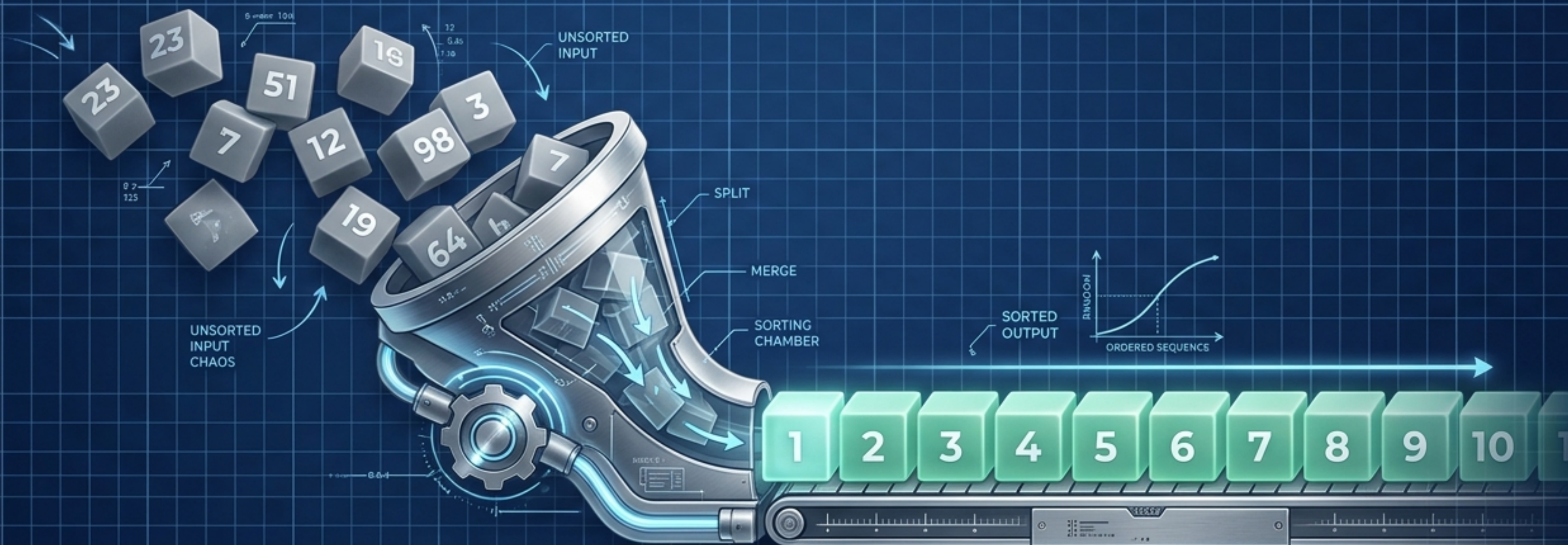


完全に理解する「マージソート」

初心者のための図解アルゴリズム入門



巨大な問題を、解けるサイズまで小さくする

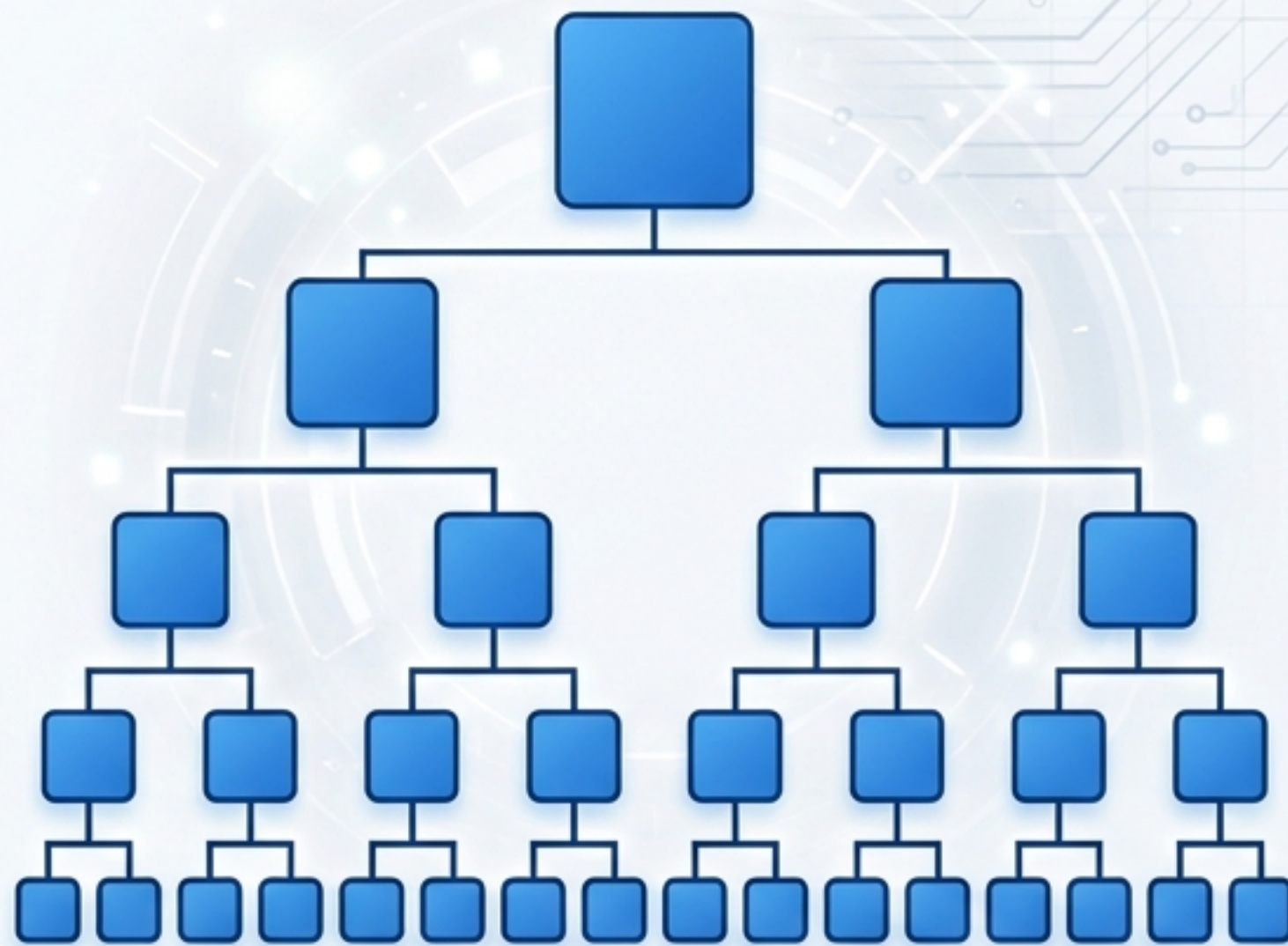
