Algorithmie et structures de données

ENSISA 1A

Jonathan Weber

Hiver 2024

Plan de la séance

Introduction aux algorithmes de tri

Pourquoi trier les données ?

- Les données triées facilitent l'accès et la manipulation.

Pourquoi trier les données ?

- Les données triées facilitent l'accès et la manipulation.
- Utilisation dans des domaines variés :
  - Bases de données (tri des enregistrements).
  - Recherche rapide (recherche binaire).
  - Préparation pour d'autres algorithmes avancés (ex. : algorithmes de graphe, clustering).

Pourquoi trier les données ?

- Les données triées facilitent l'accès et la manipulation.
- Utilisation dans des domaines variés :
  - Bases de données (tri des enregistrements).
  - Recherche rapide (recherche binaire).
  - Préparation pour d'autres algorithmes avancés (ex. : algorithmes de graphe, clustering).
- Cas d'usage concret :
  - Recherche binaire : nécessite une liste triée pour fonctionner efficacement.

Exemple : Recherche Binaire

Principe

- Fonctionne uniquement sur des données triées.
- Divise la liste en deux à chaque itération pour trouver l'élément recherché.

Principe

- Fonctionne uniquement sur des données triées.
- Divise la liste en deux à chaque itération pour trouver l'élément recherché.

Complexité

- Temps d'exécution : $O(\log n)$.
- Plus rapide que la recherche linéaire ($O(n)$) sur de grandes listes.

Classification des Algorithmes de Tri

- 2 approches
  - **Tri par comparaison :**
    - Compare les éléments deux à deux.
    - Exemples : tri rapide, tri fusion.
  - **Tri sans comparaison :**
    - Utilise les propriétés des données (ex. : tri par comptage).
    - Adapté aux entiers ou données avec plages limitées.

Classification des Algorithmes de Tri

- 2 approches
  - **Tri par comparaison :**
    - Compare les éléments deux à deux.
    - Exemples : tri rapide, tri fusion.
  - **Tri sans comparaison :**
    - Utilise les propriétés des données (ex. : tri par comptage).
    - Adapté aux entiers ou données avec plages limitées.
- Stabilité du tri
  - **Tri stable :** conserve l'ordre relatif des éléments égaux.
  - **Tri non stable :** ne garantit pas cet ordre.

- 2 approches
  - **Tri par comparaison :**
    - Compare les éléments deux à deux.
    - Exemples : tri rapide, tri fusion.
  - **Tri sans comparaison :**
    - Utilise les propriétés des données (ex. : tri par comptage).
    - Adapté aux entiers ou données avec plages limitées.
- Stabilité du tri
  - **Tri stable :** conserve l'ordre relatif des éléments égaux.
  - **Tri non stable :** ne garantit pas cet ordre.
  - Exemple avec la liste : {(3, A), (2, B), (3, C)}
    - Tri stable : {(2, B), (3, A), (3, C)}
    - Tri non stable : {(2, B), (3, C), (3, A)}

- **Complexité temporelle :**
  - Temps requis pour trier une liste de taille n.
  - Exemples :
    - $O(n^2)$: tri par insertion, tri à bulles.
    - $O(n \log n)$: tri rapide, tri fusion.
- **Complexité spatiale :**
  - Mémoire additionnelle utilisée par l'algorithme.
  - Tri fusion : nécessite une mémoire auxiliaire ($O(n)$).
  - Tri rapide : mémoire en place ($O(\log n)$).

Tri par comparaison

Tri par Comparaison : Les Bases

- Méthodes simples et intuitives pour trier des données.
- Comparaison des éléments deux à deux pour déterminer leur ordre.
- Nous allons étudier trois algorithmes de base :
    - Tri par sélection.
    - Tri par insertion.
    - Tri à bulles.

Principe :

- Trouver le plus petit élément de la liste.
- Le placer à la première position.
- Répéter pour le reste de la liste.

Liste initiale :
{29, 10, 14, 37, 13}

- Étape 1 : Trouver le minimum (10) et l'échanger avec le premier élément.
- Étape 2 : Répéter pour les éléments restants.

