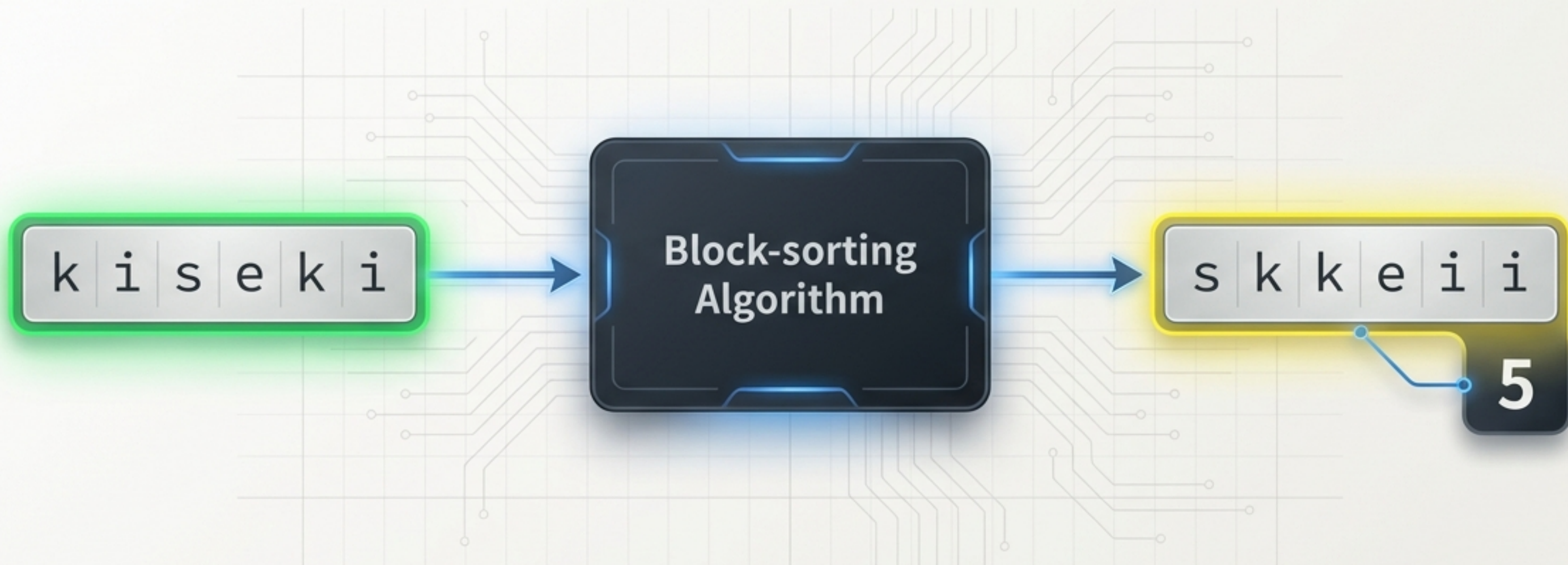
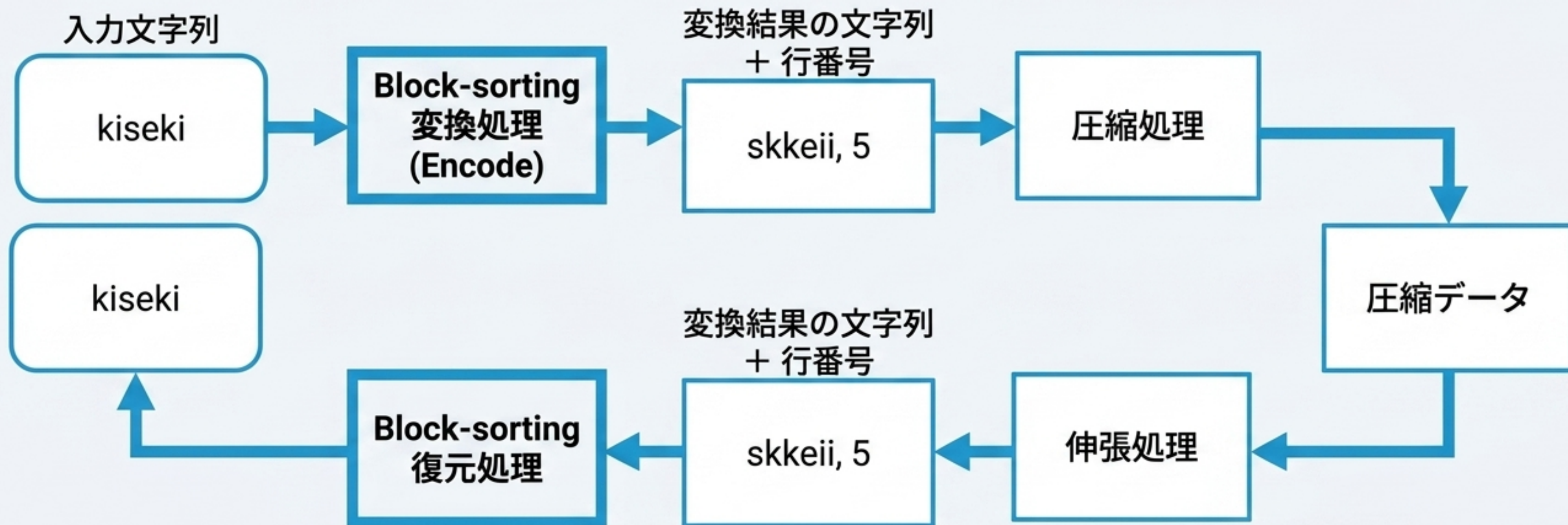