データ全体を一度に並べ替えようとすると、計算量が膨大になります。

マージソートの根本的なアプローチは「分割統治法（Divide and Conquer）」。

解決可能な「要素1つ」のサイズになるまでデータを細かく割り、賢く組み立て直します。

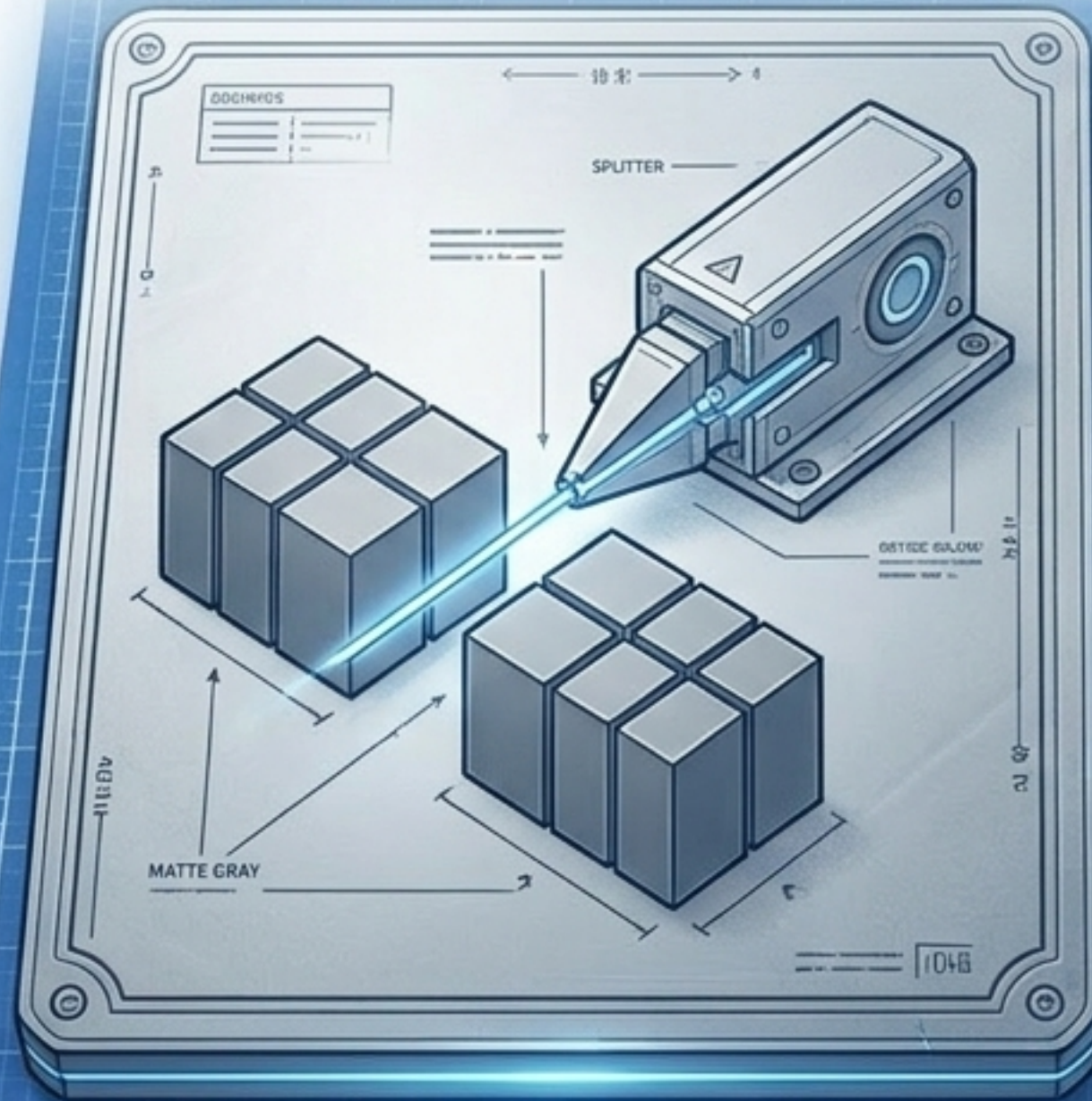


「全体を一度に処理する（非効率）」



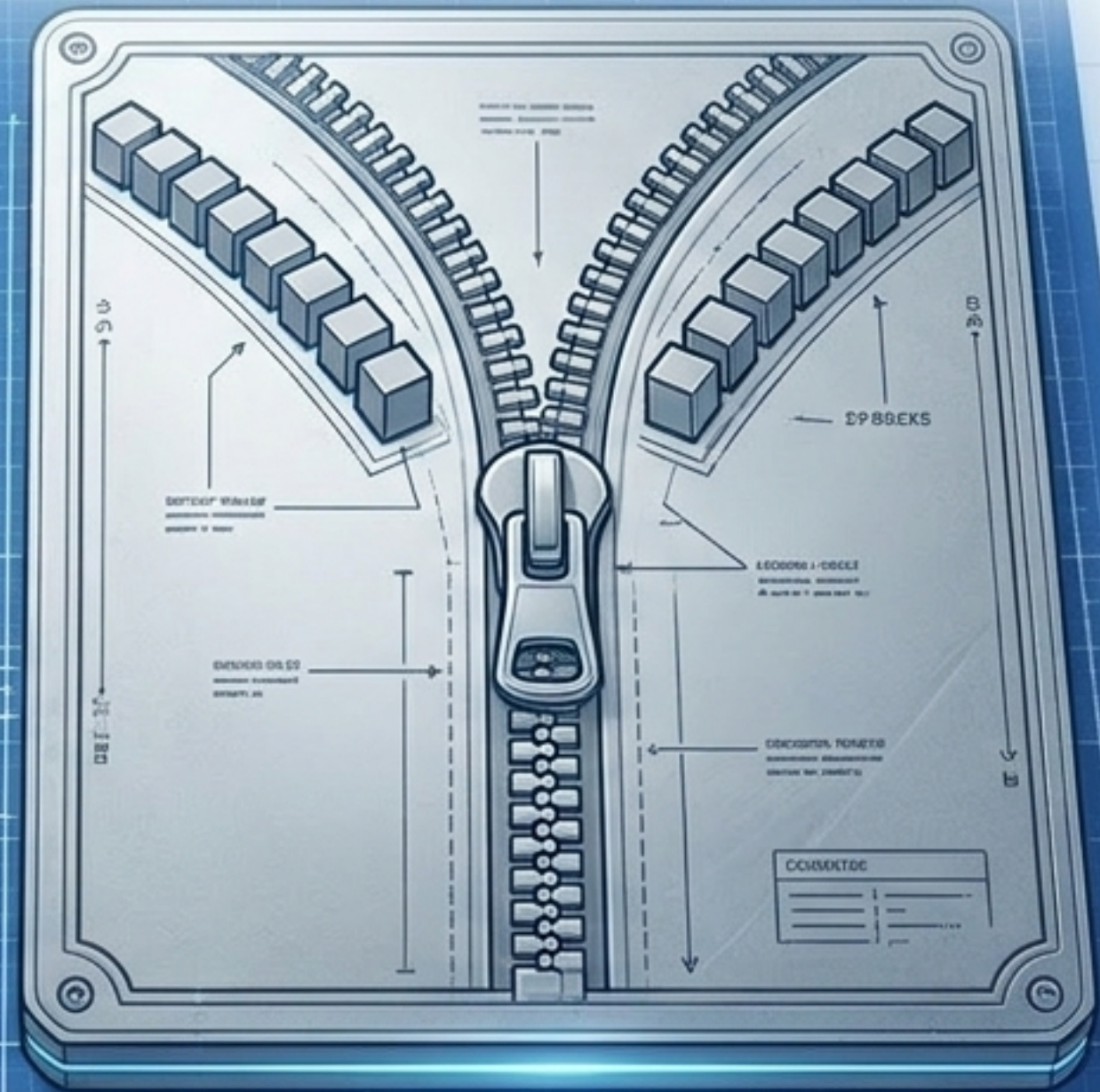
「最小単位に分割して解決する（高効率）」

