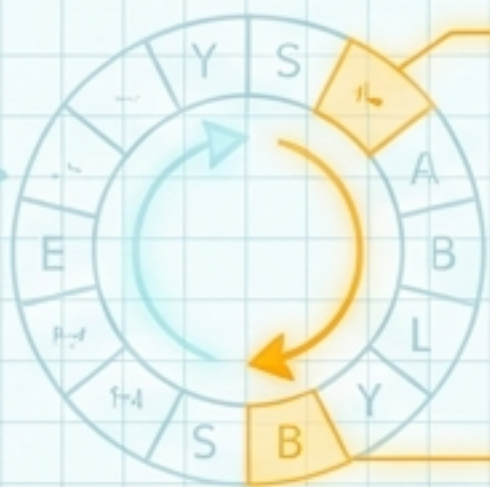


[S, Y, M, B, O, L, []] → [S, S, Y, M, B, O, L,] > [S, Y, M, B, O, L, L, []]

Block-sorting アルゴリズムの深淵

データ圧縮処理におけるループ境界とソートの安定性の考察
(設問3(2)・設問4 解説)



```
// Londar  
block = [ , * B, L ]  
[ S, Y, M, B, O, L ];
```

```
for (int i=0; i<N; i++) {  
    rotate(block);  
    sort(block);  
}
```

STABLE_SORT_IMPLEMENTATION

```
// Loop boundary & Stability Check  
int boundary = find(EOS);
```

[S, Y, M, B, O, L] (EOS)

[S, Y, M, B, O, L]

...

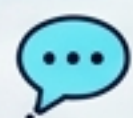
本資料で解き明かす2つの技術的「謎」

単なる解答の暗記ではなく、コードの意図とデータ構造の不変条件を論理的に導き出します。

Mystery 1: ループの境界条件 (設問3-2)

文字列復元ループは「何回」回るのか？
コードの構造から実行回数を論理的に導き出す。

```
while ( ? ) {  
    // コードの論理  
}
```



LOGICAL
DEDUCTION

LOOP COUNT: N

Mystery 2: ソートアルゴリズムの罠 (設問4)

なぜ「クイックソート」を使うとデータが
破壊されるのか？原因と対策を紐解く。



INITIAL ARRAY



ROOT CAUSE
ANALYSIS

DATA DESTRUCTION:
STABLE SORT NEEDED

ANALYTICAL SYNTHESIS

Mystery 1: ターゲットとなるコードブロックの特定

着目すべきは、復元文字列(OutputString)を組み立てるこのブロックです。

```
function decode(BlockSortString, Line)
  for( i を 1から BlockSortString の長さまで 1ずつ増やす )
    DecodeArray[1][i] ← BlockSortString の i 文字目
    DecodeArray[2][i] ← i
```

```
  endfor
```

```
  sort2(DecodeArray)
```

OutputString を空文字列に初期化する

OutputString の末尾に エ に格納されている 1文字を追加する

n ← オ

```
  while( カ )
```

(α) OutputString の末尾に DecodeArray[1][n]に格納されている 1文字を追加する
 n ← DecodeArray[2][n]

```
  endwhile
```

```
endfunction
```

