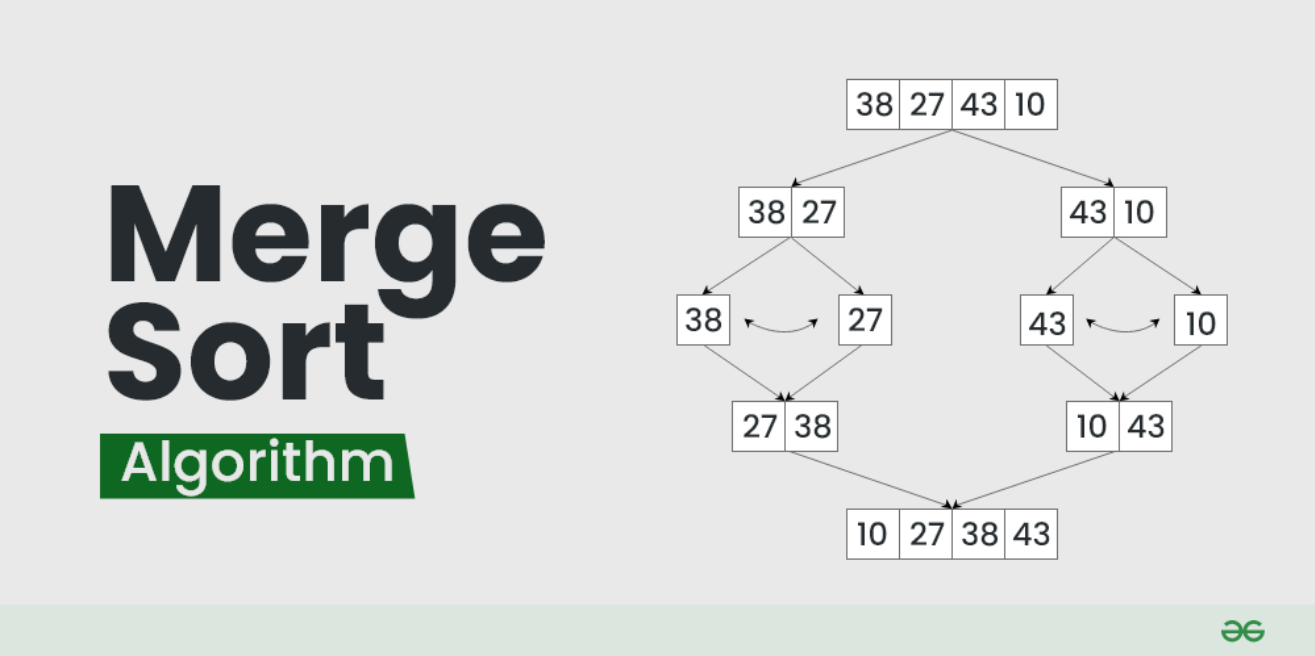
Détails sur l'algorithme de tri des listes chaînées

Implémentation du tri sur une liste chaînée

Merge Sort appliqué aux listes chaînées

Merge Sort est un algorithme de tri efficace qui fonctionne bien avec les listes chaînées. Il repose sur la division récursive de la liste en deux moitiés, le tri de chaque moitié, puis la fusion des listes triées.



Implémentation en Python

Merge sort sur un tableau

```
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    # Create temp arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]

    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = left  # Initial index of merged subarray
```

```
    # Merge the temp arrays back
    # into arr[left..right]
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[],
    # if there are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[],
    # if there are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2

        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)

def print_list(arr):
    for i in arr:
        print(i, end=" ")
    print()

# Driver code
if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is")
    print_list(arr)

    merge_sort(arr, 0, len(arr) - 1)

    print("\nSorted array is")
    print_list(arr)
```