マージソートを動かす2つの歯車



1. 分割 (Divide)

リストを真ん中で半分に分けます。
これを要素が「1つ」になるまで繰り返します。



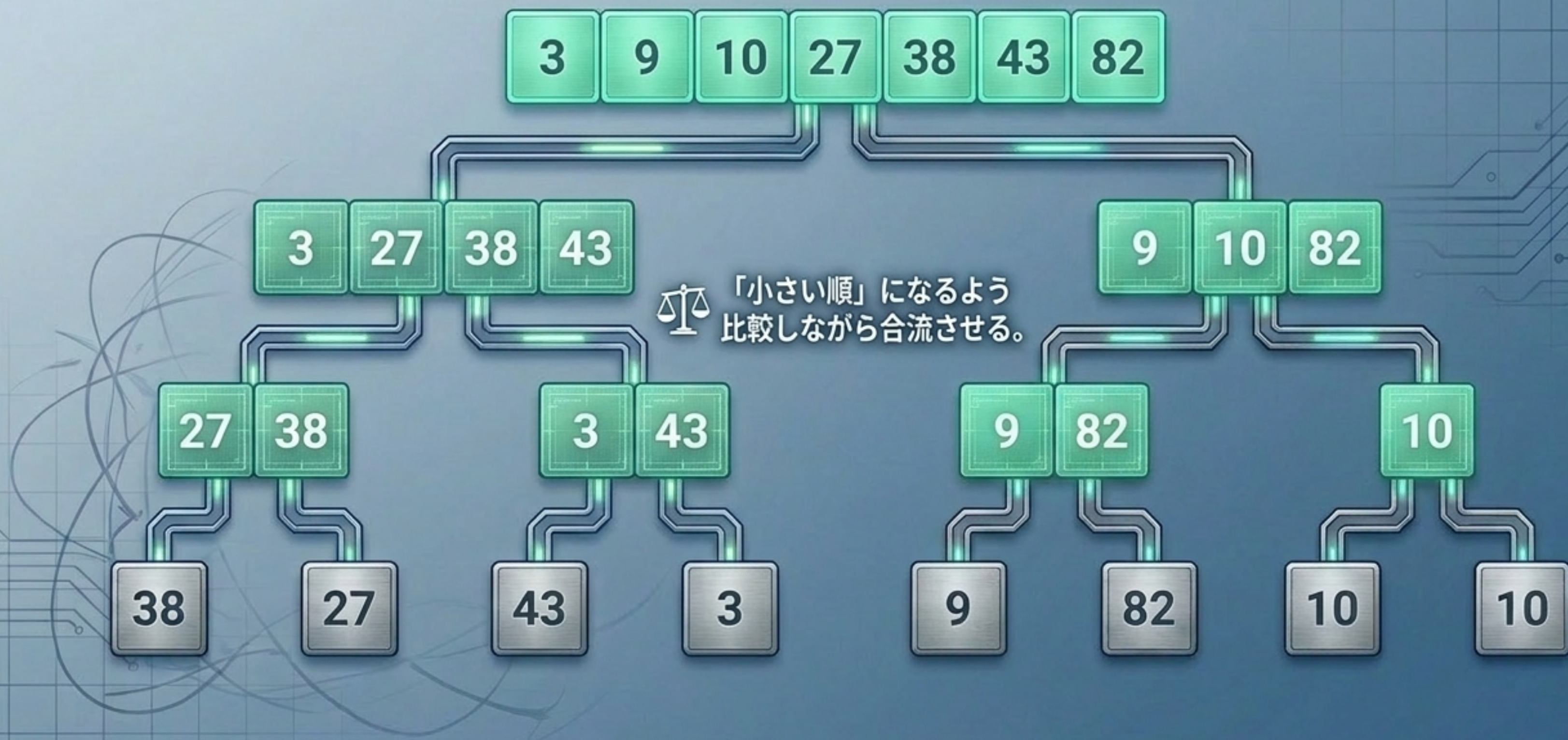
2. 統合 (Merge)