Block-sortingの解剖とPython実装

平成27年春期 応用情報技術者試験（午後 問3）から学ぶ、可逆変換アルゴリズムの美しさ



The Context: データ圧縮の前処理



Takeaway: 単なる暗号化ではなく、情報を1ビットも失わずに「圧縮しやすい形」へ姿を変える魔法のステップ。

The Transformation (Encode): ブロックの生成



①	k	i	s	e	k	i
②	i	s	e	k	i	k
③	s	e	k	i	k	i
④	e	k	i	k	i	s
⑤	k	i	k	i	s	e
⑥	i	k	i	s	e	k

手順1: 巡回シフト

文字列長と同じ回数、1文字左巡回シフトを回数、1文字左巡回シフトを繰り返して正方形のブロック（2次元配列）を生成する。

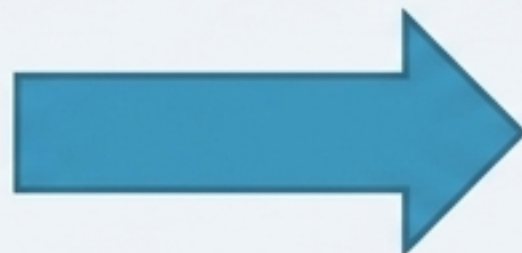
この操作により、文字列のすべての順列パターンが網羅される。

The Transformation (Encode): ソートと抽出

巡回シフト後のブロック

k	i	s	e	k	i
i	s	e	k	i	k
s	e	k	i	k	i
e	k	i	k	i	s
k	i	k	i	s	e
i	k	i	s	e	k

行単位で
辞書順にソート



ソート後のブロック

1	e	k	i	k	i	s
2	i	k	i	s	e	k
3	i	s	e	k	i	k
4	k	i	k	i	s	e
5	k	i	s	e	k	i
6	s	e	k	i	k	i

元の入力文字列
(行番号 5)

抽出される変換結果

設問1 解答: 「skkeii', 5」

抽出のルール: ソート後、一番右の文字を上から抽出。元の文字列が存在する「行番号」を記録。

Code Mapping: 変換処理の実装

試験の擬似言語 (図3)

```
for( i を ア から イ まで 1 ずつ増やす )  
    EncodeArray[i] ← rString  
    rString ← rotation(rString)  
endfor  
...  
for( k を ア から イ まで 1 ずつ増やす )  
    if( ウ )  
        Line ← k
```

