

# アルゴリズム読解の極意：幅優先探索の実装ロジック

令和7年春 応用情報技術者 午後問3（設問2 空欄イ・ウ・エ・オ） 攻略アプローチ

開始時点の盤面



空白マス

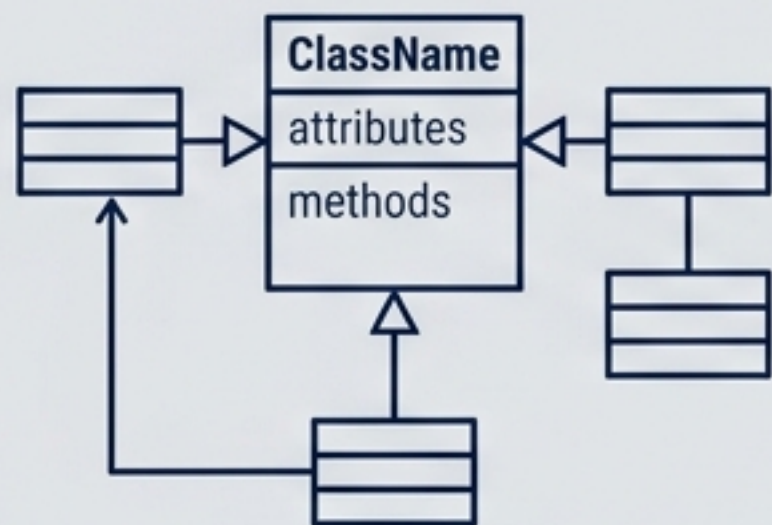
ゴールの盤面



```
11: start_state ← BoardState()
12: goal_board ← createGoal(board_size)
13: start_state, board ← createStart(board_size)
14: start_state.space ← getSpace(start_state.board)
15: direction ← {(1, 0), (0, -1), (-1, 0), (0, 1)}
16: explore_queue ← Queue()
17: explore_queue.add(start_state)
18: check_list ← List()
19: check_list.add(start_state.board)
20: while (  がfalseと等しい)
21:   state ← explore_queue.
22:   for (iを1からdirectionの要素数まで1ずつ増やす)
23:     if ((state.space[1] +  が1以上) かつ
24:         (state.space[1] +  が  以下) かつ
25:         (state.space[2] +  が1以上) かつ
26:         (state.space[2] +  が  以下))
27:       new_state ← BoardState(state)
28:       new_state.space[1] ← state.space[1] + 
29:       new_state.space[2] ← state.space[2] + 
30:       change_num ← new_state.board[new_state.space[1]][new_state.space[2]]
31:       new_state.board[new_state.space[1]][new_state.space[2]] ← board_size × board_size
32:       new_state.board[state.space[1]][state.space[2]] ← change_num
33:       if (  がtrueと等しい)
34:         printResult(new_state)
35:         return
36:       endif
37:       if (checkSameBoard(new_state.board, check_list)がfalseと等しい)
38:         explore_queue.
39:         check_list.
40:       endif
41:     endif
42:   endfor
43: endwhile
44: ゴールの盤面に至る軌の移動方法がない旨のメッセージを出力
45: return
```

