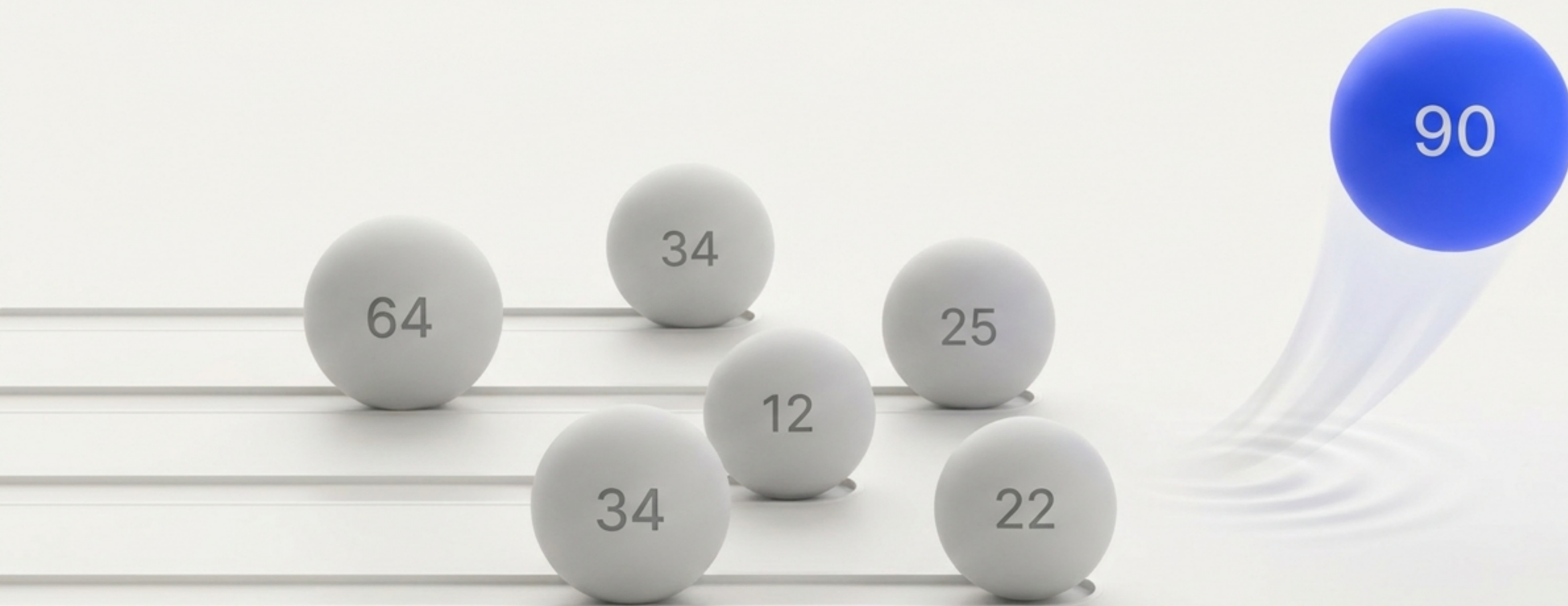


バブルソート入門

アルゴリズムの基本を、視覚的に理解しよう

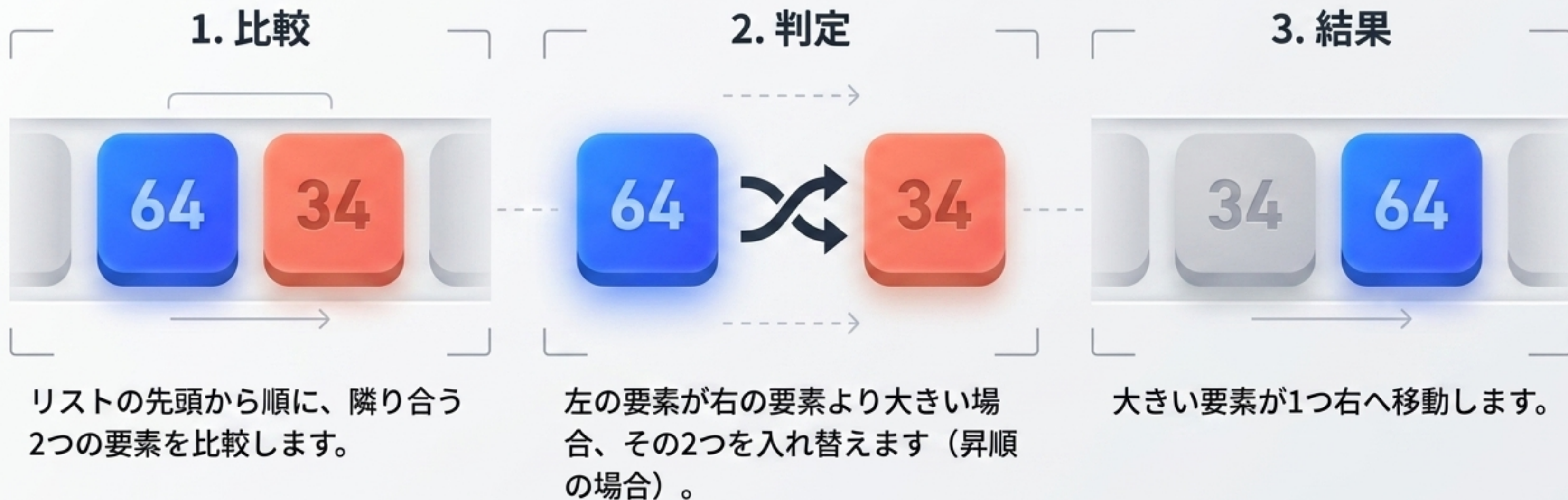


基本ルール：最も大きい要素が「泡」のように浮かび上がる

「リストの端から隣同士を比較し、大きい方を後ろへ送る」



アクションの最小単位：隣り合う2つの比較と交換



〔1回のパス（走査）で、最大値が最後尾に到達する〕



これをリストの最後まで繰り返すと、一番大きな要素が最後に配置されます。

確定した要素を除外し、同じ手順を繰り返す