Python実装

```
length = len(input_string)  
for i in range(1, length + 1):  
    encode_array.append(r_string)  
    r_string = r_string[1:] + r_string[0]  
...  
for k in range(length):  
    if encode_array[k] == input_string:  
        line = k + 1
```

設問2 解答: ア=1, イ=InputStringの長さ, ウ=InputStringとEncodeArray[k]が等しい

The Restoration (Decode): 辞書の構築

添字の付与 (インデックスの記録)

s	k	k	e	i	i
s(1)	k(2)	k(3)	e(4)	i(5)	i(6)



文字をキーにして辞書順ソート

ソートされた2次元配列 (DecodeArray)

DecodeArray[1] (文字)	e	i	i	k	k	s
DecodeArray[2] (元の添字)	4	5	6	2	3	1

核心となるメカニズム

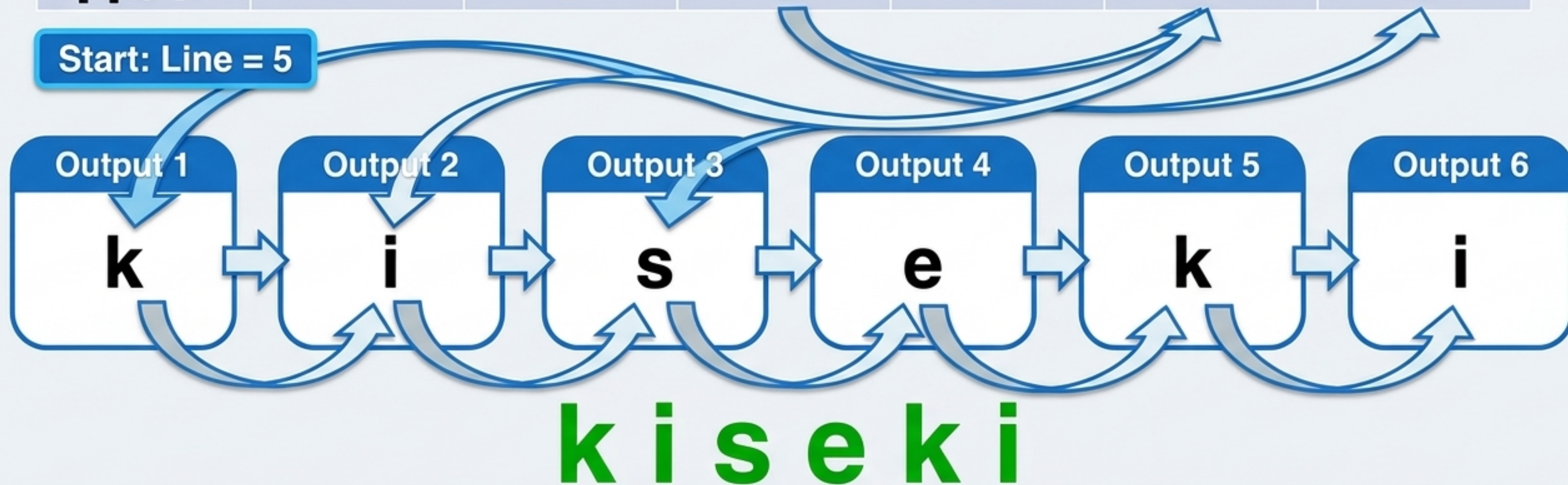
このソートされた2次元配列が、元の文字列を紡ぎ出すための「ポインタ辞書」として機能する。

The Restoration (Decode): ポインタが紡ぐ魔法の鎖

DecodeArray

Index:	1	2	3	4	5	6
[1] 文字:	e	i	i	k	k	s
[2] 添字:	4	5	6	2	3	1

Start: Line = 5



添字（インデックス）が「次に見るべき要素の場所」を示すポインタとして機能し、元の順序を完全に復元する。

Code Mapping: 復元処理の実装

試験の擬似言語 (図4)

