

```
function decode(block) {
  int n = block.length;
  for (int i = 0; i < n; i++) {
    int n.*rr = [+ *];
    return arr>:arrayd1.set[];
  }
}
```

2D array

[0, 0]	[0, 1]	[0, 2]	[0, 3]
[0, 0]	[0, 1]	[0, 2]	[0, 3]

Block-sorting 復元処理の解剖学

設問3：DecodeArrayはプログラム内でどのように振る舞うのか？

```
int n = block.length;
for (int i = 0; i < n; i++) {
  x = vcrs.d[i] + (i + 1) * n;
  (1.set[2]);
}
```



※本スライドに設問の直接的な解答（空欄の答え）は記載されていません。
アルゴリズムの「動き」を視覚的に理解し、自力で解答を導くための解説ドキュメントです。

今回のターゲット：復元処理（decode関数）

本デッキでは、変換結果の文字列から元の文字列を取り戻す関数 **decode** の内部メカニズムを視覚化します。

```
decode(  
    [uint8_t] src, write:  
    [uint8_t] write = (BlockSortString).String;  
    try {  
        int aity = String(0);  
        int tnatdrne = -exnom:RackBackString[11];  
        int test = aneat;  

```

```
    if (fafeatSure=100p) {
```

```
        isan  
        40E  
        string = tsr.111;  

```

```
        return atetesrt_String[0];  

```

```
    } in:snw(funatdat) {  

```

```
        return 0  

```

```
        updrable(Dettwirst(geaa, syppaaa);  
        toOermat=ex(oranaki_Arument(3%u);  
    }  
}
```

入力文字列

変換処理

圧縮

伸張

復元処理

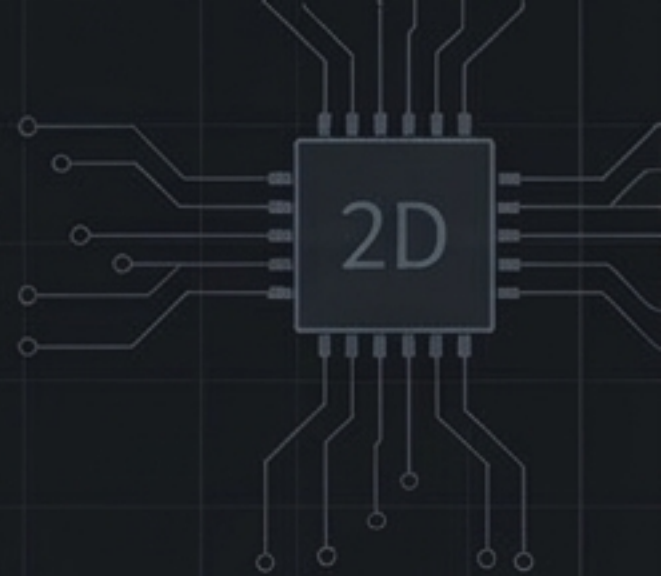
【検証する初期条件】

- 入力文字列
(**BlockSortString**):
yppaaa
- 行番号 (**Line**): **4**



ミッション: このデータから、元の文字列 **papaya** をどうやって再構築するのか？

復元の鍵を握るデータ構造：DecodeArray



[1] [2] [3] [4] [5] [6]

DecodeArray[1]



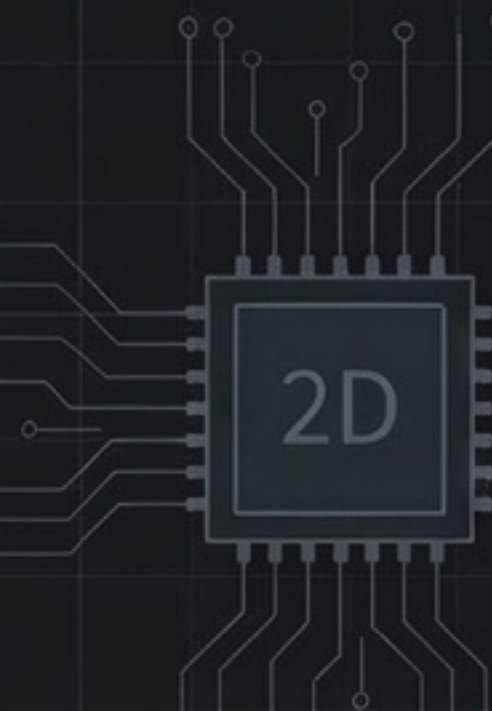
DecodeArray[2]