この処理が『何回』実行されるかを導き出す。

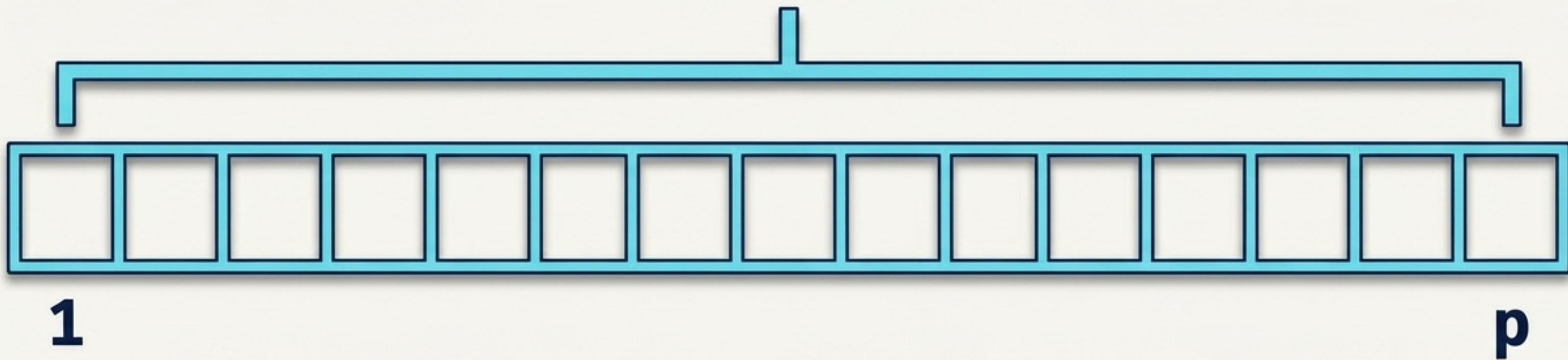


完成形から逆算してループの構造を捉える

ゴールとなる文字列の全体像を視覚化する

変数 `BlockSortString` の長さが p であるとき、最終的に完成させたい `OutputString` (復元された文字列) の長さも当然 p となります。

Total Length = p



ループに突入する「直前」の状態変化に着目する

事前処理がもたらす状態変化

コードを順に追うと、while ループに入る前に、既に OutputString に対して文字を1つ追加する初期処理が行われています。

OutputString を空文字列に初期化する

OutputString の末尾に [エ] に格納されている 1文字を追加する

```
while( カ )
```

```
    OutputString の末尾に DecodeArray
```

```
    n ← DecodeArray[2][n]
```

```
endwhile
```