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

n ← ^(DecodeArray[2][Line]) [才]

while([力])

 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

Python実装

```
output_string += decode_array[line - 1][0]
n = decode_array[line - 1][1]
# (OutputStringの長さがBlockSortStringの長さより小さい)
while len(output_string) < length:
    output_string += decode_array[n - 1][0]
    n = decode_array[n - 1][1]
```

設問 3 (1) 解答：エ=DecodeArray[1][Line], 才=DecodeArray[2][Line],
力=OutputStringの長さがBlockSortStringの長さより小さい

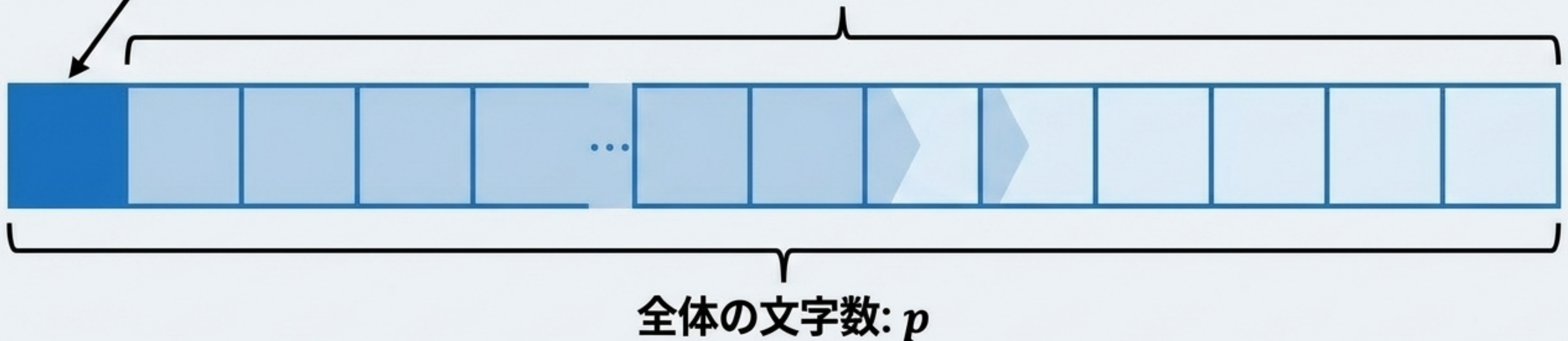