Exercice: Faire un merge sort sur une liste chaînée

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

    def merge_sort(self):
        if not self.head or not self.head.next:
            return self.head  # Déjà trié ou vide

        mid = self.get_middle(self.head)
        right_head = mid.next
        mid.next = None  # Coupe la liste en deux

        left = self.merge_sort_recursive(self.head)
        right = self.merge_sort_recursive(right_head)

        self.head = self.merge(left, right)  # Fusionne les deux moitiés

    def merge_sort_recursive(self, head):
        if not head or not head.next:
            return head

        mid = self.get_middle(head)
        right_head = mid.next
        mid.next = None

        left = self.merge_sort_recursive(head)
        right = self.merge_sort_recursive(right_head)

        return self.merge(left, right)
```

```
    def merge(self, left, right):
        if not left:
            return right
        if not right:
            return left

        if left.data < right.data:
            result = left
            result.next = self.merge(left.next, right)
        else:
            result = right
            result.next = self.merge(left, right.next)

        return result

    def get_middle(self, head):
        if not head:
            return head

        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        return slow
```

Explication de `get_middle()`

La fonction `get_middle()` utilise la **méthode des deux pointeurs** (*slow and fast pointer technique*) pour trouver le **milieu** d'une liste chaînée de manière optimale.

Principe

- **Un pointeur lent (`slow`)** avance **d'un nœud à la fois**.
- **Un pointeur rapide (`fast`)** avance **de deux nœuds à la fois**.
- Lorsque `fast` atteint la fin de la liste, `slow` est **au milieu** de la liste.

Pourquoi cette méthode est efficace ?

- **Complexité en O(n)** : On parcourt la liste **une seule fois**.
- **Optimisée pour les listes chaînées** : Contrairement à un accès indexé (`arr[len(arr)//2]` pour un tableau), ici, on évite un second passage pour compter les éléments.

Explication de `if not self.head or not self.head.next:`

Dans la méthode `merge_sort()`, cette condition permet de gérer deux cas particuliers :

1. **La liste est vide (`self.head is None`)** → On retourne immédiatement `None`.

2. **La liste contient un seul élément (`self.head.next is None`)** → Une liste d'un seul élément est déjà triée, donc on la retourne telle quelle.

Ceci évite d'exécuter inutilement l'algorithme de tri sur une liste déjà triée ou vide.

Comparaison avec d'autres types de listes en Python

1. Liste chaînée vs. liste Python (`list`)

Critère	Liste Chaînée	Liste Python (`list`)
Accès indexé	❌ Lent (O(n))	☑ Rapide (O(1))
Insertion/Suppression	☑ Rapide (O(1) en début)	❌ Lent (O(n) au milieu)
Utilisation mémoire	☑ Économique (pas de redimensionnement)	❌ Peut occuper plus d'espace
Structure dynamique	☑ Extensible facilement	❌ Nécessite une allocation en bloc

2. Liste chaînée vs. `deque` (double-ended queue)

Python offre aussi `collections.deque`, une structure optimisée pour les ajouts et suppressions aux extrémités.

Critère	Liste Chaînée	`deque`
Ajout/suppression en début	☑ O(1)	☑ O(1)
Accès indexé	❌ O(n)	☑ O(1)
Stockage mémoire	☑ Minimal	❌ Plus élevé

Cas concrets d'utilisation des listes chaînées

1. **Implémentation de structures de données avancées** : Les piles et files d'attente (FIFO/LIFO) sont souvent basées sur des listes chaînées.
2. **Manipulation de grandes quantités de données dynamiques** : Lorsqu'on ne connaît pas la taille des données à l'avance, une liste chaînée peut être plus efficace qu'une liste Python classique.
3. **Systèmes nécessitant une gestion flexible de la mémoire** : Comme dans les systèmes embarqués ou les bases de données, où la fragmentation mémoire est une contrainte importante.
4. **Optimisation des algorithmes gourmands en insertion/suppression** : Par exemple, dans les algorithmes de type cache (LRU Cache) ou les simulations temps réel.