Total Length = p

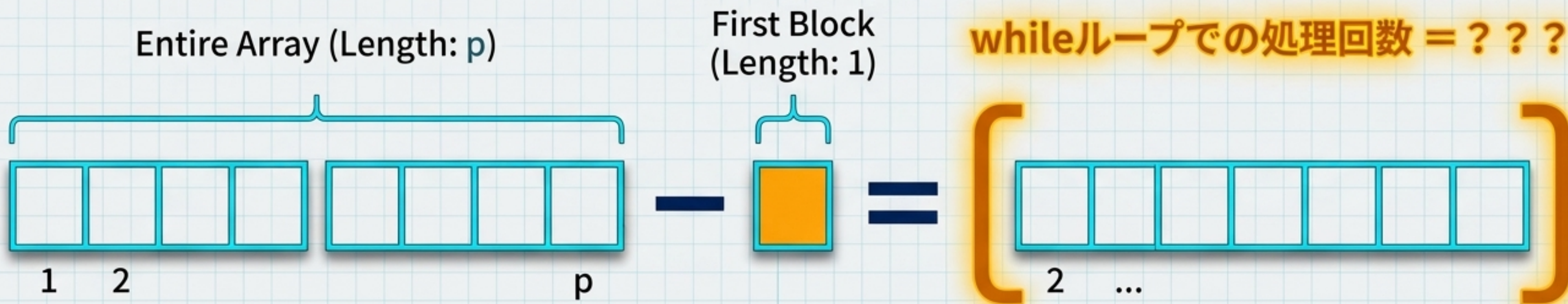


下線(α) が担うべき「残りのタスク」を算出する

全体から初期状態を引き算する

下線(α) は、残りのスロットを1文字ずつ埋めていく処理です。
以下の思考プロセスから、実行回数が自然と導き出されます。

1. 完成させたい全体の文字数は？
2. ループに入る前に、既に何文字追加されているか？
3. したがって、ループ内で処理すべき残りの文字数は？



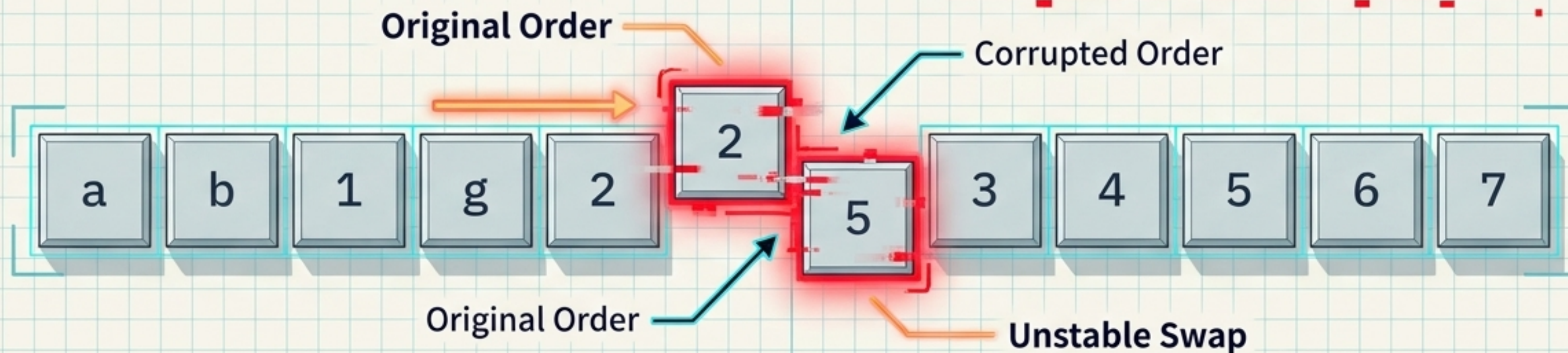
Mystery 2: 不安定なソートアルゴリズムが引き起こすバグ

クイックソートの性質がデータ復元を破壊するメカニズム

設問4では、関数 `sort2` 内部で「クイックソート（不安定なソート）」を使用した場合、復元結果が入力文字列と一致しなくなる問題が発生しました。

なぜ「ソートアルゴリズムの性質」がデータ復元を破壊するのか？

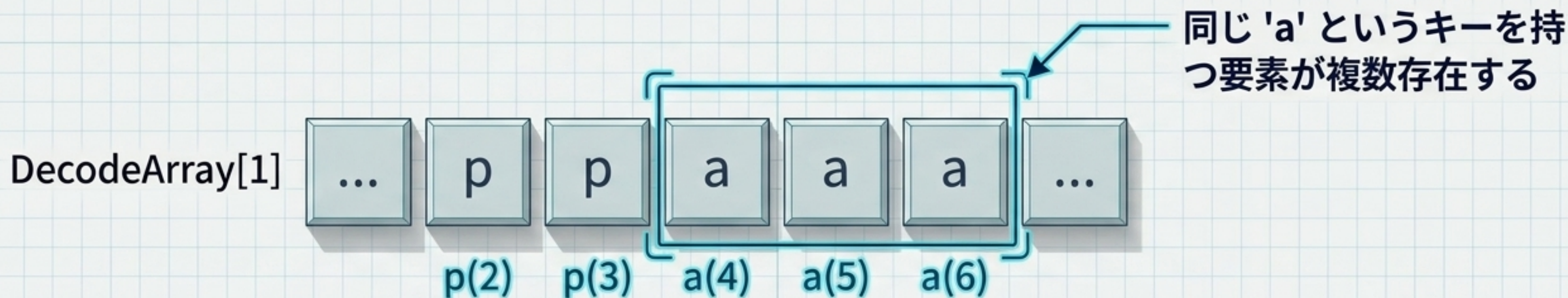
Quick Sort (**Unstable**)



Block-sorting の特性：大量に発生する「同値」のキー

変換後データには同じ文字が多く続く傾向がある

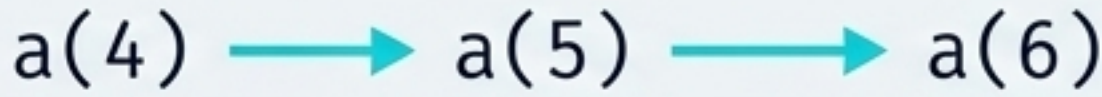
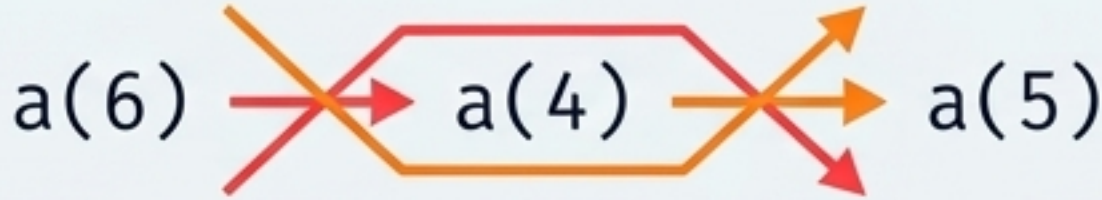
Block-sorting の目的は圧縮率の向上であり、変換結果には同じ文字が多く続く傾向があります。文字をキーとしてソートする際、大量の「同値（同じ文字）」を扱うことになります。



安定なソート vs 不安定なソートの違い

同値の要素を並べ替えた際の振る舞いの差異

同値の要素を並べ替えた際、元々格納されていた相対的な順序（添字の順序）がどうなるかに決定的な違いがあります。

ソートの性質	同値キーの振る舞い（例: 複数の“a”）	元の順序（添字）の保持
安定なソート (Stable Sort)		保持される（元の順番通り）✓
不安定なソート (Unstable Sort)		破壊される可能性がある ✗