# 「答え合わせ」ではなく「思考プロセスの獲得」へ

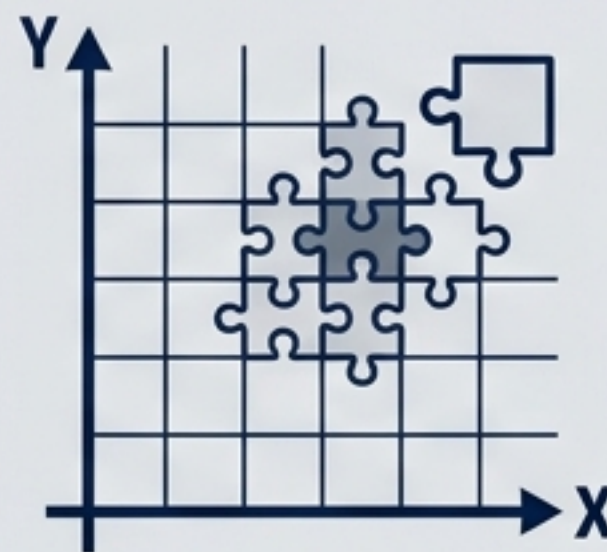
未知のソースコードに直面したとき、闇雲に選択肢を当てはめるのは危険です。このデバッグセッションでは、3つの武器を使ってコードの文脈（Context）を読み解き、論理的に答えを導き出すスキルを身につけます。



**クラス仕様の確認**  
(Check Class Specs)



**変数のライフサイクル**  
(Track State Changes)



**物理モデルとの対応**  
(Map to Reality)

# 迷子にならないための全体マップ

ターゲットとなる空欄は、すべて中央の巨大な while ループ内に集中しています。このループが「盤面を 1つ取り出し、次の一手をすべて試す」という動作を繰り返していることを意識しましょう。

`</>` 初期化ブロック (L11-L19) : 盤面とキューの準備

`</>` 探索ループ (L20-L43) : 幅優先探索のコアロジック

[イ]

[ウ]

[エ]

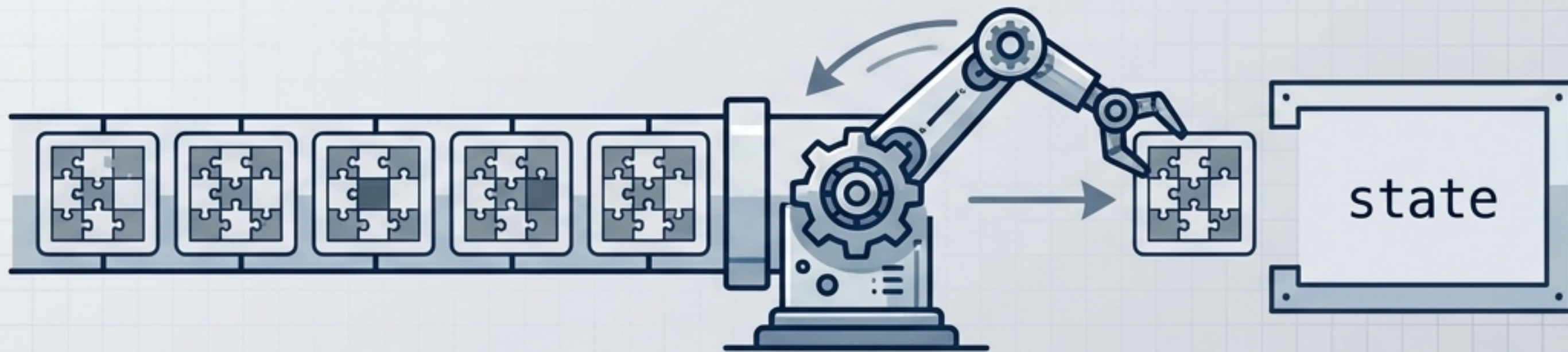
[オ]

`</>` 終了処理 (L44-L45) : 解なしの出力

# 空欄 [イ] のコンテキスト：キューからのデータ取得

```
while ( [ア] が false と等しい )  
    state <- explore_queue.[イ]
```

explore\_queue は、現在探索対象となっている盤面を保持するキューです。while ループが回るたびに、このキューから盤面を1つ取得し、state に格納して次の一手を計算します。



問うべき問い：次のループに進むためには、取得した要素はキュー内でどう扱われるべきでしょうか？

# 比較マトリックス：無限ループを防ぐメソッドはどちらか

クラス仕様には、先頭要素を取得する2つのメソッドが定義されています。幅優先探索では、一度探索したノードはキューから「消費」しなければ、同じ盤面を永遠に調べ続ける無限ループに陥ります。空欄 [イ] には、キューを正しく「消費」するメソッドが入ります。

| メソッドA: poll()                                                                                   | メソッドB: peek()                                                                                     |
|-------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| 取り出して返す<br> | 参照のみ（返す）<br> |