最後の要素を除いた残りの部分で、再び同じ操作を繰り返します。

パス 1 完了



パス 2 完了

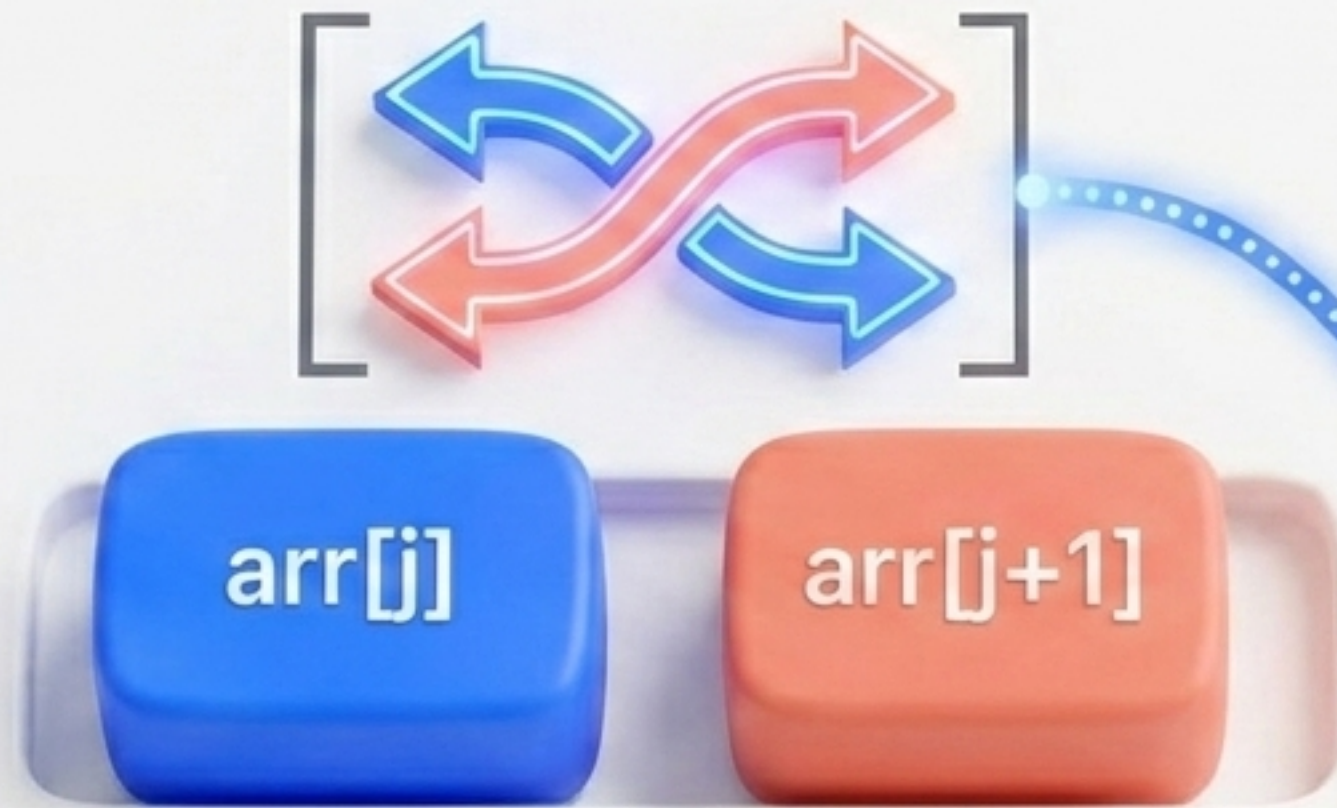


ソート完了



入れ替えが発生しなくなるか、すべての要素が確定するまで続けます。

視覚的な「交換」をPythonコードに翻訳する



隣り合う要素を比較

```
if arr[j] > arr[j + 1]:
```

要素を入れ替える

```
arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

バブルソートの全体構造 (Python実装)