順序の破壊が引き起こす復元プロセスの破綻

復元手順の大前提が崩壊する

表1の復元手順2には「同じ文字の場合は添字の順に並べる」という厳密な仕様が存在します。
不安定なソートではこの大前提が崩壊します。

表1 Block-sorting の復元手順（抜粋）

手順	処理	内容
2	文字をソートする。同じ文字の場合は添字の順に並べる。	“y(1),p(2),p(3),a(4),a(5),a(6)” → “a(4),a(5),a(6),p(2),p(3),y(1)”
3	手順2でソートした文字を次の手順で並べる。 ・ 変換結果の行番号“4”から、ソート後の文字列 “a(4),a(5),a(6),p(2),p(3),y(1)”の4番目の要素 “p(2)”を取り出して並べる。	→ “p(2)” → “p(2),a(5)” → “p(2),a(5),p(3)” ...

不安定なソートの結果、a(6) が a(4) より先に来る場合：
元の正しい配列： [p(2), p(3), a(4), a(5), a(6), y(1)]
不安定なソート後： [p(2), p(3), a(6), a(5), a(4), y(1)]

復元プロセスが崩壊

正しい復元：p - a - p - a - y - a

誤った復元：p - y - ? → **p - y - ? - ? - ? - ?**

Solution A：ソートアルゴリズム自体を見直す場合

アルゴリズム自体の性質を変更するアプローチ

問題の根本原因は、「ソートアルゴリズムが同値要素の順序を保持しなかったこと」にあります。

~~現在の実装：
不安定なソート
(クイックソート)~~



改善案：
○○○ソートアルゴリズムを
使用する

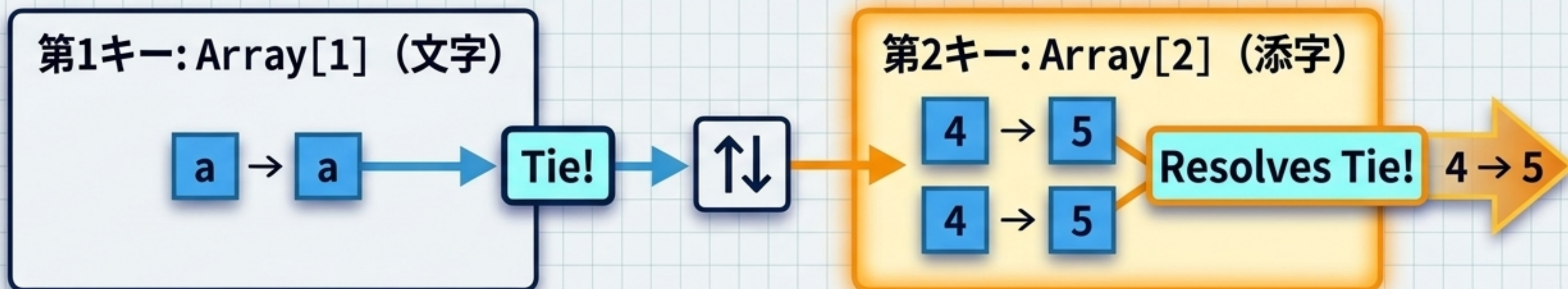
思考のヒント

- クイックソートは「不安定なソート」であると問題文に明記されています。
- では、その逆の性質、つまり「常に元の順序を保持する性質」を持つソートアルゴリズムを何と呼ぶでしょうか？

Solution B: ソートキーを見直す場合

不安定なアルゴリズムのまま順序を保証するアプローチ

不安定なアルゴリズムのまま順序を保証するには、同値要素同士を比較する際に「一意になる別のデータ」を第2のキーとして用いる必要があります。



思考のヒント

- 表2の DecodeArray[2][n] には元の順番を示す「添字」が格納されています。
- 文字(Array[1])が同じなら、この添字を次の判断基準に加えれば順序はコントロール可能です。

Synthesis: 本問が求めていたアルゴリズム的思考

コードリーディングとデータ構造の深い理解

本問を解くために必要だったのは、表面的なコードの追跡だけでなく、アルゴリズムの「不変条件」を見抜く設計者の視点です。

境界値の分析 (Q3)



全体の長さ（ゴール）と初期化処理（スタート）を把握し、ループが担うべき「差分」を論理的に導き出す力。

ソート特性とデータ構造の整合性 (Q4)



アルゴリズム（安定/不安定）とデータ構造（重複キー）の相性を理解し、仕様を満たす代替手段を設計する力。