DecodeArray[1]:
文字そのものを格納
する「データ領域」

DecodeArray[2]:
その文字が元々何番目
だったかを記憶する
「元の添字領域」

ルール: 添字は1から始まります (文字列の長さ = 6)。



```
function decode(BlockSortString, Line)
```

Step 1: 配列への初期データのセット

```
DecodeArray[1][i] ← BlockSortString の i 文字目  
DecodeArray[2][i] ← i  
endfor  
// ... subsequent code omitted ...  
}
```

```
function decode(BlockSortString, Line)
```

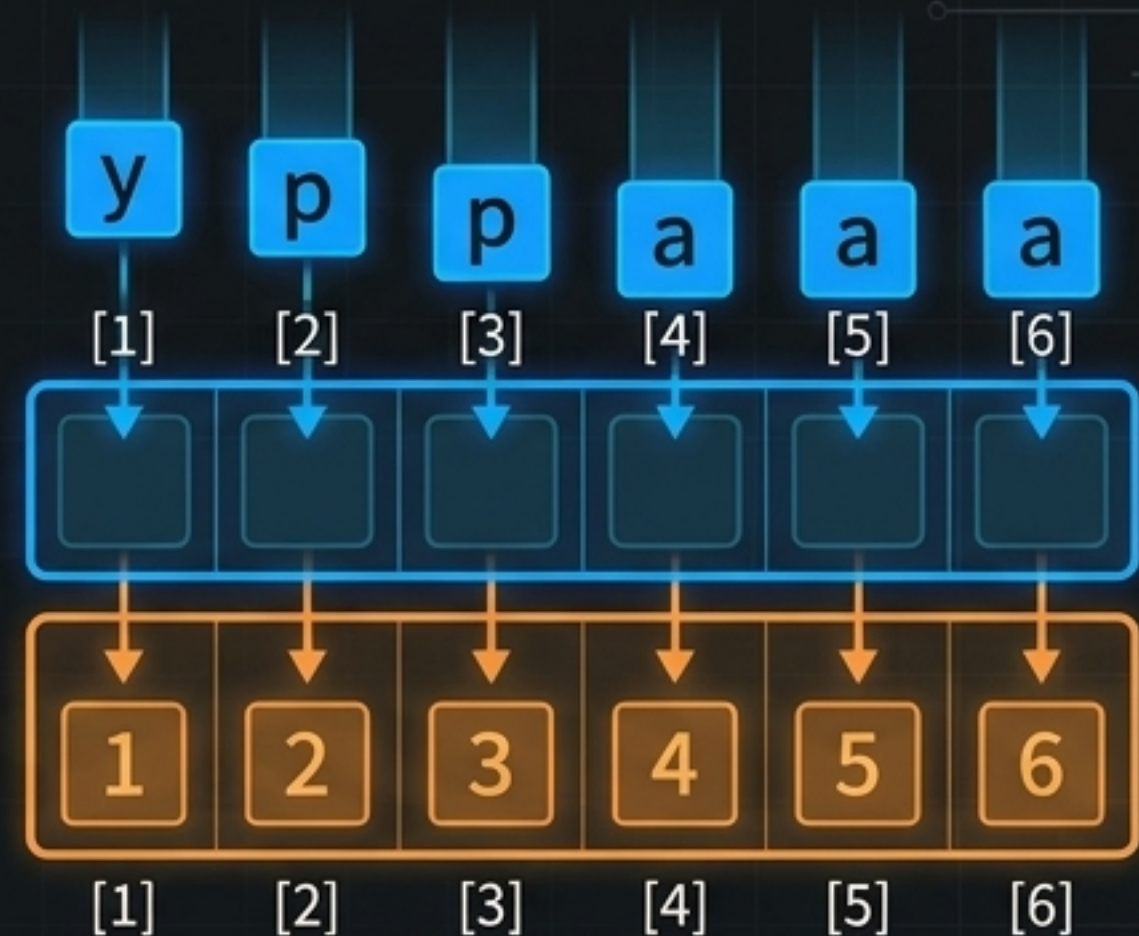
```
  for( i を 1 から BlockSortString の長さまで 1 ずつ増やす )  
    DecodeArray[1][i] ← BlockSortString の i 文字目  
    DecodeArray[2][i] ← i  
  endfor
```

```
  // ... subsequent code omitted ...
```

```
}
```

DecodeArray[1]

DecodeArray[2]



最初のループ処理により、指定された文字列 yppaaa と、連番 1~6 が配列に格納されます。これが本解説における初期状態です。

【確認】指定された初期状態の完成



[1] [2] [3] [4] [5] [6]

[y]	[p]	[p]	[a]	[a]	[a]
[1]	[2]	[3]	[4]	[5]	[6]

設問で提示された「DecodeArray[1]がyppaaaで
DecodeArray[2]が123456」という状態がメモリ上に構築されました。ここからが復元アルゴリズムの真骨頂です。