Tri par Sélection - Exemple

Liste initiale :
{29, 10, 14, 37, 13}

- Étape 1 : Trouver le minimum (10) et l'échanger avec le premier élément.
- Étape 2 : Répéter pour les éléments restants.

Déroulement :

1. {29, 10, 14, 37, 13}

Liste initiale :
{29, 10, 14, 37, 13}

- Étape 1 : Trouver le minimum (10) et l'échanger avec le premier élément.
- Étape 2 : Répéter pour les éléments restants.

Déroulement :

1. {29, 10, 14, 37, 13}
2. {**10**, 29, 14, 37, 13}

Tri par Sélection - Exemple

Liste initiale :
{29, 10, 14, 37, 13}

- Étape 1 : Trouver le minimum (10) et l'échanger avec le premier élément.
- Étape 2 : Répéter pour les éléments restants.

Déroulement :

1. {29, 10, 14, 37, 13}
2. {**10**, 29, 14, 37, 13}
3. {**10**, **13**, 14, 37, 29}

Tri par Sélection - Exemple

Liste initiale :
{29, 10, 14, 37, 13}

- Étape 1 : Trouver le minimum (10) et l'échanger avec le premier élément.
- Étape 2 : Répéter pour les éléments restants.

Déroulement :

1. {29, 10, 14, 37, 13}
2. {**10**, 29, 14, 37, 13}
3. {**10**, **13**, 14, 37, 29}
4. {**10**, **13**, **14**, 37, 29}

Tri par Sélection - Exemple

Liste initiale :
{29, 10, 14, 37, 13}

- Étape 1 : Trouver le minimum (10) et l'échanger avec le premier élément.
- Étape 2 : Répéter pour les éléments restants.

Déroulement :

1. {29, 10, 14, 37, 13}
2. {**10**, 29, 14, 37, 13}
3. {**10**, **13**, 14, 37, 29}
4. {**10**, **13**, **14**, 37, 29}
5. {**10**, **13**, **14**, **29**, **37**}

Complexité :

- $O(n^2)$ comparaisons dans le pire et le cas moyen.
- Pas adapté aux grandes listes.

Complexité :

- $O(n^2)$ comparaisons dans le pire et le cas moyen.
- Pas adapté aux grandes listes.

Pseudocode :

```
fonction triParSelection(liste):
    n = longueur(liste)
    POUR i de 0 à n-2:
        indexMin = i
        POUR j de i+1 à n-1:
            SI liste[j] < liste[indexMin]:
                indexMin = j
            FINSI
        FINPOUR
        échanger(liste[i], liste[indexMin])
    FINPOUR
```

Principe :

- Considérer le premier élément comme trié.
- Insérer les éléments suivants dans leur position correcte dans la partie triée.

Déroulement :

1. $\{7\} \leftarrow \{7, 3, 5, 2, 4\}$

Principe :

- Considérer le premier élément comme trié.
- Insérer les éléments suivants dans leur position correcte dans la partie triée.

Déroulement :

1. $\{7\} \leftarrow \{7, 3, 5, 2, 4\}$
2. $\{3, 7\} \leftarrow \{3, 5, 2, 4\}$

Principe :

- ▷ Considérer le premier élément comme trié.
- ▷ Insérer les éléments suivants dans leur position correcte dans la partie triée.

Déroulement :

1. $\{7\} \leftarrow \{7, 3, 5, 2, 4\}$
2. $\{3, 7\} \leftarrow \{3, 5, 2, 4\}$
3. $\{3, 5, 7\} \leftarrow \{5, 2, 4\}$

Principe :

- Considérer le premier élément comme trié.
- Insérer les éléments suivants dans leur position correcte dans la partie triée.

Déroulement :

1. $\{7\} \leftarrow \{7, 3, 5, 2, 4\}$
2. $\{3, 7\} \leftarrow \{3, 5, 2, 4\}$
3. $\{3, 5, 7\} \leftarrow \{5, 2, 4\}$
4. $\{2, 3 ,5, 7\} \leftarrow \{2, 4\}$

Principe :

- Considérer le premier élément comme trié.
- Insérer les éléments suivants dans leur position correcte dans la partie triée.