```
def bubble_sort(arr):  
    n = len(arr)
```

全体のパスを繰り返す
(確定部分を増やす動き)

```
    for i in range(n):
```

```
        for j in range(0, n - i - 1):
```

```
            if arr[j] > arr[j + 1]:
```

```
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

隣同士の比較と交換

最後の i 個はすでに
確定しているので無視する

```
    return arr
```

プログラムの実行例

```
data = [64, 34, 25, 12, 22, 11, 90]
```

Input



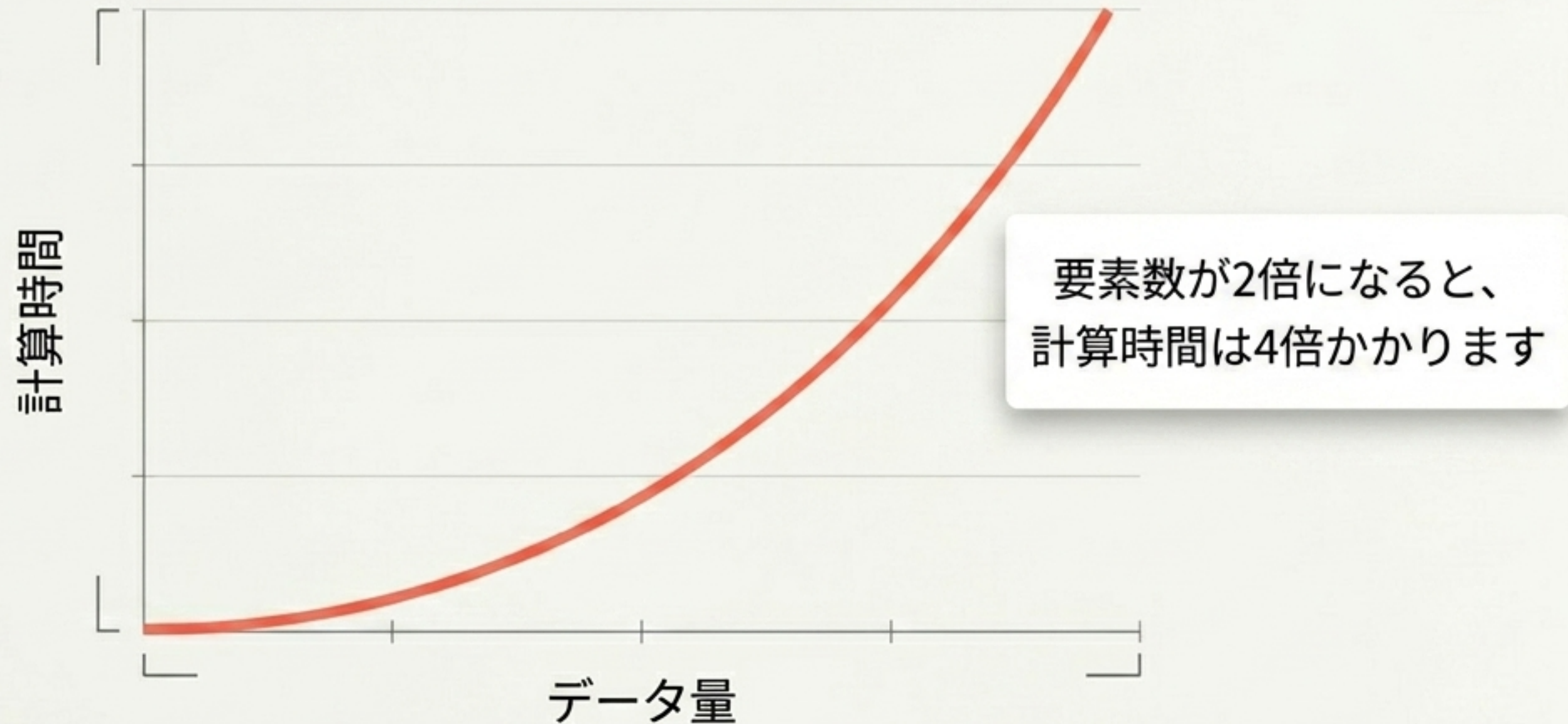
bubble_sort(data)

Output



```
sorted_data = bubble_sort(data)  
print(f"ソート後: {sorted_data}")
```

