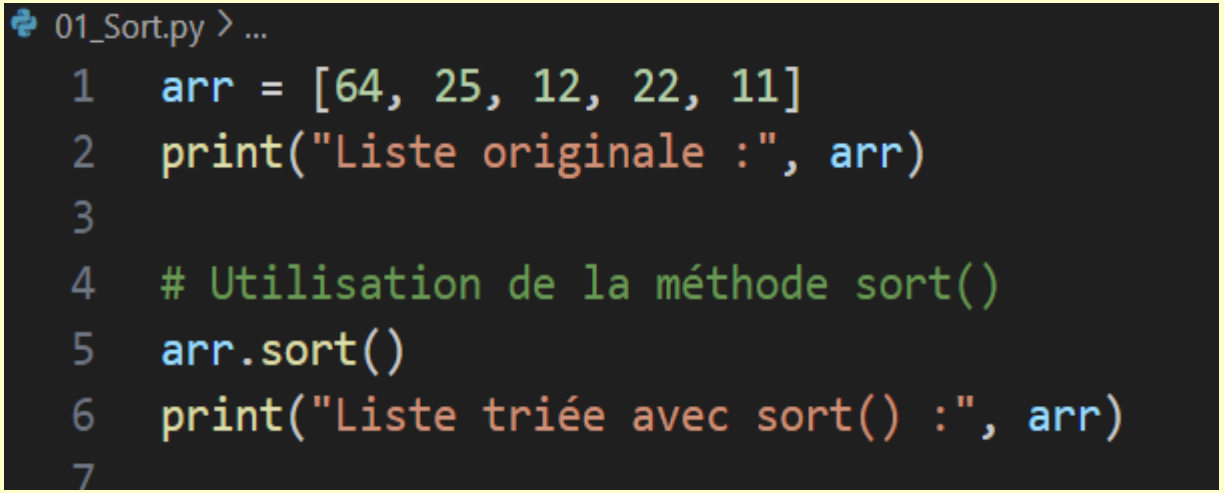
Algorithme de tri

Principe et complexité

01_sort_python.py
Utilisation de la fonction sort()

La méthode sort() de Python est beaucoup plus optimisée que les algorithmes de tri basiques comme le tri par sélection. Voici un exemple d'utilisation :



La fonction sort() utilise l'algorithme **Timsort**, qui est une combinaison de tri par insertion et de tri fusion, avec une complexité temporelle de O(nlogn) dans le pire des cas.

Conclusion (sort)

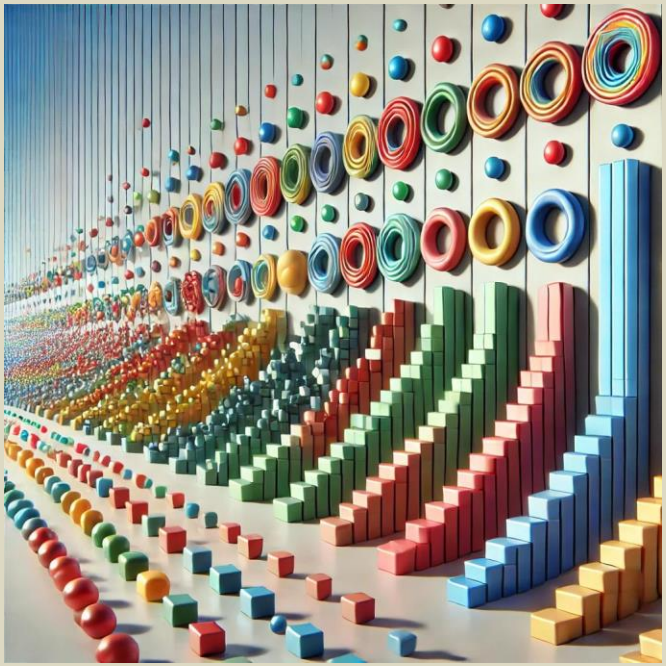
Utilisation pratique :

Pour trier des listes dans des applications réelles, il est préférable d'utiliser sort() ou sorted() en raison de leur efficacité et de leur optimisation.

Algorithmie et complexité :

Implémenter des algorithmes comme le tri par sélection permet de mieux comprendre les concepts fondamentaux de l'algorithmie, d'apprendre à analyser la complexité et d'apprécier les avantages des algorithmes optimisés intégrés dans les bibliothèques standard.