Déroulement :

1. $\{7\} \leftarrow \{7, 3, 5, 2, 4\}$
2. $\{3, 7\} \leftarrow \{3, 5, 2, 4\}$
3. $\{3, 5, 7\} \leftarrow \{5, 2, 4\}$
4. $\{2, 3 ,5, 7\} \leftarrow \{2, 4\}$
5. $\{2, 3, 4, 5, 7\} \leftarrow \{4\}$

Complexité :

- Cas moyen : $O(n^2)$.
- Meilleur cas (liste déjà triée) : $O(n)$.
- Adapté aux petites listes.

Complexité :

- Cas moyen : $O(n^2)$.
- Meilleur cas (liste déjà triée) : $O(n)$.
- Adapté aux petites listes.

Pseudocode :

```
fonction triParInsertion(liste):
    n = longueur(liste)
    POUR i de 1 à n-1:
        valeur = liste[i]
        j = i
        TANTQUE j > 0 et liste[j-1] > valeur:
            liste[j] = liste[j-1]
            j = j - 1
        FINTANTQUE
        liste[j] = valeur
    FINPOUR
```

Tri à Bulles

Principe :

- Comparer les éléments adjacents.
- Les échanger s'ils sont dans le mauvais ordre.
- Répéter jusqu'à ce que la liste soit triée.

Principe :

- Comparer les éléments adjacents.
- Les échanger s'ils sont dans le mauvais ordre.
- Répéter jusqu'à ce que la liste soit triée.

Exemple :

Déroulement :

1. $\{5, 1, 4, 2, 8\}$

Tri à Bulles

Principe :

- Comparer les éléments adjacents.
- Les échanger s'ils sont dans le mauvais ordre.
- Répéter jusqu'à ce que la liste soit triée.

Exemple :

Déroulement :

1. $\{5, 1, 4, 2, 8\}$
2. $\{1, 5, 4, 2, 8\}$

Principe :

- Comparer les éléments adjacents.
- Les échanger s'ils sont dans le mauvais ordre.
- Répéter jusqu'à ce que la liste soit triée.

Exemple :

Déroulement :

1. {5, 1, 4, 2, 8}
2. {1, 5, 4, 2, 8}
3. {1, 4, 5, 2, 8}

Principe :

- Comparer les éléments adjacents.
- Les échanger s'ils sont dans le mauvais ordre.
- Répéter jusqu'à ce que la liste soit triée.

Exemple :

Déroulement :

1. $\{5, 1, 4, 2, 8\}$
2. $\{1, 5, 4, 2, 8\}$
3. $\{1, 4, 5, 2, 8\}$
4. $\{1, 4, 2, 5, 8\}$

Principe :

- ▹ Comparer les éléments adjacents.
- ▹ Les échanger s'ils sont dans le mauvais ordre.
- ▹ Répéter jusqu'à ce que la liste soit triée.

Exemple :

Déroulement :

1. $\{5, 1, 4, 2, 8\}$
2. $\{1, 5, 4, 2, 8\}$
3. $\{1, 4, 5, 2, 8\}$
4. $\{1, 4, 2, 5, 8\}$
5. $\{1, 2, 4, 5, 8\}$

Tri à Bulles - Complexité et Code

Complexité :

- $O(n^2)$ pour le pire et le cas moyen.
- Peu efficace pour de grandes données.

Complexité :

- $O(n^2)$ pour le pire et le cas moyen.
- Peu efficace pour de grandes données.

Code simplifié :

```
fonction triABulles(liste):
    n = longueur(liste)
    POUR i allant de n-1 à 1
       POUR j allant de 0 à i-1
           SI liste[j+1] < liste[j]
               échanger(liste[j], liste[j+1])
           FINSI
       FINPOUR
    FINPOUR
```

- Les algorithmes de tri par comparaison sont faciles à comprendre et à implémenter.
- Limites :
  - Inefficaces pour des grandes données ($O(n^2)$).
- À venir : tri par comparaison avancé ($O(n \log n)$).

Tri par comparaison avancé

- Algorithmes plus efficaces que les méthodes basiques ($O(n^2)$).
- Complexité moyenne : $O(n \log n)$.
- Étude de deux algorithmes :
  - Tri fusion (*Merge Sort*).
  - Tri rapide (*Quick Sort*).

Principe : Diviser pour mieux régner.

