, 40])` doit renvoyer `[5, 10, 15, 20, 25, 30, 40]`.

Exercice A4 — Les devis du plus cher au moins cher (tri décroissant)

Un client compare des devis et veut les voir du plus cher au moins cher. Pour trier par ordre *décroissant*, il suffit d'adapter la fusion : on modifie la comparaison pour placer d'abord la plus grande des deux têtes. On obtient alors `fusion_decroissante`, puis `tri_fusion_decroissant` qui a exactement la même structure que `tri_fusion`.

```
def fusion_decroissante(A, B):
    """Renvoie la liste triee par ordre decroissant des elements de A et de
       B."""
    if A == []:
        return B
    if B == []:
```

```
        return A
    # A COMPLETER : comparaison >= pour trier par ordre decroissant
    # A COMPLETER : deux cas recursifs (comme dans la fusion du cours)

def tri_fusion_decroissant(T):
    """Renvoie une copie de T triee par ordre decroissant."""
    n = len(T)
    if n <= 1:
        return T
    milieu = n // 2
    # A COMPLETER : appels recursifs et fusion_decroissante
```

Point de validation

Vérifie : `tri_fusion_decroissant([45, 12, 78, 3, 56])` doit renvoyer `[78, 56, 45, 12, 3]`. Réponds en une phrase : la fonction `tri_fusion_decroissant` diffère-t-elle de `tri_fusion` autrement que par la fonction de fusion appelée ?

Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

Exercice B1 — Le relevé de transactions trié

Un commerçant note les montants de ses ventes de la journée (en euros, arrondis à l'euro). Il veut les afficher triés du plus petit au plus grand. Écrire un programme qui demande le nombre de ventes, puis chaque montant, et qui affiche la liste triée à l'aide du tri fusion.

Spécification.

— *Entrées* : un entier N (nombre de ventes, $N \geq 1$), puis N montants (entiers, en euros).

— *Traitement* : construire une liste des montants, puis la trier avec la fonction `tri_fusion` (celle de l'exercice A2, à recopier dans le programme).

— *Sorties* : la liste triée des montants.

— *Contraintes* : `tri_fusion` (et la fusion qu'elle utilise) définies dans le programme ; ne pas utiliser `sorted()` ni `.sort()` ; la saisie se fait dans une boucle.

Pour t'aider — questions de conception (sans donner le code)

— Combien de valeurs va-t-on saisir ? Quelle boucle utiliser pour la saisie ?

— `tri_fusion` renvoie une nouvelle liste au lieu de modifier la liste de départ : comment récupérer son résultat ?

— Comment vérifier à la main que la liste affichée est juste ? (on peut comparer avec `sorted()` dans un test, sans l'utiliser dans le programme)

Point de validation

Vérifie : avec 5 ventes de montants 150, 32, 480, 75 et 210, le programme doit afficher `[32, 75, 150, 210, 480]`.

Exercice B2 — La fusion de deux relevés bancaires

On dispose de deux relevés bancaires triés par montant croissant (deux listes de montants en euros). On veut un récapitulatif unique, trié. Écrire un programme qui déclare les deux listes, les affiche, puis affiche leur fusion triée, sans utiliser `sorted()` ni `.sort()`. La fonction de fusion peut être la version récursive ou la version itérative, au choix.

Spécification.

— *Entrées* : deux listes triées A et B (déclarées en dur dans le programme).

— *Traitement* : fusionner A et B avec une fonction de fusion écrite dans le programme.

— *Sorties* : les deux listes de départ, puis la liste fusionnée triée.

— *Contraintes* : ne pas utiliser `sorted()` ni `.sort()` ; la fonction de fusion doit être écrite dans le programme ; choix libre entre version récursive et itérative.

Pour t'aider — questions de conception (sans donner le code)

— Quels sont les cas de base d'une fusion récursive ? Que renvoie-t-on si A est vide ?

— Pourquoi suffit-il de comparer les têtes $A[0]$ et $B[0]$ à chaque étape ?

— Combien d'étapes au maximum pour fusionner deux listes de tailles a et b ? (réponds en une phrase)

— Que se passe-t-il si l'une des deux listes est vide ?

Point de validation

Vérifie : avec $A = [10, 20, 30]$ et $B = [15, 25, 35]$, le programme doit afficher la fusion `[10, 15, 20, 25, 30, 35]`.

Exercice B3 — Les devis du plus cher au moins cher

Une entreprise a reçu cinq devis : 320, 145, 480, 210 et 95 euros. Le directeur veut les afficher du plus cher au moins cher. Écrire un programme qui déclare cette liste, la trie par ordre décroissant avec un tri fusion adapté, et affiche le résultat, sans utiliser `sorted()` ni `.sort()`.

Spécification.

— *Entrées* : la liste des prix $[320, 145, 480, 210, 95]$ (déclarée en dur).

— *Traitement* : tri décroissant par tri fusion adapté (`fusion_decroissante` et `tri_fusion_decroissant` écrites dans le programme).

— *Sorties* : la liste triée par ordre décroissant.

— *Contraintes* : ne pas utiliser `sorted()` ni `.sort()` ; écrire les deux fonctions (fusion décroissante et tri fusion décroissant) ; seule la comparaison change par rapport au tri croissant.

Pour t'aider — questions de conception (sans donner le code)

— Quelle comparaison faut-il modifier pour trier par ordre décroissant ? Dans quelle fonction ?