キュー要素が減る = ループが進行する

# 空欄 [ウ] [エ] [オ] のコンテキスト：移動のバリデーション

空白マスを「下・左・上・右」に動かそうとする際、盤面の端にいるときはその方向へ動かさせません。この if 文は、「移動後の座標が盤面の外にはみ出さないか」をチェックする安全装置（バリデーション）として機能しています。

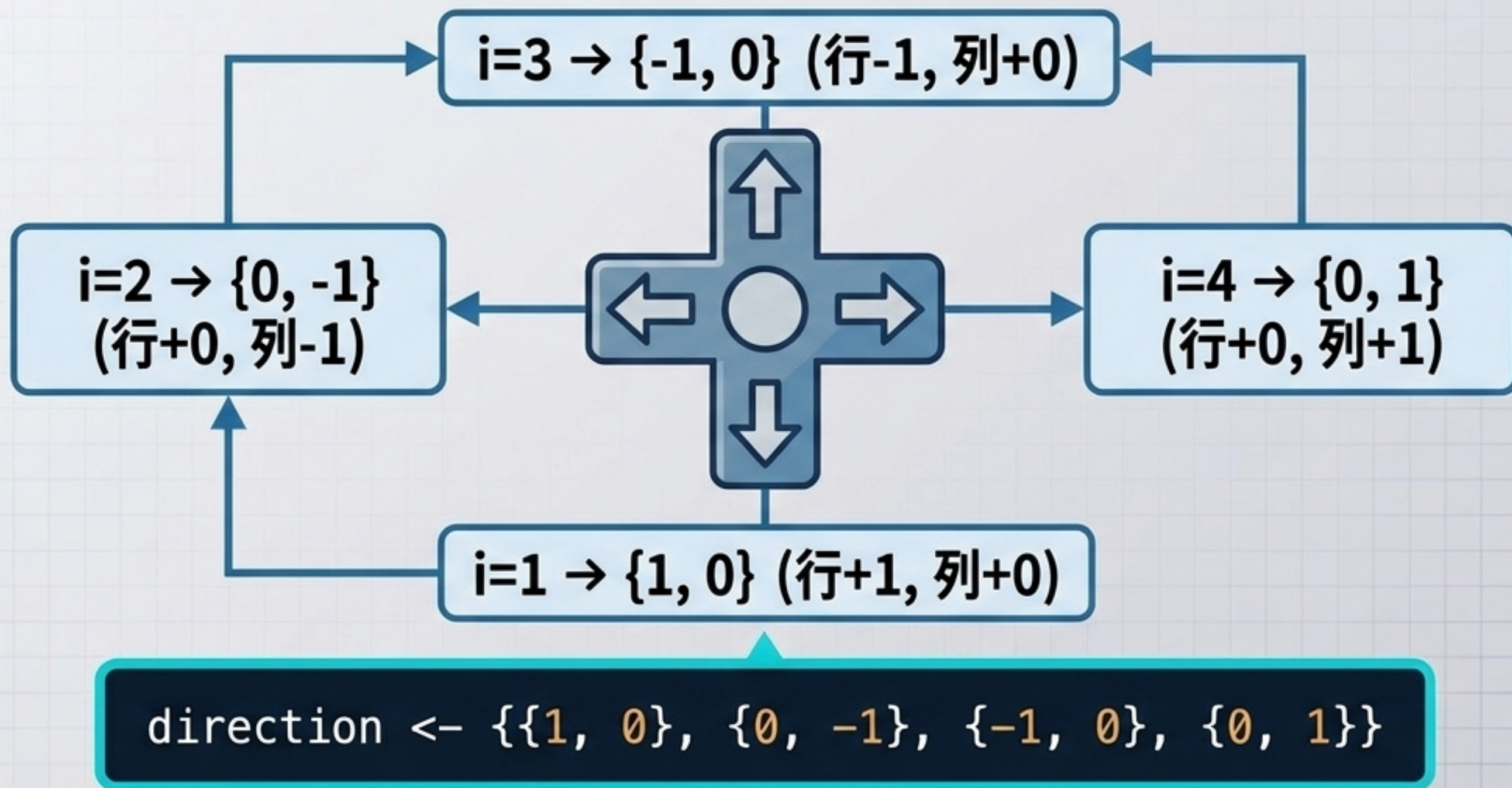
```
if ((state.space[1] + [ウ] が1以上) かつ  
    ((state.space[1] + [ウ] が [エ] 以下) かつ  
    ((state.space[2] + [オ] が1以上) かつ  
    ((state.space[2] + [オ] が [エ] 以下)))
```