- Diviser : Découper la liste en deux moitiés égales.
- Conquérir : Trier récursivement chaque moitié.
- Fusionner : Combiner les deux moitiés triées en une seule liste triée.

Principe : Diviser pour mieux régner.

- Diviser : Découper la liste en deux moitiés égales.
- Conquérir : Trier récursivement chaque moitié.
- Fusionner : Combiner les deux moitiés triées en une seule liste triée.

Complexité :

- Temps d'exécution : $O(n \log n)$ (toutes les étapes).
- Espace supplémentaire requis : $O(n)$.

Tri Fusion - Exemple Visuel

- Étape 1 : Diviser jusqu'à obtenir des listes d'un seul élément.
- Étape 2 : Fusionner les listes triées.

Déroulement

1. {38, 27, 43, 3, 9, 82, 10}

Tri Fusion - Exemple Visuel

- Étape 1 : Diviser jusqu'à obtenir des listes d'un seul élément.
- Étape 2 : Fusionner les listes triées.

Déroulement

1. {38, 27, 43, 3, 9, 82, 10}
2. {38, 27, 43, 3} {9, 82, 10}

- Étape 1 : Diviser jusqu'à obtenir des listes d'un seul élément.
- Étape 2 : Fusionner les listes triées.

Déroulement

1. {38, 27, 43, 3, 9, 82, 10}
2. {38, 27, 43, 3} {9, 82, 10}
3. {38, 27} {43, 3} {9, 82} {10}

- Étape 1 : Diviser jusqu'à obtenir des listes d'un seul élément.
- Étape 2 : Fusionner les listes triées.

Déroulement

1. {38, 27, 43, 3, 9, 82, 10}
2. {38, 27, 43, 3} {9, 82, 10}
3. {38, 27} {43, 3} {9, 82} {10}
4. {38} {27} {43} {3} {9} {82} {10}

Tri Fusion - Exemple Visuel

- Étape 1 : Diviser jusqu'à obtenir des listes d'un seul élément.
- Étape 2 : Fusionner les listes triées.

Déroulement

1. {38, 27, 43, 3, 9, 82, 10}
2. {38, 27, 43, 3} {9, 82, 10}
3. {38, 27} {43, 3} {9, 82} {10}
4. {38} {27} {43} {3} {9} {82} {10}
5. {27, 38} {3, 43} {9, 82} {10}

- Étape 1 : Diviser jusqu'à obtenir des listes d'un seul élément.
- Étape 2 : Fusionner les listes triées.

Déroulement

1. {38, 27, 43, 3, 9, 82, 10}
2. {38, 27, 43, 3} {9, 82, 10}
3. {38, 27} {43, 3} {9, 82} {10}
4. {38} {27} {43} {3} {9} {82} {10}
5. {27, 38} {3, 43} {9, 82} {10}
6. {3, 27, 38, 43} {9, 10, 82}

- Étape 1 : Diviser jusqu'à obtenir des listes d'un seul élément.
- Étape 2 : Fusionner les listes triées.

Déroulement

1. {38, 27, 43, 3, 9, 82, 10}
2. {38, 27, 43, 3} {9, 82, 10}
3. {38, 27} {43, 3} {9, 82} {10}
4. {38} {27} {43} {3} {9} {82} {10}
5. {27, 38} {3, 43} {9, 82} {10}
6. {3, 27, 38, 43} {9, 10, 82}
7. {**3, 9, 10, 27, 38, 43, 82**}

```
fonction triFusion(liste):
    n = longueur(liste)
    SI n <= 1: retourner liste
    SINON
        retourner fusion(triFusion(liste[0;n/2-1]),
                         triFusion(liste[n/2, n-1]))
    FINSI
fonction fusion(gauche, droite):
    resultat = []
    TANTQUE gauche OU droite ne sont pas vides:
        SI gauche.premierElement <= droite.premierElement:
            resultat.ajouter(gauche.sortirPremierElement)
        SINON
            resultat.ajouter(droite.sortirPremierElement)
        FINSI
    FINTANTQUE
    resultat.ajouter(gauche).ajouter(droite)
    retourner resultat
```

Principe : Partitionnement avec un pivot.