Algorithme : Tri Insertion



Principe de l'algorithme

Le tri par insertion est un algorithme de tri simple mais inefficace pour les grandes listes. Il est souvent utilisé pour trier de petites listes ou comme partie d'algorithmes plus complexes (comme Timsort). Voici les étapes principales :

1. **Diviser** : Considérer que le premier élément de la liste est trié.
2. **Insertion** : Prendre le prochain élément et l'insérer à sa place correcte dans la partie triée de la liste.
3. **Répéter** : Répéter le processus jusqu'à ce que toute la liste soit triée.

02_tri_insertion.py - Code en Python

00_Tri_insertion.py > ...

```python
def insertion_sort(liste):
    for i in range(1, len(liste)):
        key = liste[i]
        j = i - 1
        # Déplacer les éléments de liste[0..i-1], qui sont plus grands que la clé,
        # d'une position vers la droite pour faire de la place à la clé
        while j >= 0 and key < liste[j]:
            liste[j + 1] = liste[j]
            j -= 1
        liste[j + 1] = key
    return liste

# Exemple d'utilisation
liste = [64, 25, 12, 22, 11]
print("Liste originale :", liste)
sorted_liste = insertion_sort(liste)
print("Liste triée :", sorted_liste)
```

Analyse de la complexité

- **Complexité temporelle** :
  - **Meilleur cas** : O(n). Cela se produit lorsque la liste est déjà triée. L'algorithme ne fait que passer une fois à travers la liste, vérifiant que chaque élément est à sa place correcte.
  - **Cas moyen et pire cas** : O(n2). Dans ces cas, chaque élément est comparé avec chaque autre élément de la partie triée de la liste, entraînant une complexité quadratique.
- **Complexité spatiale** : O(1). Le tri par insertion est un algorithme en place, ce qui signifie qu'il trie la liste sans nécessiter de mémoire additionnelle proportionnelle à la taille de la liste.
- **Stabilité** : Le tri par insertion est un algorithme stable. Cela signifie que si deux éléments ont la même valeur, leur ordre relatif est conservé.

Exemple étape par étape

Prenons un exemple pour illustrer le fonctionnement de l'algorithme :
Liste originale : [64, 25, 12, 22, 11]

- **Première itération (i = 1)** :
  - Clé = 25.
  - Comparer avec 64 et échanger. Liste : [25, 64, 12, 22, 11].
- **Deuxième itération (i = 2)** :
  - Clé = 12.
  - Comparer avec 64 et échanger. Liste : [25, 12, 64, 22, 11].
  - Comparer avec 25 et échanger. Liste : [12, 25, 64, 22, 11].
- **Troisième itération (i = 3)** :
  - Clé = 22.
  - Comparer avec 64 et échanger. Liste : [12, 25, 22, 64, 11].
  - Comparer avec 25 et échanger. Liste : [12, 22, 25, 64, 11].
- **Quatrième itération (i = 4)** :
  - Clé = 11.
  - Comparer avec 64 et échanger. Liste : [12, 22, 25, 11, 64].
  - Comparer avec 25 et échanger. Liste : [12, 22, 11, 25, 64].
  - Comparer avec 22 et échanger. Liste : [12, 11, 22, 25, 64].
  - Comparer avec 12 et échanger. Liste : [11, 12, 22, 25, 64].

- Liste triée : [11, 12, 22, 25, 64]

Conclusion

Le tri par insertion est un algorithme simple à comprendre et à implémenter, ce qui en fait un bon choix pour des fins pédagogiques et pour des listes très petites.

Cependant, en raison de sa complexité temporelle quadratique, il n'est pas adapté pour trier de grandes listes en pratique.

Pour des applications réelles nécessitant des performances élevées, il est préférable d'utiliser des algorithmes de tri plus efficaces comme le tri rapide ou le tri fusion.

Tri par Sélection (Sélection Sort)

Principe de l'algorithme

Le tri par sélection est un algorithme de tri simple mais inefficace pour les grandes listes. Son principe repose sur les étapes suivantes :

1. **Recherche du minimum** : Rechercher le plus petit élément dans la liste non triée.
2. **Échange** : Échanger cet élément avec le premier élément de la liste non triée.
3. **Itération** : Recommencer les étapes 1 et 2 pour la sous-liste restante (excluant le premier élément trié) jusqu'à ce que toute la liste soit triée.

Pour chaque position dans la liste, l'algorithme sélectionne l'élément le plus petit parmi les éléments restants et le place à cette position.

