Merge Sort

>> Bei **Merge Sort** ist die Grundidee, kleinere Arrays zu sortieren und diese Arrays dann sortiert zusammenzuführen (zu mergen). Merge Sort nutzt dabei Rekursion.

Wie sortiert man ein Array mit Merge Sort?
→ Durch Aufruf von Merge Sort auf beiden Hälften!

```
MergeSort(Array)
├── MergeSort(linke Hälfte)
└── MergeSort(rechte Hälfte)
```

Pseudocode

A. Prüfe Basisfall

Ist das Array einelementig?
Dann ist es bereits sortiert!

B. Teile (Divide)

Teile Array in zwei Hälften.

C. Sortiere rekursiv (Conquer)

Rufe MergeSort auf beiden Hälften auf.

D. Führe zusammen (Combine)

Merge die sortierten Hälften.

Beispiel

```
MergeSort([5, 2, 1, 3, 6, 4])
│
├── MergeSort([5, 2, 1])      ←
│   ├── MergeSort([5])
│   └── MergeSort([2, 1])
│
└── MergeSort([3, 6, 4])
    ├── MergeSort([3])
    └── MergeSort([6, 4])
```

Erste Teilung

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([5, 2, 1])
├── MergeSort([5]) ✓
└── MergeSort([2, 1])      ←
    ├── MergeSort([2])
    └── MergeSort([1])
```

```
A: Basisfall? Nein, 2 Elemente
B: Teile in [2] und [1]
C: Rekursive Aufrufe
```

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([5, 2, 1])
├── MergeSort([5]) ✓
└── MergeSort([2, 1])
    ├── MergeSort([2]) ✓
    ├── MergeSort([1]) ✓
    └── Merge([2], [1]) → [1, 2]   ←
```

```
Vergleiche: 2 mit 1
Ergebnis: [1, 2]
```

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([5, 2, 1])
├── MergeSort([5]) → [5] ✓
├── MergeSort([2, 1]) → [1, 2] ✓
└── Merge([5], [1, 2]) → [1, 2, 5]   ←
```

1. Vergleiche: 5 mit 1 → [1]
2. Vergleiche: 5 mit 2 → [1, 2]
3. Übrige Elemente: 5 → [1, 2, 5]

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([5, 2, 1, 3, 6, 4])
├── MergeSort([5, 2, 1]) → [1, 2, 5] ✓
└── MergeSort([3, 6, 4])                    ←
    ├── MergeSort([3])
    └── MergeSort([6, 4])
```

Linke Hälfte abgeschlossen.
Weiter mit rechter Hälfte.

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([3, 6, 4])
├── MergeSort([3]) → [3] ✓
└── MergeSort([6, 4])          ←
    ├── MergeSort([6])
    └── MergeSort([4])
```

```
A: Basisfall für [3]? Ja ✓
B: Teilung von [6, 4]
```

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([6, 4])
├── MergeSort([6]) → [6] ✓
├── MergeSort([4]) → [4] ✓
└── Merge([6], [4]) → [4, 6]   ←


Vergleiche: 6 mit 4 → [4]
Übrig: 6 → [4, 6]
```

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([3, 6, 4])
├── MergeSort([3]) → [3] ✓
├── MergeSort([6, 4]) → [4, 6] ✓
└── Merge([3], [4, 6]) → [3, 4, 6]    ←
```

1. Vergleiche: 3 mit 4 → [3]
2. Übrig: [4, 6] → [3, 4, 6]

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

Beispiel

```
MergeSort([5, 2, 1, 3, 6, 4])
├── [1, 2, 5] ✓
├── [3, 4, 6] ✓
└── Merge([1, 2, 5], [3, 4, 6])    ←
```

```
Vergleiche
1 mit 3 → [1]
2 mit 3 → [1, 2]
5 mit 3 → [1, 2, 3]
5 mit 4 → [1, 2, 3, 4]
5 mit 6 → [1, 2, 3, 4, 5]
Rest: 6 → [1, 2, 3, 4, 5, 6]
```

A. Array einelementig? Dann ist es sortiert!

B. Teile Array in zwei Hälften.

C. Rufe Merge Sort auf beiden Hälften auf.

D. Merge die sortierten Hälften.

>> **Rekursive Struktur**

Jeder Aufruf folgt dem gleichen Muster.

Vollständige Bearbeitung der linken Seite vor der rechten.

Erstellen einer Schrittliste zum besseren Verständnis

Zeichnen Sie den **Rekursionsbaum**:

- nutzen Sie Einrückungen für die Rekursionstiefe
- markieren Sie Ihren aktuellen Standort im Baum
- nutzen Sie eine hierarchische Nummerierung zur eindeutigen Bezeichnung der Schritte.