- Choisir un pivot dans la liste.
- Réorganiser les éléments :
    - Les éléments plus petits que le pivot à gauche.
    - Les éléments plus grands à droite.
- Appliquer récursivement à chaque sous-liste.

Principe : Partitionnement avec un pivot.

- Choisir un pivot dans la liste.
- Réorganiser les éléments :
  - Les éléments plus petits que le pivot à gauche.
  - Les éléments plus grands à droite.
- Appliquer récursivement à chaque sous-liste.

Complexité :

- Moyen : $O(n \log n)$.
- Pire cas : $O(n^2)$ (si le pivot est mal choisi).

Méthodes pour choisir un pivot :

- Premier élément de la liste.
- Dernier élément.
- Élément aléatoire.
- Médiane de trois (médiane entre le premier, le dernier, et le milieu).

Méthodes pour choisir un pivot :

- Premier élément de la liste.
- Dernier élément.
- Élément aléatoire.
- Médiane de trois (médiane entre le premier, le dernier, et le milieu).

Impact :

- Un bon choix de pivot réduit la probabilité du pire cas ($O(n^2)$).

Tri Rapide - Exemple Visuel

Liste initiale :
{30, 90, 10, 80, 50, 40, 70}

- Pivot : 70.
- Partitionnement :
  - {10, 30, 40, 50} (gauche).
  - {80, 90} (droite).
- Appliquer récursivement sur chaque sous-liste.

Déroulement

1. {30, 90, 10, 80, 50, 40, 70}

Tri Rapide - Exemple Visuel

Liste initiale :
{30, 90, 10, 80, 50, 40, 70}

- Pivot : 70.
- Partitionnement :
  - {10, 30, 40, 50} (gauche).
  - {80, 90} (droite).
- Appliquer récursivement sur chaque sous-liste.

Déroulement

1. {30, 90, 10, 80, 50, 40, 70}
2. {30, 10, 50, 40, **70**, 90, 80}

Tri Rapide - Exemple Visuel

Liste initiale :
{30, 90, 10, 80, 50, 40, 70}

- Pivot : 70.
- Partitionnement :
  - {10, 30, 40, 50} (gauche).
  - {80, 90} (droite).
- Appliquer récursivement sur chaque sous-liste.

Déroulement

1. {30, 90, 10, 80, 50, 40, 70}
2. {30, 10, 50, 40, **70**, 90, 80}
3. {30, 10, 50, **40**, **50**, **70**, **80**, **90**}

Liste initiale :
{30, 90, 10, 80, 50, 40, 70}

- Pivot : 70.
- Partitionnement :
  - {10, 30, 40, 50} (gauche).
  - {80, 90} (droite).
- Appliquer récursivement sur chaque sous-liste.

Déroulement

1. {30, 90, 10, 80, 50, 40, 70}
2. {30, 10, 50, 40, **70**, 90, 80}
3. {30, 10, 50, **40**, **50**, **70**, **80**, **90**}
4. {**10**, **30**, **50**, **40**, **50**, **70**, **80**, **90**}

Pseudocode :

```
fonction triRapide(liste):
    SI longueur(liste) <= 1:
        retourner liste
    FINSI
    pivot = choisirPivot(liste)
    gauche = [], egal=[], droite=[]
    POUR i de 0 à longueur(liste)-1:
        SI liste[i] < pivot: gauche.ajouter(liste[i])
        SINON SI liste[i] == pivot: egal.ajouter(liste[i])
        SINON droite.ajouter(liste[i])
        FINSI
    FINPOUR
    retourner triRapide(gauche).ajouter(egal)
                        .ajouter(triRapide(droite))
```

Conclusion

- Tri fusion et tri rapide sont très efficaces pour de grandes données.
- Choix entre les deux :
  - Tri fusion : stable, mais utilise plus de mémoire ($O(n)$).
  - Tri rapide : en place, mais peut être inefficace si le pivot est mal choisi.
- Prochaines étapes : Algorithmes de tri sans comparaison.