行 (Y軸) の境界チェック

列 (X軸) の境界チェック

# 物理モデル変換：「方向」を配列でどう表現するか

direction 配列は、ループ変数  $i$  番目の方向に対する {行番号の増減, 列番号の増減} を保持する2次元配列です。1つ目が「行の移動量」、2つ目が「列の移動量」を表します。



# 空欄 [ウ] の構築：行（Y軸）の移動計算

現在  $i$  番目の方向をチェックしているとき、足し合わせるべき「行番号の増減値」は、`direction` 配列のどこに格納されているでしょうか？ 2次元配列の要素指定を考えます。

|   |                                                                                     |   |
|---|-------------------------------------------------------------------------------------|---|
| 1 | 2                                                                                   | 3 |
| 2 |  | 4 |
| 3 |                                                                                     | 5 |

$(\text{state.space}[\textcolor{brown}{1}] + [\textcolor{yellow}{ウ}])$

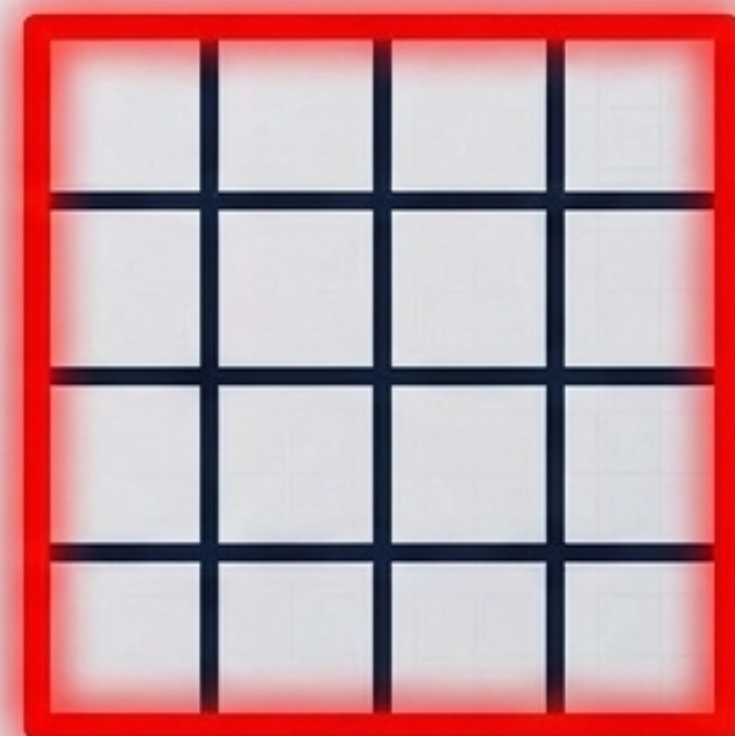
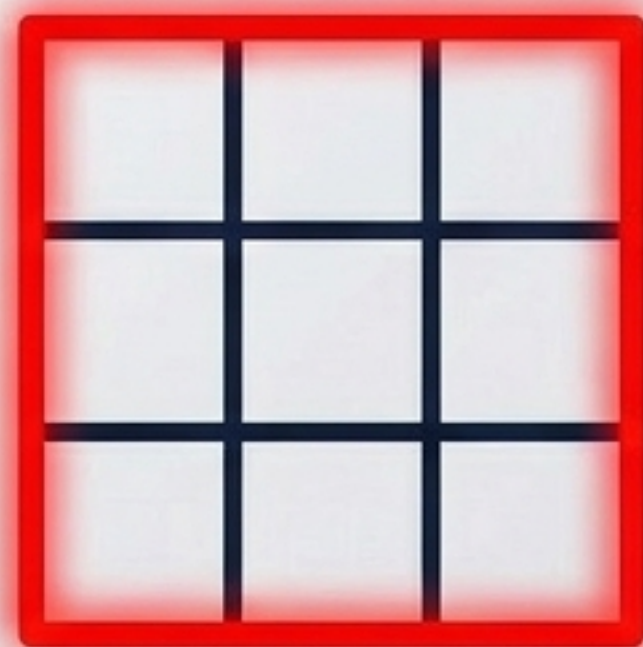
現在の行番号