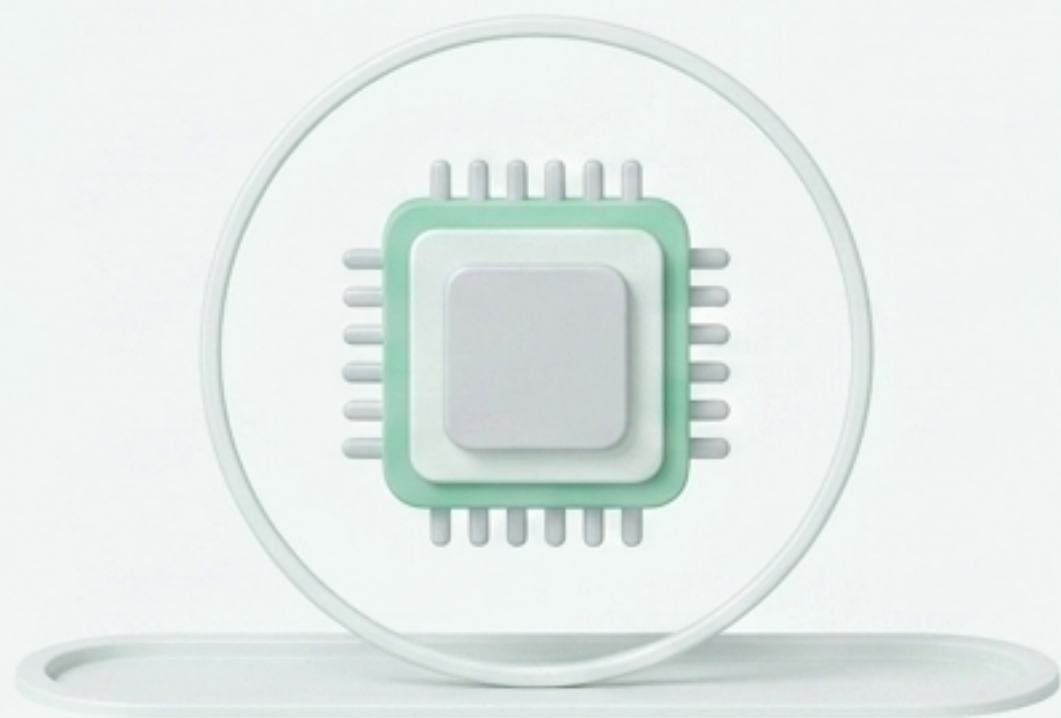
実行速度の代償：時間計算量 $O(n^2)$



平均・最悪の場合の時間計算量は共に $O(n^2)$ となります。

バブルソートの長所：省スペースと安定性

空間計算量 $O(1)$



追加のメモリをほとんど使わないため、極めて省スペースです。元のリスト内で直接要素を入れ替えるだけで完了します。

安定なソート

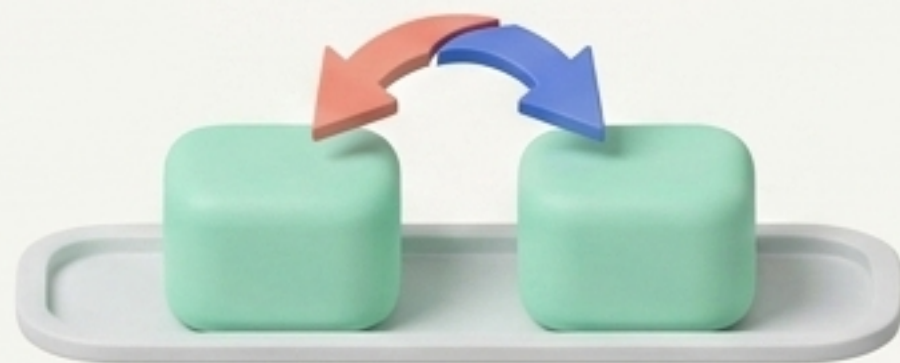


同じ値の要素が複数ある場合、その前後関係は維持されます。

バブルソートの適性評価

使用ケース	スコア	理由
教育用		シンプルで直感的なため、アルゴリズムの基礎やループ処理の学習に最適。
実用・大量データ		大量データを扱う実用的な場面では、 $O(n^2)$ の計算時間がボトルネックとなり、あまり使われません。

まとめ：バブルソートの3つの要点



隣同士を比較・交換

リストの端から隣り合う要素を比較し、条件を満たせば入れ替えるという単純なルールの繰り返し。



シンプルなコードで実装可能

複雑な関数や追加のメモリを必要とせず、短いコードで完結する。



学習には最適だが大規模データには不向き

アルゴリズムの理解は非常に優れているが、計算時間が $O(n^2)$ のため、実務のビッグデータ処理には適さない。