Algorithmes de tri sans comparaison

Algorithmes de Tri sans Comparaison

- Contrairement aux tris par comparaison, ces algorithmes ne comparent pas les éléments directement.
- Utilisent des propriétés des données (comme leur plage ou leur représentation en chiffres).
- Adaptés à des cas spécifiques, notamment pour des entiers ou des données structurées.
- Deux algorithmes étudiés ici :
  - Tri par comptage (*Counting Sort*).
  - Tri par base (*Radix Sort*).
  - Tri par paquets (*Bucket Sort*).

Principe :

- Créer un tableau pour compter le nombre d'occurrences de chaque élément.
- Utiliser ce tableau pour reconstituer la liste triée.

Principe :

- Créer un tableau pour compter le nombre d'occurrences de chaque élément.
- Utiliser ce tableau pour reconstituer la liste triée.

Caractéristiques :

- Adapté aux entiers dans une plage restreinte $[0, k]$.
- Non basé sur la comparaison d'éléments.

Liste initiale : $\{4, 2, 2, 8, 3, 3, 1\}$

Liste initiale : {4, 2, 2, 8, 3, 3, 1}
Étapes :

1. Créer un tableau de comptage.
   Compte : {0, 1, 2, 2, 1, 0, 0, 0, 1}.
2. Reconstituer la liste triée.
   Résultat : {1, 2, 2, 3, 3, 4, 8}.

Complexité :

- Temps : $O(n+k)$, où n est la taille de la liste et k la plage des valeurs.
- Espace : $O(k)$ pour le tableau de comptage.

Complexité :

- Temps : $O(n+k)$, où n est la taille de la liste et k la plage des valeurs.
- Espace : $O(k)$ pour le tableau de comptage.

Pseudocode :

```
fonction triParComptage(liste):
    k = max(liste)
    compte = liste de taille k+1 de 0
    POUR i de 0 à longueur(liste)-1:
        compte[liste[i]] += 1
    FINPOUR
    resultat = nouvelle liste
    POUR i de 0 à k:
        POUR j de 0 à compte[i]:
            resultat.ajouter(i)
        FINPOUR
    FINPOUR
    retourner resultat
```

Principe :

- Trier les nombres chiffre par chiffre, en partant du chiffre le moins significatif (unités).
- Utiliser un tri stable (comme le tri par comptage) pour chaque chiffre.
- Reconstituer la liste triée.

Principe :

- Trier les nombres chiffre par chiffre, en partant du chiffre le moins significatif (unités).
- Utiliser un tri stable (comme le tri par comptage) pour chaque chiffre.
- Reconstituer la liste triée.

Caractéristiques :

- Très efficace pour des nombres entiers avec une longueur de chiffres similaire.
- Basé sur les représentations numériques.

Liste initiale : {170, 45, 75, 90, 802, 24, 2, 66}

Liste initiale : {170, 45, 75, 90, 802, 24, 2, 66} **Étapes :**

- Étape 1 : Trier par le chiffre des unités.
  Résultat : {802, 2, 24, 45, 75, 66, 170, 90}.
- Étape 2 : Trier par le chiffre des dizaines.
  Résultat : {802, 2, 24, 45, 66, 75, 90, 170}.
- Étape 3 : Trier par le chiffre des centaines.
  Résultat final : {2, 24, 45, 66, 75, 90, 170, 802}.

Complexité :

- Temps : $O(d \cdot (n+k))$, où d est le nombre de chiffres.
- Espace : $O(n)$

Complexité :

- Temps : $O(d \cdot (n+k))$, où d est le nombre de chiffres.
- Espace : $O(n)$

Pseudocode :

```
fonction triParBase(liste):
    maxVal = max(liste)
    d = nombreDeChiffres(maxVal)
    position = 1
    POUR i de 1 à d:
        liste = triParComptage(liste, position)
        position = position + 1
    FINPOUR
    retourner liste
```

Principe :

- Diviser les données en plusieurs paquets ou seaux (*buckets*).
- Trier les éléments dans chaque seau (avec un tri simple comme le tri par insertion).
- Fusionner les seaux triés pour obtenir la liste finale.

Principe :

- Diviser les données en plusieurs paquets ou seaux (*buckets*).
- Trier les éléments dans chaque seau (avec un tri simple comme le tri par insertion).
- Fusionner les seaux triés pour obtenir la liste finale.

