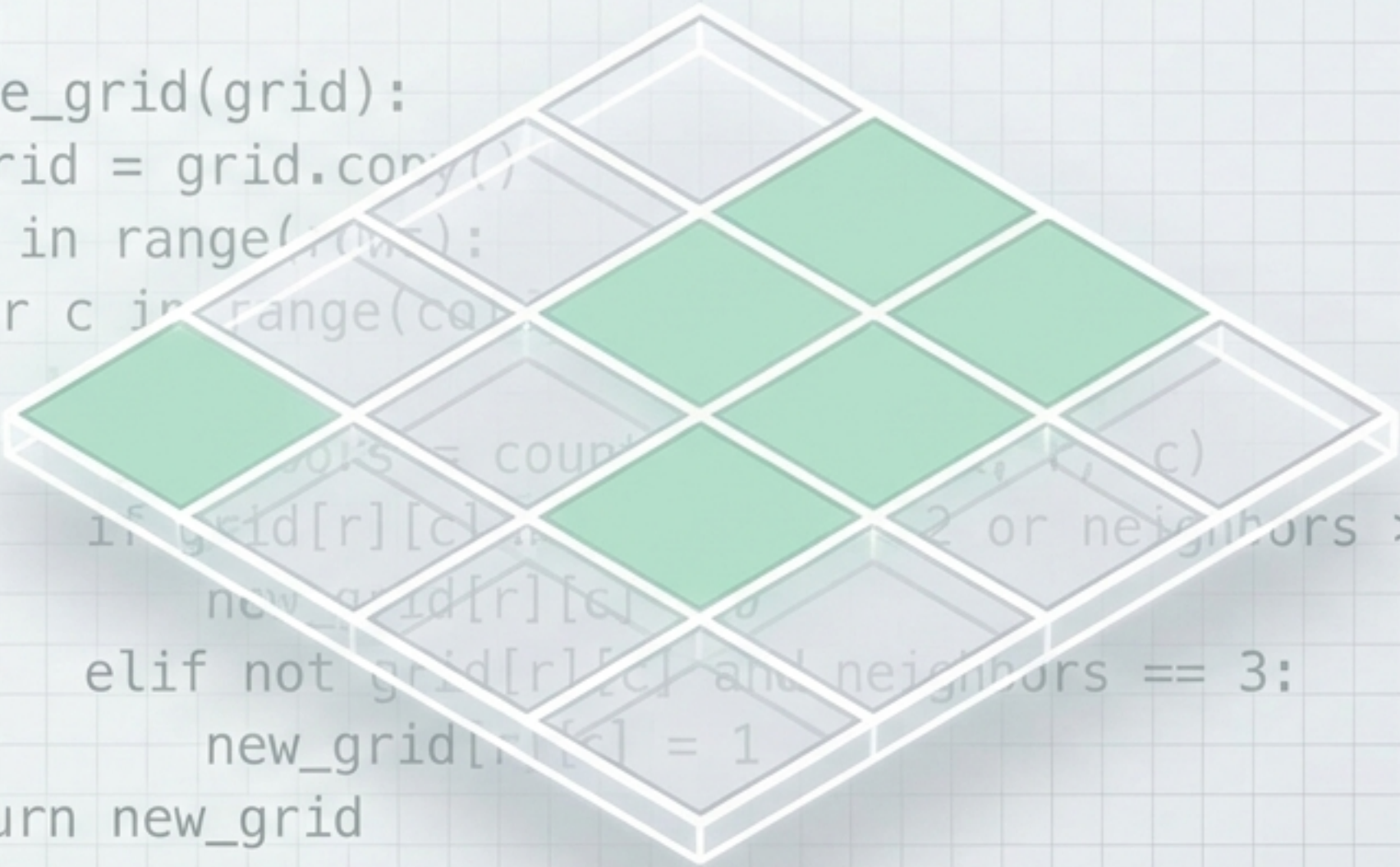


ライフゲーム実装の裏側

世代更新アルゴリズムのコード解剖とPython的思考

```
def update_grid(grid):  
    new_grid = grid.copy()  
    for r in range(rows):  
        for c in range(cols):  
            neighbors = count_neighbors(grid, r, c)  
            if grid[r][c] and (neighbors < 2 or neighbors > 3):  
                new_grid[r][c] = 0  
            elif not grid[r][c] and neighbors == 3:  
                new_grid[r][c] = 1  
    return new_grid
```



