Merge Sort:

Implemente um algoritmo eficiente para realizar a ordenação:

O Merge Sort é um algoritmo de ordenação eficiente baseado em dividir para conquistar. Ele divide a lista em metades recursivamente, ordena cada metade e depois mescla as metades ordenadas para produzir a lista final ordenada. O Merge Sort é estável e possui complexidade de tempo O(n log n), tornando-o eficiente para grandes conjuntos de dados.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

    return arr

arr = [64, 34, 25, 12, 99, 21, 777, 22, 11, 90]
print("Antes da ordenação:", arr)
print("Após Merge Sort:", merge_sort(arr.copy()))
```

Execução:

```
.exe c:/Users/migue/teste/marge.py
Antes da ordenação: [64, 34, 25, 12, 99, 21, 777, 22, 11, 90]
Após Merge Sort: [11, 12, 21, 22, 25, 34, 64, 90, 99, 777]
PS C:\Users\migue\teste>
```

Documente o algoritmo utilizado e forneça uma análise da sua complexidade temporal:

Merge Sort é um algoritmo de ordenação eficiente que segue o paradigma "dividir para conquistar". Ele divide repetidamente a lista em duas metades, ordena cada metade de forma recursiva e depois combina as duas metades ordenadas. O Merge Sort tem uma complexidade temporal de O(n log n) para todos os casos, tornando-o muito eficiente para listas grandes em comparação com Bubble Sort, Selection Sort e Insertion Sort.

Complexidade Temporal

- Melhor caso: O(n log n).
- Caso médio: O(n log n).
- Pior caso: O(n log n).