Caractéristiques :

- Adapté aux données uniformément réparties.
- Complexité en temps : Pire cas $O(n^2)$ - Moyen $O(n+k+\frac{n^2}{k})$, où k est le nombre de seaux.
- Complexité en espace : $O(n+k)$

Liste initiale : $\{0.42, 0.32, 0.23, 0.52, 0.25, 0.47, 0.51\}$

Bucket Sort - Exemple

Liste initiale : {0.42, 0.32, 0.23, 0.52, 0.25, 0.47, 0.51}

1. **Répartition dans les seaux.**
   Seaux : {[0.23, 0.25], [0.32], [0.42, 0.47], [0.51, 0.52]}.
2. **Trier chaque seau.**
   Résultat des seaux triés : {[0.23, 0.25], [0.32], [0.42, 0.47], [0.51, 0.52]}.
3. **Fusionner les seaux.**
   Liste triée : {0.23, 0.25, 0.32, 0.42, 0.47, 0.51, 0.52}.

Pseudocode :

```
fonction triParPaquets(liste, nombrePaquets):
    paquets = liste de nombrePaquets liste vide
    maxVal = max(liste) + 1
    for i = 0 to length(array) do
        insert liste[i] into paquets[floor(nombrePaquets
                                         × array[i] / maxVal)]
    for i = 0 to k do
        triParInsertion(paquets[i])
    retourner la concatenation des paquets triés de 0 à k
```

Remarque : Le choix du nombre de seaux (k) influence fortement les performances.

Conclusion

- Les algorithmes de tri sans comparaison sont très efficaces pour des cas spécifiques.
- **Tri par comptage :** Adapté aux entiers dans une plage restreinte.
- **Tri par base :** Adapté aux entiers avec une longueur fixe.
- **Bucket Sort :** performant pour des données uniformément réparties
- Limites :
  - Dépendent fortement des données.
  - Nécessitent des structures de données supplémentaires.

Conclusion

Récapitulatif des Points Clés

- Les algorithmes de tri sont essentiels pour organiser et manipuler efficacement les données.
- Classification des algorithmes :
  - Algorithmes simples ($O(n^2)$ comme tri par insertion, sélection).
  - Algorithmes avancés ($O(n \log n)$ comme tri fusion, tri rapide).
  - Algorithmes sans comparaison (tri par comptage, tri par base).
- Le choix d'un algorithme dépend des contraintes :
  - Taille et structure des données.
  - Ressources mémoire.

Timsort : Un exemple d'algorithme hybride

- Utilisé dans Python et Java.
- Combine les forces de :
  - Tri par insertion (pour petites sous-listes).
  - Tri fusion (pour grandes sous-listes).
- Performant sur des données réelles, souvent partiellement triées.

Timsort : Un exemple d'algorithme hybride

- Utilisé dans Python et Java.
- Combine les forces de :
  - Tri par insertion (pour petites sous-listes).
  - Tri fusion (pour grandes sous-listes).
- Performant sur des données réelles, souvent partiellement triées.

Autres algorithmes hybrides :

- Tri introspectif (*Introsort*) : combine tri rapide et tri par tas.
- Approches adaptatives basées sur les données.

Quid des tris parallèles ou distribués ?

- Tris parallèles :
  - Répartition des tâches sur plusieurs processeurs.
  - Exemple : Tri bitonique, tri parallèle à bulles.
- Tris distribués :
  - Partage des données sur plusieurs machines.
  - Utilisation dans les grands systèmes (ex. : Hadoop, Spark).

Quid des tris parallèles ou distribués ?

- Tris parallèles :
  - Répartition des tâches sur plusieurs processeurs.
  - Exemple : Tri bitonique, tri parallèle à bulles.
- Tris distribués :
  - Partage des données sur plusieurs machines.
  - Utilisation dans les grands systèmes (ex. : Hadoop, Spark).

Autres perspectives :

- Optimisations basées sur l'architecture matérielle (mémoire cache, GPU).
- Algorithmes de tri quantique : explorations théoriques.

Conclusion Générale

- Les algorithmes de tri illustrent les compromis entre performance, stabilité et mémoire.
- Aucun algorithme unique ne peut répondre à tous les besoins.
- Avec des jeux de données massifs, les approches parallèles et distribuées deviennent incontournables.