03_selection.py - Code en Python

01_Algorithme de tri (Sort).py > ...

```python
def selection_sort(liste):
    n = len(liste)
    for i in range(n):
        # Trouver le minimum dans la sous-liste liste[i:n]
        min_index = i
        for j in range(i+1, n):
            if liste[j] < liste[min_index]:
                min_index = j
        # Échanger le minimum trouvé avec le premier élément de la sous-liste
        liste[i], liste[min_index] = liste[min_index], liste[i]
    return liste

# Exemple d'utilisation
liste = [64, 25, 12, 22, 11]
print("Liste originale :", liste)
sorted_liste = selection_sort(liste)
print("Liste triée :", sorted_liste)

```

Analyse de la complexité

- **Complexité temporelle** : L'algorithme de tri par sélection a une complexité en temps de O(n^2). Cette complexité provient des deux boucles imbriquées.

- **Complexité spatiale** : Le tri par sélection est un algorithme en place, ce qui signifie qu'il ne nécessite pas d'espace additionnel proportionnel à la taille de la liste, en dehors de quelques variables temporaires. Ainsi, sa complexité en espace est O(1).

- **Stabilité** : Le tri par sélection n'est pas un algorithme stable. Cela signifie que si deux éléments ont la même valeur, leur ordre relatif peut être modifié par l'algorithme.

Exemple étape par étape

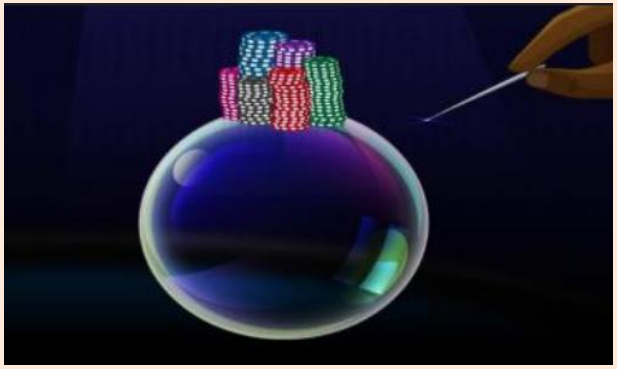
- Prenons un exemple pour illustrer le fonctionnement de l'algorithme :
- Liste originale : [64, 25, 12, 22, 11]
- **Première itération (i = 0)** :
  - Rechercher le minimum dans [64, 25, 12, 22, 11]. Le minimum est 11.
  - Échanger 11 avec 64.
  - Liste après échange : [11, 25, 12, 22, 64].
- **Deuxième itération (i = 1)** :
  - Rechercher le minimum dans [25, 12, 22, 64]. Le minimum est 12.
  - Échanger 12 avec 25.
  - Liste après échange : [11, 12, 25, 22, 64].
- **Troisième itération (i = 2)** :
  - Rechercher le minimum dans [25, 22, 64]. Le minimum est 22.
  - Échanger 22 avec 25.
  - Liste après échange : [11, 12, 22, 25, 64].
- **Quatrième itération (i = 3)** :
  - Rechercher le minimum dans [25, 64]. Le minimum est 25.
  - Pas d'échange nécessaire (élément déjà à sa place).
  - Liste après échange : [11, 12, 22, 25, 64].
- **Cinquième itération (i = 4)** :
  - Un seul élément restant (64), déjà trié.
- Liste triée : [11, 12, 22, 25, 64]
- Le tri par sélection est donc simple à comprendre et à implémenter, mais il n'est pas efficace pour les grandes listes en raison de sa complexité temporelle quadratique.

Conclusion

Le tri par sélection est donc simple à comprendre et à implémenter, mais il n'est pas efficace pour les grandes listes en raison de sa complexité temporelle quadratique.

Algorithme de tri (Bubble Sort)

Analyse et complexité



Principe de l’algorithme

Le tri à bulles est un algorithme de tri simple mais inefficace pour les grandes listes. Son principe repose sur les étapes suivantes :

1. Parcourir la liste du début à la fin.
2. Comparer chaque paire d'éléments adjacents et les échanger s'ils sont dans le mauvais ordre (le plus grand avant le plus petit).
3. Répéter le processus pour chaque élément de la liste jusqu'à ce que la liste soit triée.

À chaque passage à travers la liste, le plus grand élément "bulle" vers la fin de la liste. L'algorithme s'arrête lorsque plus aucun échange n'est nécessaire.