— Faut-il modifier autre chose dans `tri_fusion_decroissant` par rapport à `tri_fusion` ? Pourquoi ?

— Comment vérifier le résultat sur un petit exemple à la main ?

Point de validation

Vérifie : le programme doit afficher `[480, 320, 210, 145, 95]`.

Exercice B4 — Tri fusion contre tri par insertion : le comparateur de ventes

Un site de vente en ligne veut savoir si le tri fusion est vraiment plus rapide que le tri par insertion pour trier ses volumes de ventes. Écrire un programme qui :

1. définit `tri_insertion` (rappel de Première : insérer chaque élément à sa place dans la partie déjà triée) ;
2. définit `tri_fusion` (le tri récursif du chapitre, en utilisant la fusion itérative de l'exercice A3) ;
3. pour chaque taille 100, 1 000 et 10 000, génère une liste aléatoire de montants de ventes avec `random.randint(0, 1000)` ;
4. chronomètre chaque tri avec `time.perf_counter()` ;
5. affiche un tableau des temps (taille, temps du tri fusion, temps du tri par insertion).

Spécification.

— *Entrées* : aucune (listes générées aléatoirement ; tailles 100, 1 000 et 10 000).

— *Traitement* : pour chaque taille, générer une liste, la copier pour chaque tri, chronométrer `tri_fusion` puis `tri_insertion`.

— *Sorties* : un tableau à trois colonnes : taille, temps du tri fusion, temps du tri par insertion (en secondes).

— *Contraintes* : `import random` et `import time` ; `tri_insertion` et `tri_fusion` écrites dans le programme ; chaque tri est chronométré sur sa propre copie de la liste ; `tri_fusion` renvoyant une copie triée, penser à récupérer son résultat.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi faut-il chronométrer chaque tri sur une copie de la même liste, et pas sur la liste triée par l'autre tri ?

— Pourquoi doit-on écrire `l1 = tri_fusion(l1)` et pas simplement `tri_fusion(l1)` ?

— Quelle taille risque de rendre `tri_insertion` très lent ? Pourquoi (ordre de grandeur des coûts) ?

— Que conclure en comparant les temps des deux tris ?

Point de validation

Les temps dépendent de la machine ; les ordres de grandeur suivants ont été mesurés en Python 3 : à 100, les deux tris sont quasi instantanés (moins d'un millième de seconde) ; à 1 000, `tri_insertion` est environ 10 fois plus lent que `tri_fusion` ; à 10 000, `tri_insertion` est environ 80 fois plus lent (de l'ordre de 1,5 s contre 0,02 s). Si l'écart n'apparaît pas, vérifie que le chronométrage entoure bien l'appel complet de chaque tri et que chaque tri travaille sur sa propre copie.

Partie C — Exercice de réflexion (à finir à la maison)

À finir à la maison

Le grand livre des transactions : fusion d'historiques.

Contexte. Une association tient ses comptes. Chaque responsable a trié sa liste de transactions (montants en euros, arrondis à l'euro) du plus petit au plus grand : trois relevés triés, trois listes de montants. Le trésorier veut un « grand livre » unique : toutes les transactions, triées du plus petit au plus grand montant.

Objectif. Écrire un programme qui fusionne plusieurs listes triées en une seule liste triée, et qui réponde à des questions comme : comment fusionner plus de deux listes ? Combien de comparaisons la fusion effectue-t-elle ? La fusion préserve-t-elle l'ordre des transactions de même montant (stabilité) ?

Questions à se poser avant de coder.

— Comment fusionner deux listes triées ? (fonction déjà écrite dans la partie B)

— Comment fusionner trois listes (ou plus) ? Peut-on réutiliser plusieurs fois la fusion de deux listes ?

— Combien de comparaisons au maximum pour fusionner deux listes de tailles a et b ?

— Comment compter les comparaisons dans un programme ?

— Comment vérifier qu'une fusion est *stable*, c'est-à-dire qu'elle préserve l'ordre des éléments égaux ? (piste : numéroter les transactions)

— Comment vérifier que le programme est juste ? (petites listes, comparaison avec `sorted()` dans un test)

Étapes suggérées.

1. Version 0 : réutiliser la fonction de fusion de la partie B sur deux listes triées, par exemple $[10, 20, 30]$ et $[15, 25, 35]$; vérifier le résultat à la main.
2. Version 1 : fusionner successivement trois listes triées : on fusionne les deux premières, puis le résultat avec la troisième.
3. Version 2 : généraliser à une liste de listes (par exemple cinq relevés de tailles différentes) avec une boucle.
4. Version 3 : compter les comparaisons effectuées par la fusion et comparer le nombre obtenu à l'ordre de grandeur $n \log_2 n$ (où n est le nombre total de transactions).
5. Bonus : stabilité — numéroter les transactions (tuples `(montant, numero)`) et vérifier que deux transactions de même montant conservent leur ordre initial ; mesurer le temps de fusion sur un grand nombre de transactions.

Contraintes. Les listes d'entrée sont déjà triées ; montants en euros, arrondis à l'euro (entiers) ; programme testé et commenté ; les messages peuvent être écrits sans accents (ex. `Grand livre`, `comparaisons`) si ton éditeur pose problème.

Fin du TP — la partie C est à finir à la maison.