```
1. MergeSort([5, 2, 1, 3, 6, 4])
   1A: Basisfall-Check
   1B: Teilen in [5, 2, 1] und [3, 6, 4]
   1.1 MergeSort([5, 2, 1])
       1.1A: Basisfall-Check
       1.1B: Teilen in [5] und [2, 1]
       ...
```

Fälle

Worst-Case: $O(n \log n)$

Teilung von n Elementen und
Rekombination in $\log n$ Schritten

Merges: lineare Laufzeitkomplexität

Best-Case: $\Omega(n \log n)$

Auch bei bereits sortiertem Array
vollständige Durchführung nötig

Vergleich:

Bubble Sort (Worst-Case): $O(n^2)$

Selection Sort (Worst-Case): $O(n^2)$

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

```c
void merge(int arr[], int start, int middle, int end) {
    int i, j, k;
    int n1 = middle - start + 1;
    int n2 = end - middle;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++) L[i] = arr[start + i];
    for (j = 0; j < n2; j++) R[j] = arr[middle + 1 + j];

    i = 0; j = 0; k = start;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++; k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++; k++;
    }
}
```

```c
void mergeSort(int arr[], int start, int end) {
    if (start < end) {
        int middle = start + (end - start) / 2;
        mergeSort(arr, start, middle);
        mergeSort(arr, middle + 1, end);
        merge(arr, start, middle, end);
    }
}
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
    ↑                 ↑
    i                 j

A: [ , , , , , , ]
    ↑
    k
```

Marker:
- i: Position in L
- j: Position in R
- k: Position in A

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
    ↑                 ↑
    i                 j

A: [ , , , , , , ]
    ↑
    k

Solange i < len(L) UND j < len(R):
    Wenn L[i] ≤ R[j]:
        A[k] = L[i]
        i++
    Sonst:
        A[k] = R[j]
        j++
    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
       ↑              ↑
       i              j

A: [1, , , , , , ]
      ↑
      k
```

```
Solange i < len(L) UND j < len(R):
    Wenn L[i] ≤ R[j]:
        A[k] = L[i]
        i++
    Sonst:
        A[k] = R[j]
        j++
    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
       ↑                 ↑
       i                 j

A: [1,2, , , , , ]
        ↑
        k

Solange i < len(L) UND j < len(R):
    Wenn L[i] ≤ R[j]:
        A[k] = L[i]
        i++
    Sonst:
        A[k] = R[j]
        j++
    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
          ↑              ↑
          i              j

A: [1,2,3, , , , ]
          ↑
          k

Solange i < len(L) UND j < len(R):
    Wenn L[i] ≤ R[j]:
        A[k] = L[i]
        i++
    Sonst:
        A[k] = R[j]
        j++
    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
          ↑                ↑
          i                j

A: [1,2,3,4, , , ]
            ↑
            k
```

```
Solange i < len(L) UND j < len(R):
    Wenn L[i] ≤ R[j]:
        A[k] = L[i]
        i++
    Sonst:
        A[k] = R[j]
        j++
    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
            ↑             ↑
            i             j

A: [1,2,3,4,5, , ]
              ↑
              k
```

```
Solange i < len(L) UND j < len(R):
    Wenn L[i] ≤ R[j]:
        A[k] = L[i]
        i++
    Sonst:
        A[k] = R[j]
        j++
    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
            ↑                 ↑
            i                 j    ← j == len(R)

A: [1,2,3,4,5,6, ]
                ↑
                k
```

```
Solange i < len(L) UND j < len(R):
    Wenn L[i] ≤ R[j]:
        A[k] = L[i]
        i++
    Sonst:
        A[k] = R[j]
        j++
    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
            ↑                ↑
            i                j

A: [1,2,3,4,5,6, ]
              ↑
              k
```

Fortsetzung des merge-Algorithmus:

```
Während i < len(L):    Während j < len(R):
    A[k] = L[i]            A[k] = R[j]
    i++                    j++
    k++                    k++
```

Der Merge-Algorithmus – Beispiel

```
L: [1, 3, 5, 7]   R: [2, 4, 6]
                ↑             ↑
                i             j

A: [1,2,3,4,5,6,7]
                 ↑
                 k
```

Fortsetzung des merge-Algorithmus:

```
Während i < len(L):    Während j < len(R):
    A[k] = L[i]            A[k] = R[j]
    i++                    j++
    k++                    k++
```

Der Merge-Algorithmus – Tabelle

Schritt	L[i]	R[j]	Aktion	A[k]	Marker (i,j,k)	Verbleibende Elemente
1	1	2	L[i] → A	1	i=1, j=0, k=1	L:[3,5,7], R:[2,4,6]
2	3	2	R[j] → A	2	i=1, j=1, k=2	L:[3,5,7], R:[4,6]
3	3	4	L[i] → A	3	i=2, j=1, k=3	L:[5,7], R:[4,6]
4	5	4	R[j] → A	4	i=2, j=2, k=4	L:[5,7], R:[6]
5	5	6	L[i] → A	5	i=3, j=2, k=5	L:[7], R:[6]
6	7	6	R[j] → A	6	i=3, j=3, k=6	L:[7], R:[]
7	7	-	L[i] → A	7	i=4, k=7	L:[], R:[]

Rekursiver Algorithmus:

Teilt Array, sortiert Hälften, führt zusammen

Merge kombiniert zwei sortierte Arrays systematisch mit drei Markern

Garantierte Laufzeit O(n log n) in allen Fällen

„Stabiler“ Algorithmus: Bewahrt Reihenfolge gleicher Elemente (manchmal wichtig)

Benötigt temporären Speicher in der Größe des Arrays

Besonders effizient bei großen Datenmengen und externer Sortierung