アルゴリズムの効率: ループ実行回数

1文字目

ループ突入前（空欄エの処理）に格納される

残りの $(p - 1)$ 文字

whileループ内（下線aの処理）で1文字ずつ追加される

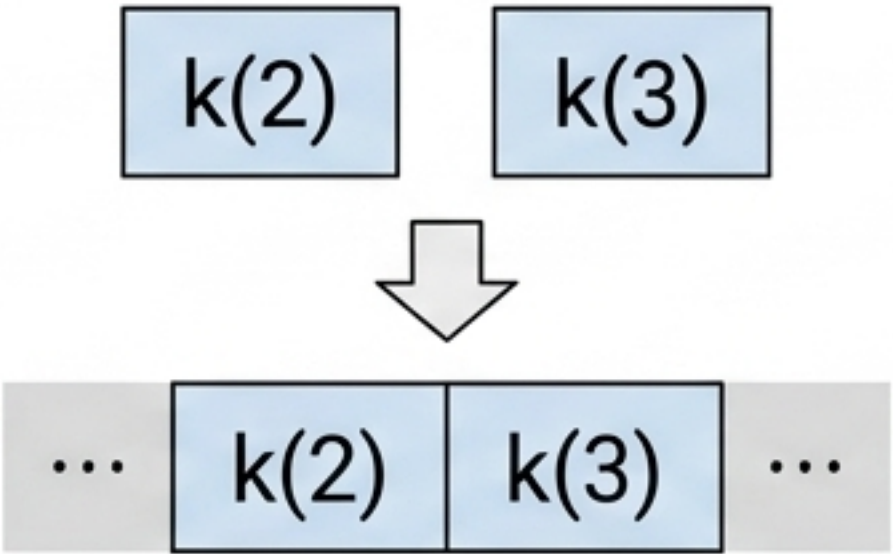
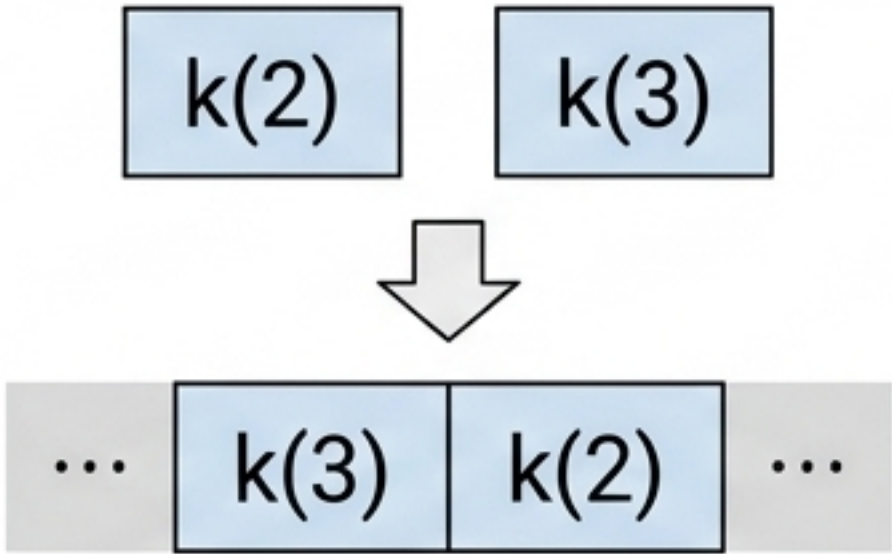


設問 3 (2) 解答: $p - 1$

解説: 復元すべき全体の文字数 p のうち、最初の1文字はループに入る前に処理が完了する。
したがって、ループ内の追加処理は残りの $(p - 1)$ 回だけ実行される。

The Gotcha: 「不安定なソート」の罠

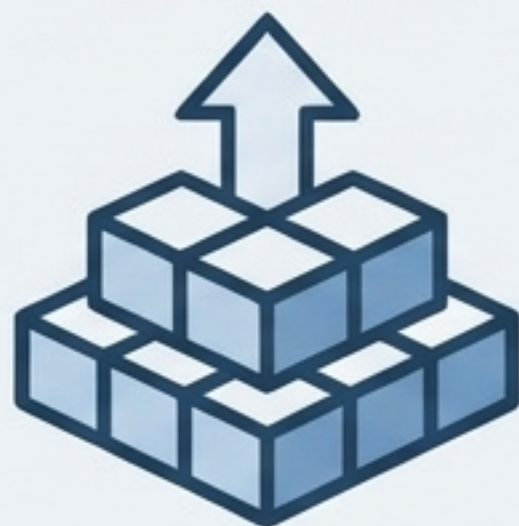
設問4: クイックソートを使用すると復元結果が一致しなくなった理由

安定なソート (Stable Sort)	不安定なソート (Unstable Sort - e.g. Quick Sort)
 元の添字の順序が保持される。 ポインタの鎖は正しく繋がる。	 同じ文字同士の順序が逆転する可能性がある。ポインタが間違ったインデックスを指し、復元の鎖が完全に崩壊する。