04_bubble.py - Code en Python

02_Bubble.py > bubble_sort

```python
def bubble_sort(liste):
    n = len(liste)
    for i in range(n):
        # Déclare une variable pour suivre si des échanges ont été faits
        swapped = False
        # Parcourt la liste jusqu'à n-i-1 car les derniers i éléments sont déjà triés
        for j in range(0, n-i-1):
            if liste[j] > liste[j+1]:
                # Échange les éléments s'ils sont dans le mauvais ordre
                liste[j], liste[j+1] = liste[j+1], liste[j]
                swapped = True
        # Si aucun échange n'a été fait, la liste est déjà triée
        if not swapped:
            break
    return liste

# Exemple d'utilisation
liste = [64, 25, 12, 22, 11]
print("Liste originale :", liste)
sorted_liste = bubble_sort(liste)
print("Liste triée :", sorted_liste)
```

Analyse de la complexité

- **Complexité temporelle** :
  - **Meilleur cas** : O(n). Cela se produit lorsque la liste est déjà triée. L'algorithme ne fait qu'un seul passage et détecte qu'aucun échange n'est nécessaire.
  - **Cas moyen et pire cas** : O(n^2). Dans ces cas, chaque élément est comparé avec chaque autre élément, résultant en une complexité quadratique.

- **Complexité spatiale** : O(1). Le tri à bulles est un algorithme en place, ce qui signifie qu'il trie la liste sans nécessiter de mémoire additionnelle proportionnelle à la taille de la liste, à part quelques variables temporaires pour les échanges.

- **Stabilité** : Le tri à bulles est un algorithme stable. Cela signifie que si deux éléments ont la même valeur, leur ordre relatif est conservé.

Exemple étape par étape

- Prenons un exemple pour illustrer le fonctionnement de l'algorithme :
- Liste originale : [64, 25, 12, 22, 11]
- **Première itération (i = 0)** :
  - Comparer 64 et 25, échanger. Liste : [25, 64, 12, 22, 11]
  - Comparer 64 et 12, échanger. Liste : [25, 12, 64, 22, 11]
  - Comparer 64 et 22, échanger. Liste : [25, 12, 22, 64, 11]
  - Comparer 64 et 11, échanger. Liste : [25, 12, 22, 11, 64]
- **Deuxième itération (i = 1)** :
  - Comparer 25 et 12, échanger. Liste : [12, 25, 22, 11, 64]
  - Comparer 25 et 22, échanger. Liste : [12, 22, 25, 11, 64]
  - Comparer 25 et 11, échanger. Liste : [12, 22, 11, 25, 64]
  - 64 est déjà à sa place correcte.
- **Troisième itération (i = 2)** :
  - Comparer 12 et 22, pas d'échange. Liste : [12, 22, 11, 25, 64]
  - Comparer 22 et 11, échanger. Liste : [12, 11, 22, 25, 64]
  - 25 et 64 sont déjà à leur place correcte.
- **Quatrième itération (i = 3)** :
  - Comparer 12 et 11, échanger. Liste : [11, 12, 22, 25, 64]
  - 22, 25 et 64 sont déjà à leur place correcte.
- **Cinquième itération (i = 4)** :
  - Un seul élément restant (12), déjà trié.
- Liste triée : [11, 12, 22, 25, 64]

Conclusion

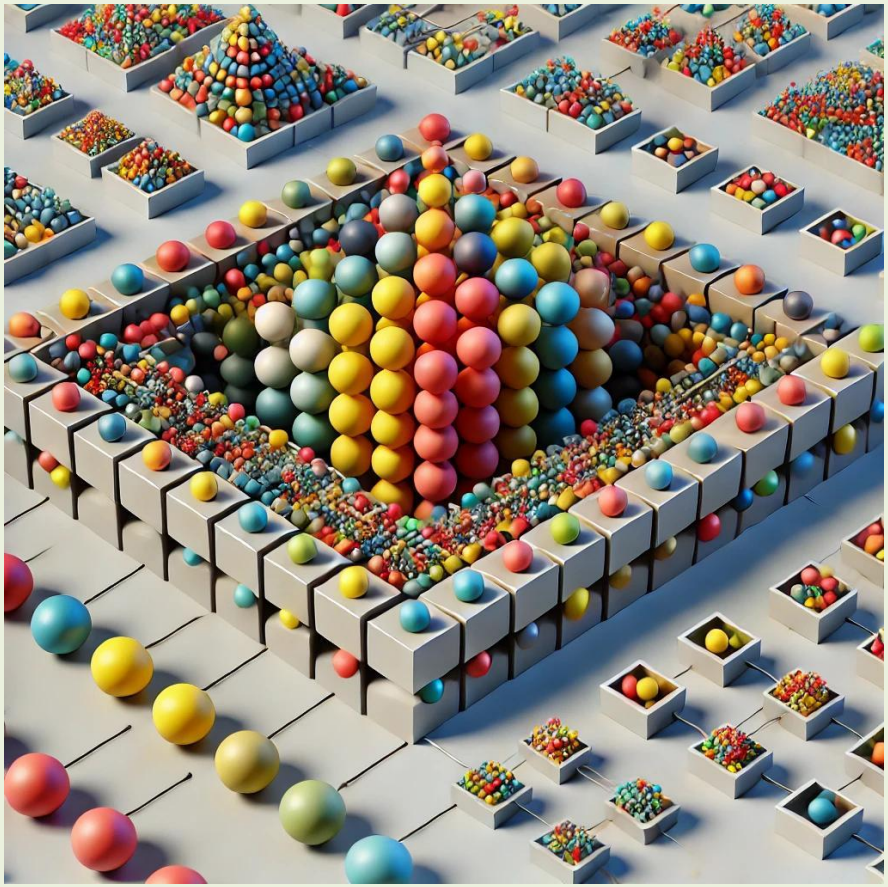
Le tri à bulles est simple à comprendre et à implémenter, ce qui en fait un bon choix pour des fins pédagogiques et pour des listes très petites.