Step 2: ソートの実行 (アルゴリズムの転換点)

sort2(DecodeArray)

```
stepx: func DecodeArray: {  
  DecodeArray = macrosortine.sort2(DecodeArray[1]);  
  
  if (sort2(DecodeArray)) {  
    if (lrenosa == 3) {  
      tDecodeArray;  
    }  
  }  
}
```



```
// ソートの並べ変替えます  
const tsort2ns(DecodeArray, dent {  
  if (ap != k1a) {  
    sort2(DecodeArray);  
  }  
  return tki#1;  
}
```



解説: 関数 sort2 は、上段の青い文字 (DecodeArray[1]) をキーにして辞書順 (a → p → y) に列ごと並べ替えます。

最重要ポイント: 上下のブロックは「鎖」で繋がれています。文字が移動すると、下段に紐づいていた「元の添字 (オレンジの数字)」もセットで一緒に移動します。



```
if (am.DecodeArray[] & !command) else {  
  sort2(DecodeArray);  
}  
  
void (DecodeArray) {  
  moreoment.vsId(DecodeArray);  
}
```

ソート前後の状態比較

Before Sort

DecodeArray[1]
(The Characters)

DecodeArray[2]
(The Original Indices)

[1]	[2]	[3]	[4]	[5]	[6]
[y]	[p]	[p]	[a]	[a]	[a]
[1]	[2]	[3]	[4]	[5]	[6]

After Stable Sort

DecodeArray[1]
(The Characters)

DecodeArray[2]
(The Original Indices)

[1]	[2]	[3]	[4]	[5]	[6]
[a]	[a]	[a]	[p]	[p]	[y]
[4]	[5]	[6]	[2]	[3]	[1]



安定なソートにより、同じ文字（例：'a'）は元の順番（4, 5, 6）を保ちます。配列の順番は入れ替わりましたが、上下の「文字」と「元の添字」のペア関係は崩れていません。

パラダイムシフト：オレンジの数字は何者か？

解説

バラバラになった**オレンジの数字**。
これは単なる元の添字ではなく、
文字列を復元するための
「次に見に行くべき列を示すポインタ
(地図)」として機能します。

鉄則：

「いま取り出した列の下段にある数字」
=
「次にジャンプすべき列のインデックス」

```
DecodeArray[1]  
(The Characters)
```

```
DecodeArray[2]  
(The Original Indices)
```

[1]	[2]	[3]	[4]	[5]	[6]
a	a	a	p	p	y
4	5	6	2	3	1



```
void (DecodeArray) {  
    movement.vslid(DecodeArray);  
}
```

Step 3: 復元のスタート (最初の1文字)

Code Reference: 空欄 [エ] 付近の処理

	[1]	[2]	[3]	[4]	[5]	[6]
DecodeArray[1] (The Characters)	a	a	a	p	p	y
DecodeArray[2] (The Original Indices)	4	5	6	2	3	1

Line (4)

p

OutputString

Output: p

1. 復元の起点は、変換時に記録された
行番号 **Line** (今回は **4**) です。

2. ソート後の配列の **4番目の列** を見に行
きます。上段には '**p**' があります。

3. これが復元文字列の**最初の1文字目**として
OutputString に追加されます。

```
//  
vbar *OutputString =  
createheneutoq'Original' f][3]41)  
{  
excharacter = DecodeArray[11Character);  
string name = $$.aceinox5vmeoe();  
// sorted {c ad end'DL:100291',J.28x18:100.  
return s:1P000: 0W15378:183  
);
```

```
//  
c0es fnog chectern t = Outend:11247104+00[] 144: 4  
var arc.andString)  
{  
co50n] s:1ra0e0e0ts < A0t0400E0E0N);  
return OutputStringy jon), 40)va:0m0(01String00;  
});
```


Step 5: whileループ内でのポインタの連鎖 (Chain)

Code Reference: 下線 (α) と $n \leftarrow \text{DecodeArray}[2][n]$



反復1: $n=2$ にジャンプ → 上段の文字 'a' を追加 → 下段を読み $n=5$ へ更新
反復2: $n=5$ にジャンプ → 上段の文字 'p' を追加 → 下段を読み $n=3$ へ更新