注: この理由により、復元関数 decode でクイックソートを使用すると、ポインタが間違った要素を指すことになり、復元が失敗する。

Architecture Fix: 安定性を担保する2つのアプローチ

解決策 A: アルゴリズムの見直し



常にソート結果が同じになる「安定なソートアルゴリズム」（マージソートやバブルソートなど）を使用する。

設問4 解答例: 安定なソートアルゴリズムを使用する

解決策 B: ソートキーの見直し

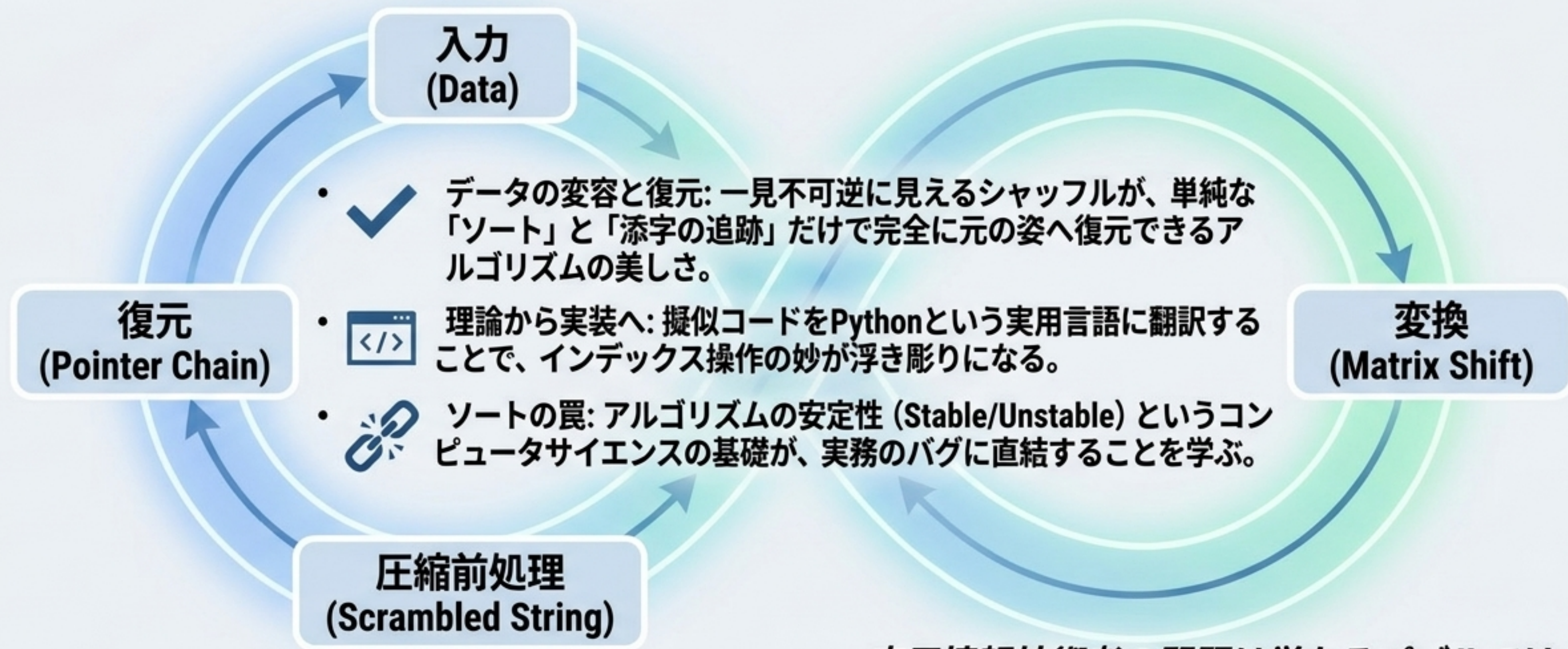
Key 1 (文字)	Key 2 (元の添字)
k	2
k	3

⇒ 文字（第1キー）が同じ場合は、元の添字（第2キー）で並べ替える複合キーを採用する。これにより、不安定なソートでも順序が強制される。

設問4 解答例: Array[2]の要素を第2キーとしてソートキーに含める

教訓: アルゴリズムの特性 (安定性) への理解が、致命的なバグを防ぐアーキテクチャ設計に直結する。

Takeaways: 思考のキャンバス



応用情報技術者の問題は単なるパズルではない。
実用的なソフトウェア設計の基礎が詰まった、
洗練された思考のキャンバスである。

End of Presentation.