Cependant, en raison de sa complexité temporelle quadratique, il n'est pas adapté pour trier de grandes listes en pratique.

Pour des applications réelles, il est préférable d'utiliser des algorithmes de tri plus efficaces, comme Timsort (utilisé par sort() en Python) qui a une complexité de O(nlogn).

Algorithme de tri (Quick Sort)

Analyse et complexité



Principe de l'algorithme

Le tri rapide, ou quicksort, est un algorithme de tri efficace et très utilisé. Son principe repose sur la technique du "diviser pour régner". Voici les étapes principales :

1. **Choisir un pivot** : Sélectionner un élément de la liste comme pivot.
2. **Partitionner** : Réorganiser les éléments de la liste de manière à ce que tous les éléments inférieurs au pivot soient à sa gauche et tous les éléments supérieurs au pivot soient à sa droite.
3. **Récursion** : Appliquer récursivement le même processus aux sous-listes de gauche et de droite.

Le choix du pivot et la méthode de partitionnement peuvent varier, mais l'idée de base reste la même.

05_quick-sort.py - Code en Python

03_Quick_Sort.py > ...

```python
def quicksort(liste):
    def partition(low, high):
        pivot = liste[high]  # Choisir le pivot
        i = low - 1       # Indice du plus petit élément
        for j in range(low, high):
            if liste[j] <= pivot:
                i = i + 1
                liste[i], liste[j] = liste[j], liste[i]
        liste[i + 1], liste[high] = liste[high], liste[i + 1]
        return i + 1

    def quick_sort_recursive(low, high):
        if low < high:
            pi = partition(low, high)
            quick_sort_recursive(low, pi - 1)
            quick_sort_recursive(pi + 1, high)

    quick_sort_recursive(0, len(liste) - 1)
    return liste

# Exemple d'utilisation
liste = [64, 25, 12, 22, 11]
print("Liste originale :", liste)
sorted_liste = quicksort(liste)
print("Liste triée :", sorted_liste)
```

Analyse de la complexité

Complexité temporelle :

- **Meilleur cas et cas moyen** : O(nlogn). Cela se produit lorsque le pivot divise toujours le tableau en deux parties à peu près égales, ce qui conduit à une hauteur de récursion de log n et à O(n) opérations de partitionnement par niveau.

- **Pire cas** : O(n2). Cela se produit lorsque le pivot choisi est toujours le plus grand ou le plus petit élément, ce qui entraîne une partition très déséquilibrée. Cependant, cela peut être atténué par une bonne stratégie de choix du pivot (comme le pivot médian ou pivot aléatoire).

Complexité spatiale : O(logn). La complexité en espace est principalement due à la pile de récursion, qui a une profondeur de logn dans le meilleur cas. Dans le pire cas (sans optimisation), la profondeur peut atteindre O(n), mais avec des optimisations, cette profondeur peut être réduite.

Stabilité : Le tri rapide n'est pas un algorithme stable. Cela signifie que si deux éléments ont la même valeur, leur ordre relatif peut être modifié.