行番号の増減値

ヒント： $i$  番目の方向の『1つ目の要素 (行の増減)』を取り出すインデックス記述は？

# 空欄 [エ] の構築：境界の最大値（壁）の定義

ここでは「下限 (1以上)」を超えないかに加え、「上限」を超えないかをチェックしています。  
このパズルは3x3とは限らず、任意の「一辺がNマス」の盤面を扱う汎用的な関数です。



... が [ エ ] 以下)

問うべき問い：この関数の引数として渡され、パズルの一辺のマス数  
（最大の壁）を保持している変数はどれでしょうか？

# 空欄 [オ] の構築：コードの美しい対称性を利用する

優れたコードには「対称性」があります。行（Y軸）の移動チェックロジックが完成すれば、列（X軸）のロジックはその鏡写しです。

$((\text{state.space}[1] + [\text{ウ}]) \text{ が } 1 \text{以上}) \text{ かつ}$

行のロジック

$((\text{state.space}[2] + [\text{オ}]) \text{ が } 1 \text{以上}) \text{ かつ}$

列のロジック

space[2] が列番号を表すならば、空欄 [オ] には direction 配列から「列番号の増減値」を取り出すロジックが入ります。  
[ウ] のインデックスを一つずらすだけで導かれます。

# 統合マトリックス：移動判定の全体構造

空欄を個別に解くのではなく、フレームワークとして俯瞰すると、この if 文がいかに完璧な矩形バリデーション（X/Y軸と最小/最大値の組み合わせ）を行っているかが視覚的に理解できます。

|                       | 変化量<br>(Modifier) | 境界下限<br>(Min) | 境界上限<br>(Max) |
|-----------------------|-------------------|---------------|---------------|
| Y軸判定<br>(行: space[1]) | [ ウ ]             | 1             | [ エ ]         |
| X軸判定<br>(列: space[2]) | [ オ ]             | 1             | [ キ ]         |

# プログラミング問題攻略の3つの原則

アルゴリズムの穴埋めはパズルではありません。設計図（コード）の構造を読み解き、現実の物理モデルと結びつけることで、必然的に導き出される論理の帰結なのです。



## 1. 物理モデルへの翻訳

抽象的な多次元配列は、必ず現実の動き（座標や十字キーなど）に図解して対応させる。



## 2. 対称性の発見

座標計算や境界チェックのコードブロックには対称性がある。片方が解ければ、もう片方は応用できる。



## 3. 状態遷移の追跡

リストやキューの操作はクラス仕様と照らし合わせ、「追加（生産）」と「取り出し（消費）」のバランスを確認する。