OutputString

OutputString: pap

Step 6: 復元の完了

Code Reference: 下線 (α) と $n \leftarrow \text{DecodeArray}[2][n]$



反復3: $n=3 \rightarrow$ 文字 'a' を追加 $\rightarrow$ 次は $n=6$

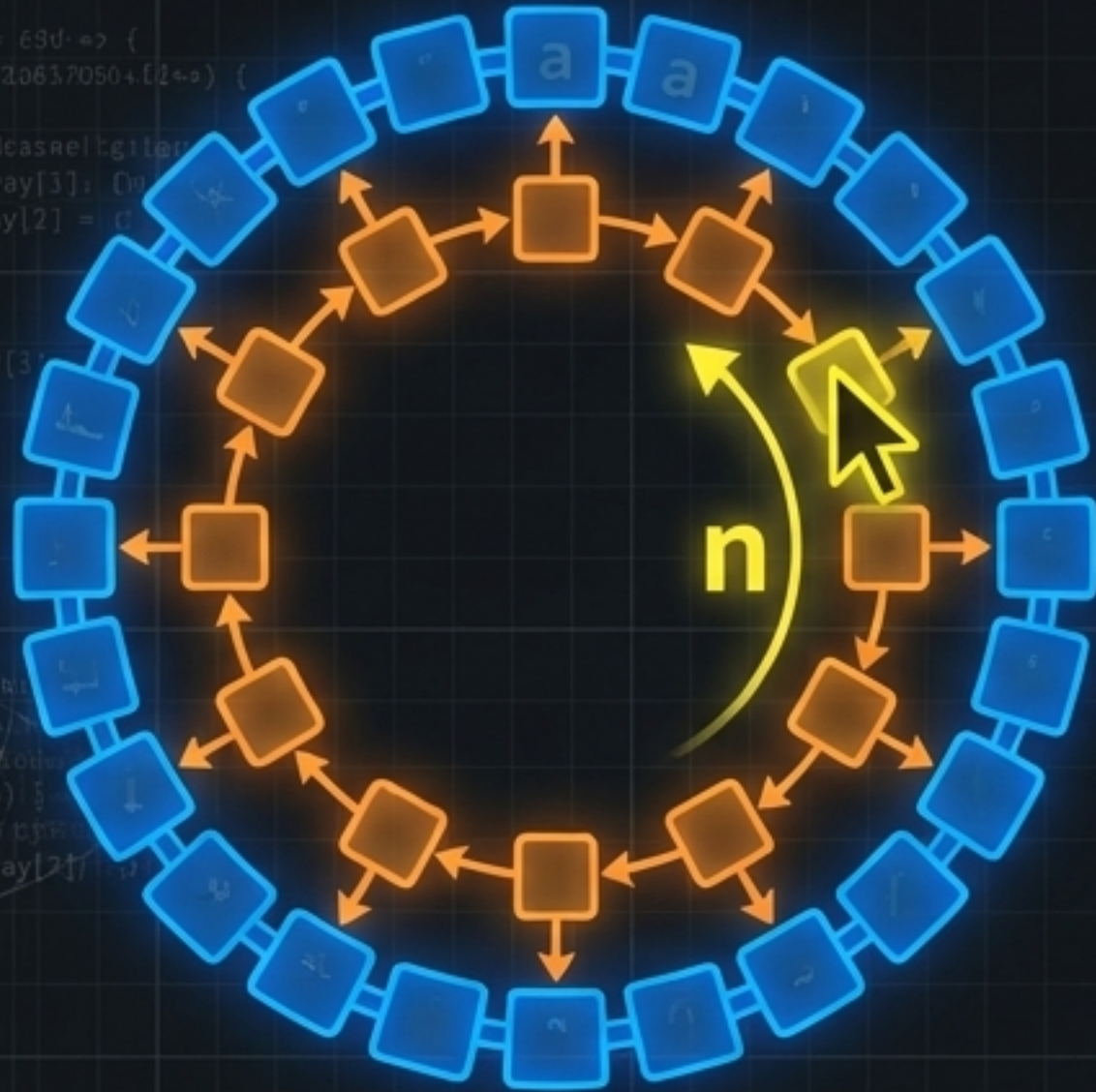
反復4: $n=6 \rightarrow$ 文字 'y' を追加 $\rightarrow$ 次は $n=1$

反復5: $n=1 \rightarrow$ 文字 'a' を追加 $\rightarrow$ (終了条件到達)

Target Length Reached

✓ Final Output: papaya

まとめ：DecodeArrayの真の役割



- 1. **DecodeArray[1] (Blue):**
復元される文字列を構成する
「データパーツ」
- 2. **DecodeArray[2] (Orange):**
パーツを正しい順序で繋ぐための
「次への道しるべ (ポインタ)」
- 3. **変数 n (Yellow):** 道しるべに従って
配列内を移動する「カーソル」

指定された初期値 yppaaa と 123456 は、ソートを経ることで、文字と次へのポインタがペアになった強力な復元マップへと姿を変えました。このデータの振る舞いを理解すれば、プログラムの空欄に何が入るべきか、自然と見えてくるはずです。