Exemple étape par étape

Prenons un exemple pour illustrer le fonctionnement de l'algorithme :
Liste originale : [64, 25, 12, 22, 11]

- **Première partition** :
  - Choisir 11 comme pivot.
  - Réorganiser les éléments par rapport à 11 :
    - [11, 25, 12, 22, 64] (11 est déjà à sa place correcte).
- **Récursion sur les sous-listes** :
  - Sous-liste gauche : [] (aucun élément à gauche de 11).
  - Sous-liste droite : [25, 12, 22, 64].
- **Deuxième partition (sous-liste droite)** :
  - Choisir 64 comme pivot.
  - Réorganiser les éléments par rapport à 64 :
    - [25, 12, 22, 64] (64 est déjà à sa place correcte).
- **Récursion sur les nouvelles sous-listes** :
  - Sous-liste gauche : [25, 12, 22].
  - Sous-liste droite : [].
- **Troisième partition (sous-liste [25, 12, 22])** :
  - Choisir 22 comme pivot.
  - Réorganiser les éléments par rapport à 22 :
    - [12, 22, 25] (22 est déjà à sa place correcte).
- **Récursion sur les nouvelles sous-listes** :
  - Sous-liste gauche : [12].
  - Sous-liste droite : [25].
- Aucune partition supplémentaire n'est nécessaire car chaque sous-liste contient un seul élément.
- Liste triée : [11, 12, 22, 25, 64]

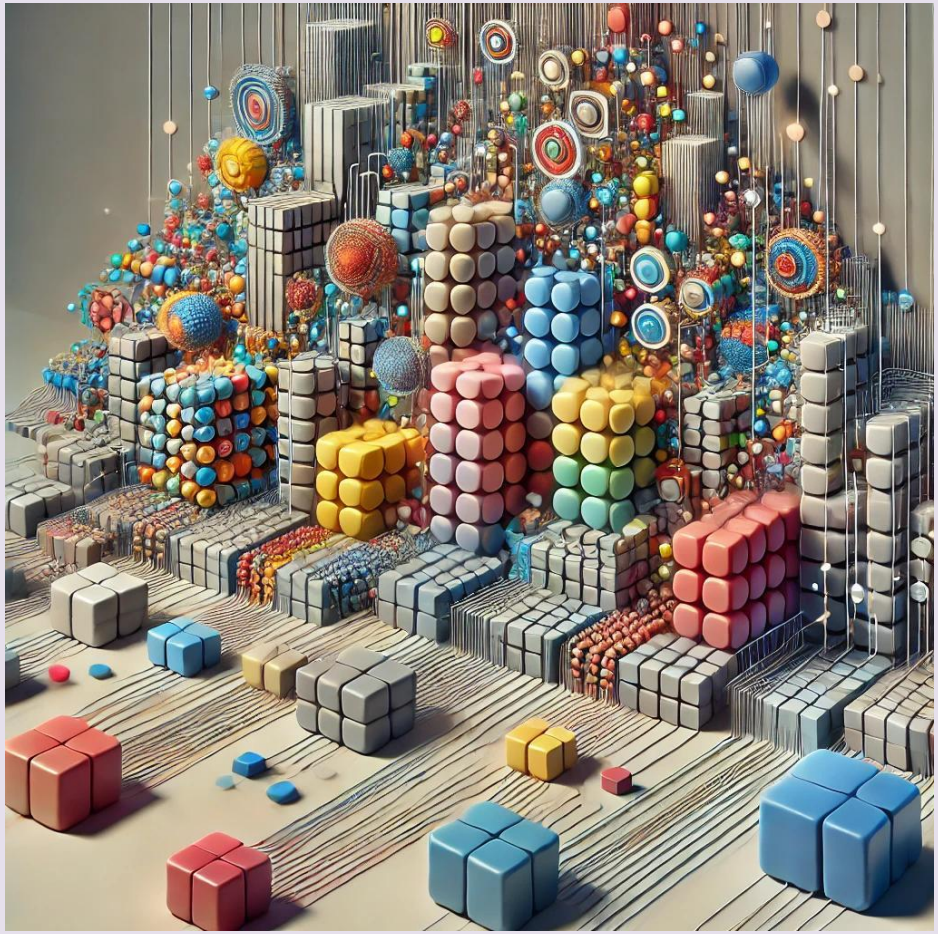
Conclusion

Le tri rapide est un algorithme très performant pour le tri de grandes listes en raison de sa complexité temporelle moyenne de O(nlogn).

Cependant, il peut avoir un pire cas de O(n2), ce qui peut être atténué par une bonne stratégie de choix du pivot.