分割されたリストを、小さい順に並ぶように合流させます。
最終的に1つの整列済みリストに戻します。

前半戦：最小単位までデータを「分割」する

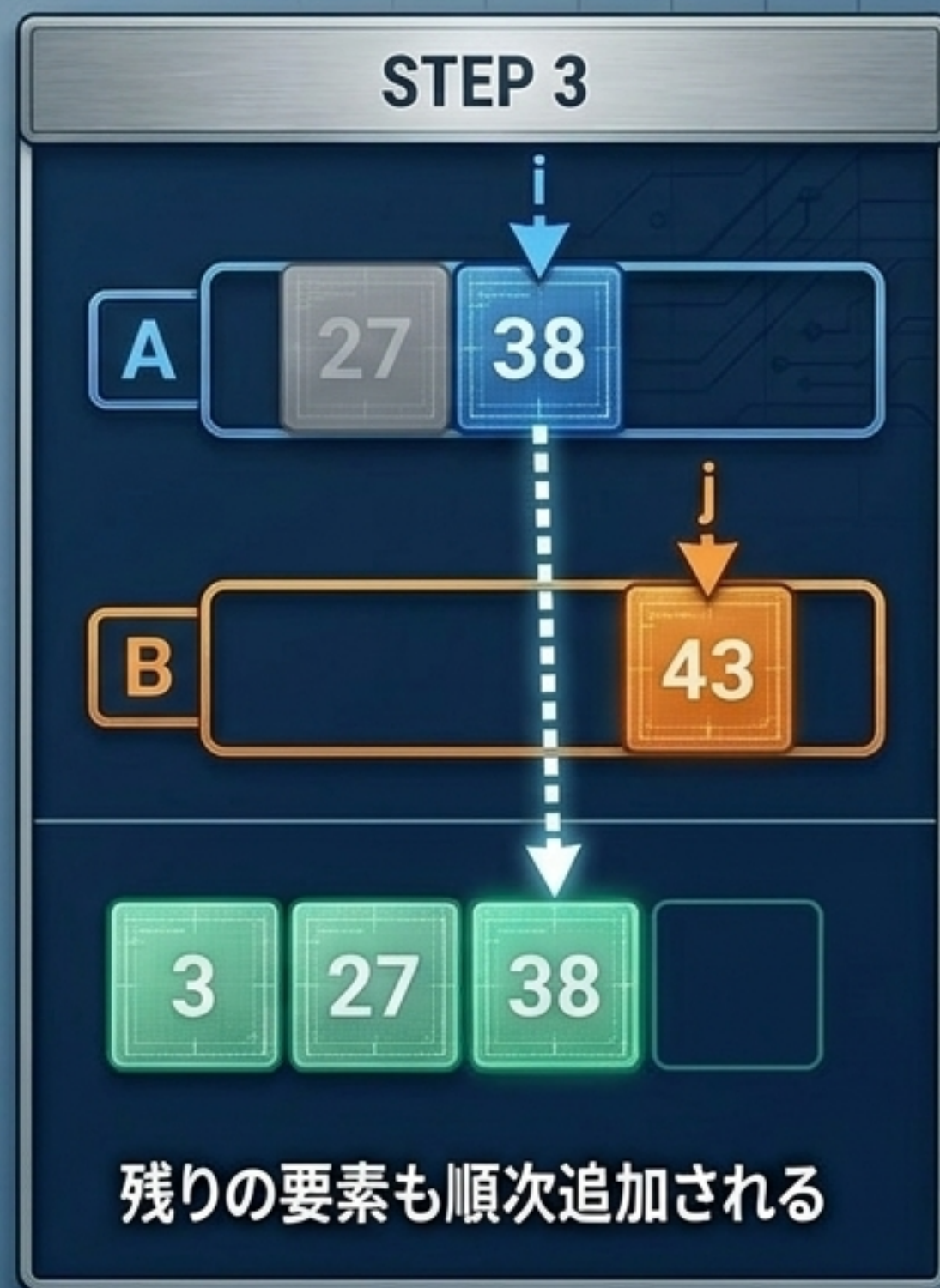
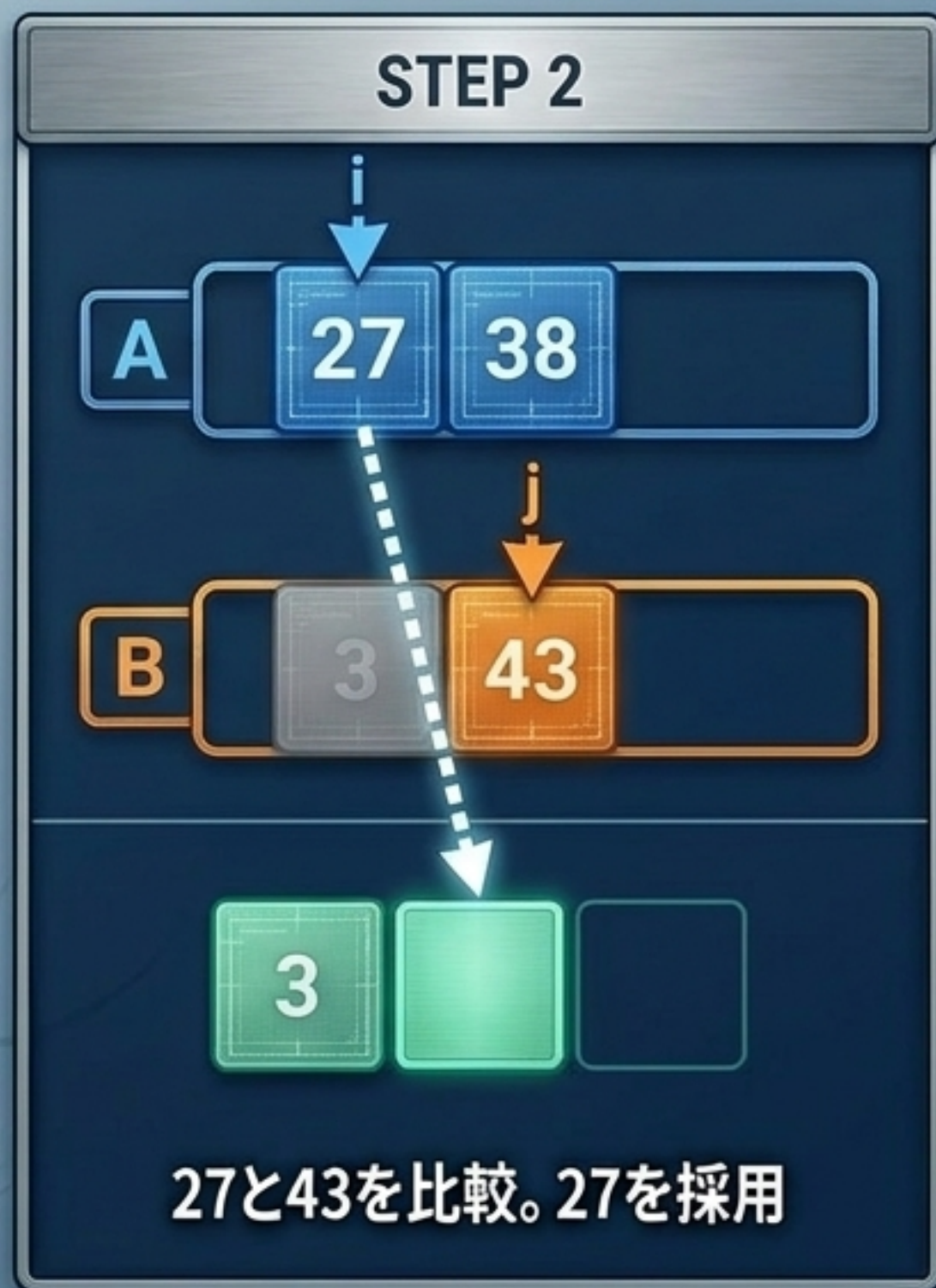