Malgré sa complexité potentiellement élevée dans le pire cas, il est généralement plus rapide en pratique que d'autres algorithmes de tri de complexité O(nlogn), comme le tri fusion, en raison de son faible overhead de mémoire et de ses bonnes performances sur les systèmes de cache modernes.

Algorithme : Tri Fusion



Principe de l'algorithme « Tri fusion »

Le tri fusion, ou mergesort, est un algorithme de tri efficace basé sur la technique du "diviser pour régner". Il fonctionne en divisant la liste en sous-listes plus petites, en les triant de manière récursive, puis en fusionnant les sous-listes triées pour obtenir la liste finale triée.

Voici les étapes principales :

1. **Diviser** : Diviser la liste en deux sous-listes de tailles à peu près égales.
2. **Récursion** : Appliquer récursivement le tri fusion sur chaque sous-liste.
3. **Fusionner** : Fusionner les deux sous-listes triées en une seule liste triée.

06-tri-fusion.py - Code en Python

04_Tri_Fusion.py > ...

```python
def merge_sort(liste):
    if len(liste) <= 1:
        return liste

    # Diviser la liste en deux sous-listes
    mid = len(liste) // 2
    left_half = merge_sort(liste[:mid])
    right_half = merge_sort(liste[mid:])

    # Fusionner les deux sous-listes triées
    return merge(left_half, right_half)

def merge(left, right):
    sorted_list = []
    i = j = 0

    # Fusionner les sous-listes en comparant les éléments
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1

    # Ajouter les éléments restants des sous-listes
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])

    return sorted_list

# Exemple d'utilisation
liste = [64, 25, 12, 22, 11]
print("Liste originale :", liste)
sorted_liste = merge_sort(liste)
print("Liste triée :", sorted_liste)
```

Analyse de la complexité

- **Complexité temporelle** : O(nlogn)
  - **Meilleur cas, cas moyen et pire cas** : Le tri fusion a une complexité temporelle de O(nlogn)O(n \log n)O(nlogn) dans tous les cas. La liste est divisée en deux parties jusqu'à ce que chaque sous-liste contienne un seul élément, nécessitant logn\log nlogn divisions, et chaque élément est comparé et fusionné en O(n)O(n)O(n).
- **Complexité spatiale** : O(n). Le tri fusion nécessite de l'espace supplémentaire pour les sous-listes temporaires utilisées pendant la fusion. La complexité spatiale est donc proportionnelle à la taille de la liste initiale.
- **Stabilité** : Le tri fusion est un algorithme stable. Cela signifie que si deux éléments ont la même valeur, leur ordre relatif est conservé.

Exemple étape par étape

Prenons un exemple pour illustrer le fonctionnement de l'algorithme :
Liste originale : [64, 25, 12, 22, 11]

1/ Diviser :

Diviser la liste en deux sous-listes : [64, 25] et [12, 22, 11].

2/ Récursion :

Appliquer récursivement le tri fusion sur chaque sous-liste :

[64, 25] est divisée en [64] et [25].

[12, 22, 11] est divisée en [12] et [22, 11], qui est ensuite divisée en [22] et [11].

3/ Fusionner :

Fusionner [64] et [25] pour obtenir [25, 64].

Fusionner [22] et [11] pour obtenir [11, 22].

Fusionner [12] et [11, 22] pour obtenir [11, 12, 22].

Fusionner [25, 64] et [11, 12, 22] pour obtenir [11, 12, 22, 25, 64].

Liste triée : [11, 12, 22, 25, 64]

Conclusion

Le tri fusion est un algorithme de tri très performant, surtout pour les grandes listes, grâce à sa complexité temporelle de O(nlogn) dans tous les cas.

Cependant, il nécessite de l'espace supplémentaire proportionnel à la taille de la liste, ce qui peut être un inconvénient par rapport à d'autres algorithmes en place comme le tri rapide.

Le tri fusion est également stable, ce qui peut être un avantage dans certaines applications où l'ordre relatif des éléments doit être conservé.

Algorithme de tri

Mise en pratique

Implémentation des Algorithmes de Tri Discutés

Nous allons implémenter les algorithmes de tri par sélection, par bulles, rapide, fusion et par insertion en Python.

Voir 10_algorithmes.py

Comparaison des Performances sur des Jeux de Données

Nous allons comparer les performances de ces algorithmes sur différents jeux de données :

Une liste triée

Une liste inversée

Une liste aléatoire

Une grande liste aléatoire.