後半戦：整列させながらデータを「統合」する



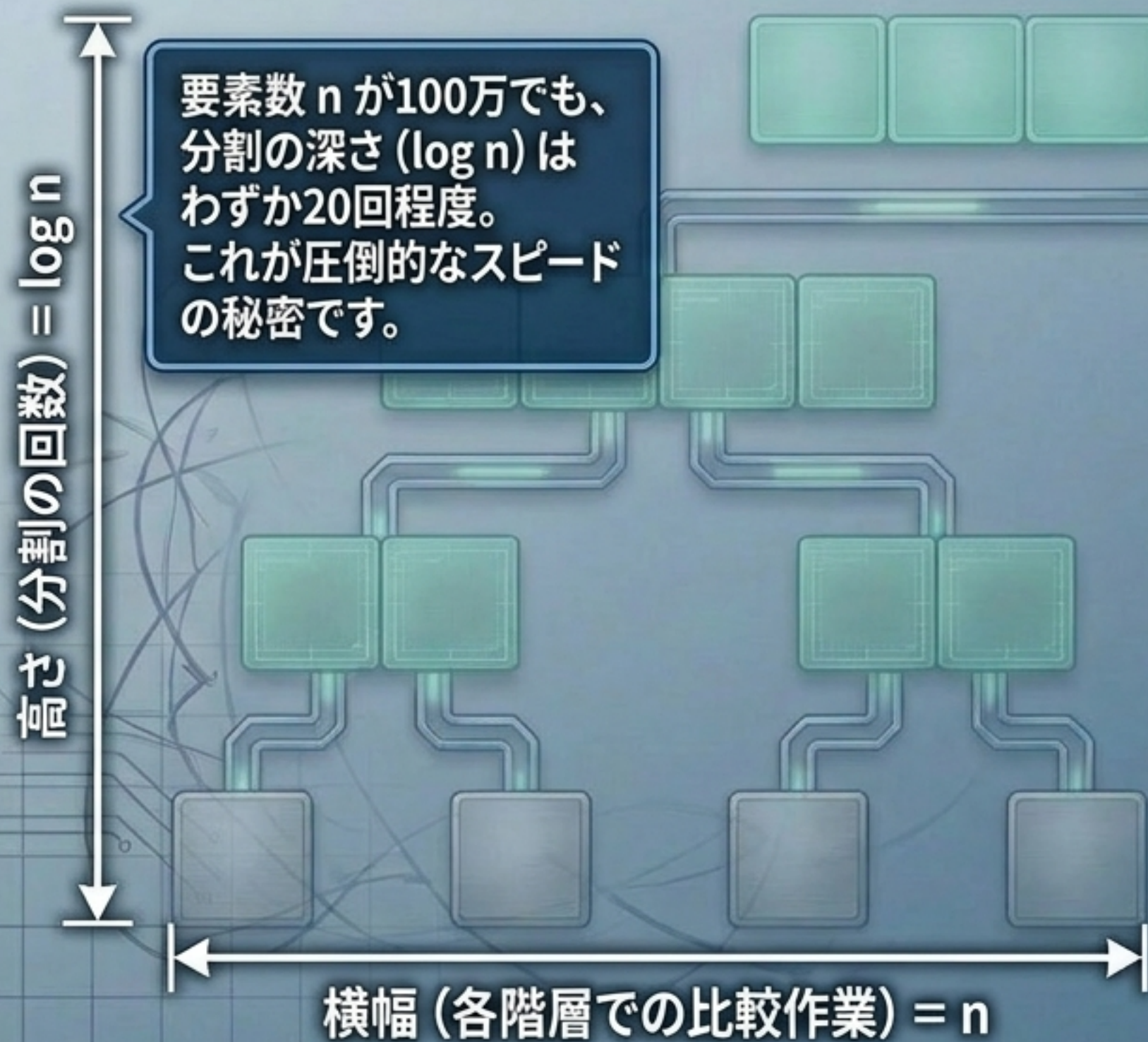
クローズアップ：2つのリストはどう合流するのか？

統合（Merge）処理では、2つの「ポインタ」が先頭の値を比較し、小さい方を新しいリストに落とします。



なぜ速いのか？視覚で理解する $O(n \log n)$

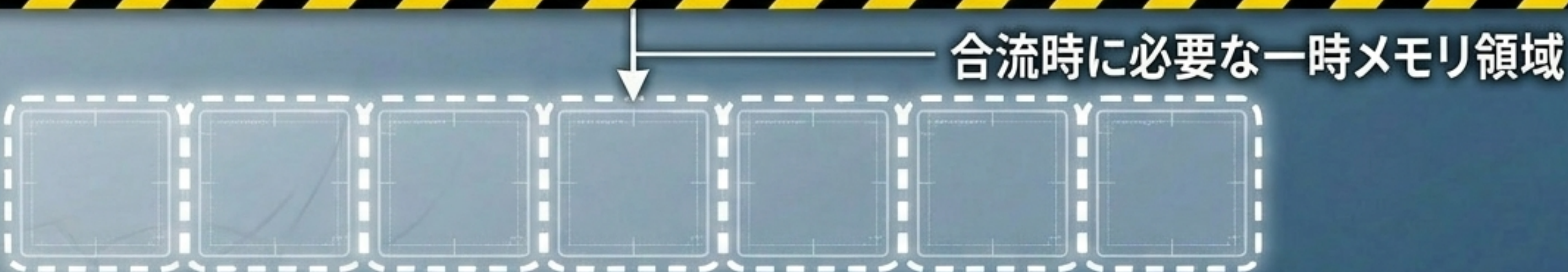
最悪の場合でも、クイックソートのように極端に遅くなることなく、計算時間 $O(n \log n)$ を保証します。



$$\log n \text{ (高さ)} \times n \text{ (横幅)} = O(n \log n)$$

唯一の弱点：一時的なメモリの消費

データを合流（統合）させる際、元の配列と同じサイズの「一時的な空きスペース（補助配列）」が必要になります。空間計算量は $O(n)$ です。



⚠ 組み込みシステムなど、メモリ制限が極端に厳しい環境 (In-place処理が必須の場面) では注意が必要です。

圧倒的な強み：「安定なソート」であること

マージソートは「安定なソート」です。同じ値を持つ要素があった場合、ソート前の「元の順番」を絶対に崩しません。

ソート前



ソート後



合流時に左側の要素を優先することで、
同じ値の順序を厳格に保持します

いつマージソートを使うべきか？（アルゴリズム比較）

	マージソート	クイックソート	ヒープソート
時間計算量 (最悪)	$O(n \log n)$ ✓	$O(n^2)$ ⚠	$O(n \log n)$ ✓