Performances

Performances en microseconde 10^{-6}

Algo :	Liste Trié	Liste Inversée	Liste Random	Liste random (large
Selection Sort	1018	0	495	27459
Bubble Sort	0	508	508	57941
Quick Sort	0	0	0	1512
Merge Sort	487	0	509	1988
Insertion Sort	0	504	494	28985

Cas d'Utilisation dans la Vie Réelle

Gestion des Stocks et Inventaires :

Exemple : Trier les articles d'un magasin par prix ou par nom pour faciliter la recherche et l'analyse.

Tri des Contacts dans un Carnet d'Adresses :

Exemple : Trier les noms des contacts par ordre alphabétique pour un accès rapide.

Organisation des Fichiers :

Exemple : Trier les fichiers dans un dossier pour faciliter la navigation et la gestion.

Affichage des Résultats de Recherche :

Exemple : Trier les résultats de recherche sur un site de commerce électronique par pertinence, popularité ou prix.

Traitement de Données en Temps Réel :

Exemple : Trier les données de capteurs dans une usine pour une analyse en temps réel.

Conclusion

Le tri est une opération fondamentale en informatique, utilisée dans de nombreuses applications réelles.

Comprendre et implémenter différents algorithmes de tri permet de choisir l'algorithme le plus adapté en fonction des besoins et des contraintes spécifiques.

Les algorithmes de tri étudiés ici, bien que simples, offrent une base solide pour aborder des concepts plus avancés et des algorithmes plus sophistiqués.

Exercice : Tri d’objet

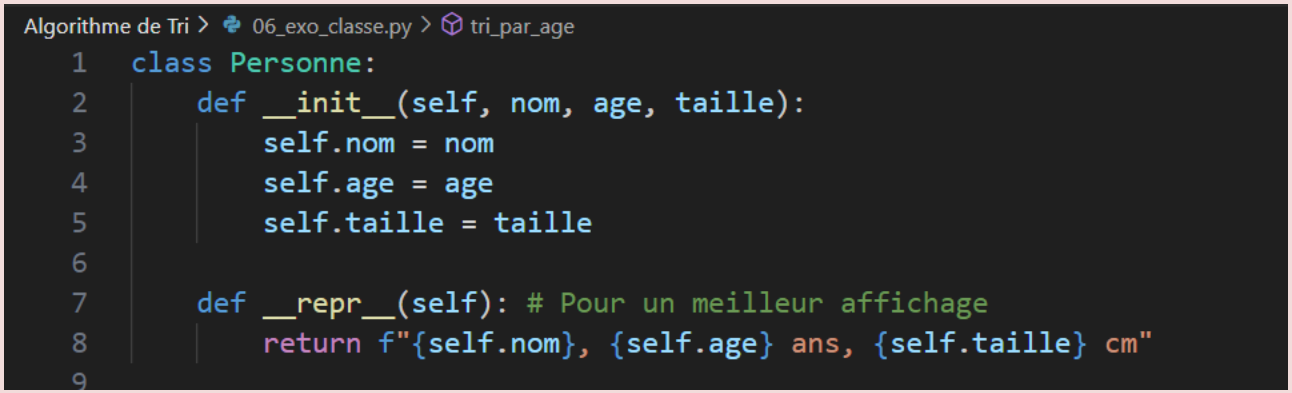
20_exo_classe.py
Tri de Listes d'Objets avec Différents Attributs

Supposons que vous avez une liste d'objets Personne définis par les attributs nom, age, et taille.

Exercice 1.1 : Implémentation de la Classe Personne

Implémentez la classe Personne avec les attributs nom, age, et taille.

20_exo_classe.py
Class personne : solution



20_exo_classe.py
Tri par attribut

Exercice 1.2

A la suite de l'exercice 1.1

Implémentez une fonction de tri qui trie une liste de Personne par age en utilisant le tri par insertion.

L'utilisation sera faite de cette manière :

```
# Exemple d'utilisation
personnes = [Personne("Alice", 25, 165), Personne("Bob", 20, 175), Personne("Charlie", 23, 180)]
print(tri_par_age(personnes))
```

20_exo_classe.py
Exercice 1.2 : solution

```python
def tri_par_age(personnes):
    for i in range(1, len(personnes)):
        key = personnes[i]
        j = i - 1
        while j >= 0 and key.age < personnes[j].age:
            personnes[j + 1] = personnes[j]
            j -= 1
        personnes[j + 1] = key
    return personnes

```

21_exo_fusion.py
Tri par Plusieurs Attributs

Exercice 1.3

Implémentez une fonction de tri qui trie une liste de Personne d'abord par age, puis par taille en utilisant le tri fusion.