空間計算量 (追加メモリ)	$O(n)$ ⚠	In-place (不要) ✓	In-place (不要) ✓
安定性 (Stability)	安定 (Yes) ★	不安定 (No)	不安定 (No)



結論：「絶対に実行時間をブレさせたくない時」や
「データの元の順序(安定性)を保ちたい時」は、マージソートが最強の選択肢です。

図解からコードへ：Pythonでの実装

```
merge_sort.py × ▶ □ ...  
  
1 def merge_sort(arr):  
2     if len(arr) <= 1: return arr  
3     # (中略: 分割処理)  
4  
5 def merge(left, right):  
6     result = []  
7     i = j = 0  
8     # (中略: 合流処理)  
9
```



自身を呼び出し、
リストを半分に分割し続ける



2つのポインタ (i, j) を使い、
小さい順に result へ
追加していく

完全に理解する「マージソート」(まとめ)



分割統治法で動く

リストを「要素1つ」になるまで半分に分割し、小さい順に合流させながら組み立て直す。



安定した高速処理

最悪のケースでも極端に遅くなることなく、常に計算量 $O(n \log n)$ を保証する。



メモリと安定性のトレードオフ

元の配列と同じサイズの一時メモリを消費する(弱点)が、値の順序を崩さない「安定なソート」である(強み)。