生命の掟：4つの状態遷移ルール

		隣接する生の数（周囲8マス中）																																					
		2つ・3つ（または3つのみ）	1つ以下 / 4つ以上																																				
現在の状態	死（0）	【誕生】 周囲がちょうど3つの場合のみ、次世代で「生」となる。 <table><tr><td>×</td><td>×</td><td>×</td></tr><tr><td>×</td><td>×</td><td>○</td></tr><tr><td>×</td><td>○</td><td>○</td></tr></table> → <table><tr><td>×</td><td>×</td><td>×</td></tr><tr><td>×</td><td>○</td><td>×</td></tr><tr><td>×</td><td>×</td><td>×</td></tr></table>	×	×	×	×	×	○	×	○	○	×	×	×	×	○	×	×	×	×	【過密】 リソース不足により、次世代では「死」となる。 <table><tr><td>×</td><td>○</td><td>○</td></tr><tr><td>○</td><td>○</td><td>○</td></tr><tr><td>×</td><td>○</td><td>×</td></tr></table> → <table><tr><td>×</td><td>×</td><td>×</td></tr><tr><td>×</td><td>×</td><td>×</td></tr><tr><td>×</td><td>×</td><td>×</td></tr></table>	×	○	○	○	○	○	×	○	×	×	×	×	×	×	×	×	×	×
	×	×	×																																				
×	×	○																																					
×	○	○																																					
×	×	×																																					
×	○	×																																					
×	×	×																																					
×	○	○																																					
○	○	○																																					
×	○	×																																					
×	×	×																																					
×	×	×																																					
×	×	×																																					
生（1）	【生存】 適切な密度（2～3）により、次世代でも「生」を維持する。 <table><tr><td>×</td><td>○</td><td>○</td></tr><tr><td>○</td><td>○</td><td>×</td></tr><tr><td>×</td><td>○</td><td>×</td></tr></table> → <table><tr><td>×</td><td>○</td><td>×</td></tr><tr><td>×</td><td>○</td><td>×</td></tr><tr><td>×</td><td>○</td><td>×</td></tr></table>	×	○	○	○	○	×	×	○	×	×	○	×	×	○	×	×	○	×	【過疎】 孤独により、次世代では「死」となる。 <table><tr><td>×</td><td>○</td><td>×</td></tr><tr><td>×</td><td>○</td><td>×</td></tr><tr><td>×</td><td>×</td><td>×</td></tr></table> → <table><tr><td>×</td><td>×</td><td>×</td></tr><tr><td>×</td><td>×</td><td>×</td></tr><tr><td>×</td><td>×</td><td>×</td></tr></table>	×	○	×	×	○	×	×	×	×	×	×	×	×	×	×	×	×	×	
×	○	○																																					
○	○	×																																					
×	○	×																																					
×	○	×																																					
×	○	×																																					
×	○	×																																					
×	○	×																																					
×	○	×																																					
×	×	×																																					
×	×	×																																					
×	×	×																																					
×	×	×																																					

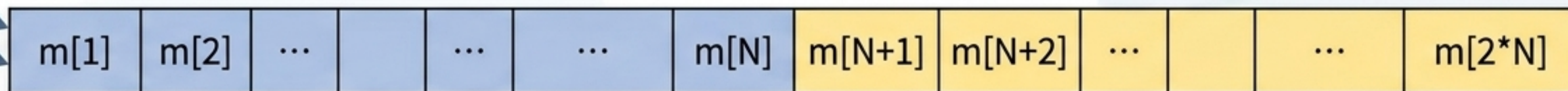
データ構造の変換：2Dグリッドを1D配列へ折りたたむ

2次元の概念的盤面

	1	2	...	j	...	N
1	m[1]	m[2]	...	...	...	m[N]
2	m[N+1]	m[N+2]	...	...	...	m[2*N]
⋮	...	...	...	...	...	...
k	...	...	...	...	...	...
⋮	...	...	...	...	...	...
N	...	...	...	...	...	...

インデックス計算式： $(k - 1) * N + j$

} $(k - 1) * N$ 目的の行 (k) に到達するまでにスキップする「k-1行分」のオフセット
} $+ j$ その行の先頭から右へ進む列の位置



1行目

2行目

アーキテクチャ：世代更新関数 update の4フェーズ



① 退避 (Save)

現在の盤面状態を別配列 (temp) にスナップショットとしてコピーする。



② 境界設定 (Boundary)

盤面の端で隣の行を巻き込まないように、探索範囲 (a, b) を制限する。



③ 走査とカウント (Scan)

周囲最大8マス走査し、し、盤外を除外しながら生きたマスをカウント。



④ 状態更新 (Update)

カウント結果とルールに基づき、次世代の盤面 (m) を書き換える。

```
function update(m)
    copy(m, temp) // 配列 m を配列 temp にコピーして退避する
    clear(in) // 配列 m の全ての要素の値を 0 にする
    for(i を 1 から N×N まで 1 ずつ増やす)
        if(i-1 が N で割り切れる)
            a ← 0
            b ← 1
        elseif(i が N で割り切れる)
            a ← -1
            b ← 0
        endif
        e ← 0
        for(y を -1 から 1 まで 1 ずつ増やす)
            for(x を a から b まで 1 ずつ増やす)
                if((y と 0 が等しくない) or (x と 0 が等しくない))
                    c ← i + y×N + x
                    if((c が 1 以上) and (c が N×N 以下))
                        if(
                            e ← e + 1
                        )
                    endif
                endif
            endfor
        endfor
        // 生死を判定する
        if(
            m[i] ← 1
        elseif((temp[i] と 1 が等しい) and ((e と 2 が等しい) or (e と 3 が等しい)))
            m[i] ← 1
        endif
    endfor
endfunction
```

ブロック1：同時性を担保する「状態の退避」

CODE

```
copy(m, temp)  
// 配列mを配列tempにコピーして退避する  
clear(m)  
// 配列mの全ての要素の値を0にする
```

DIAGRAM



② Why? なぜ退避が必要か？

世代更新は「全マス同時」に行われるルール。同一の配列を直接書き換えると、次のマスの判定時に「書き換え途中の未来のデータ」を参照してしまい、連鎖的なバグが発生する。

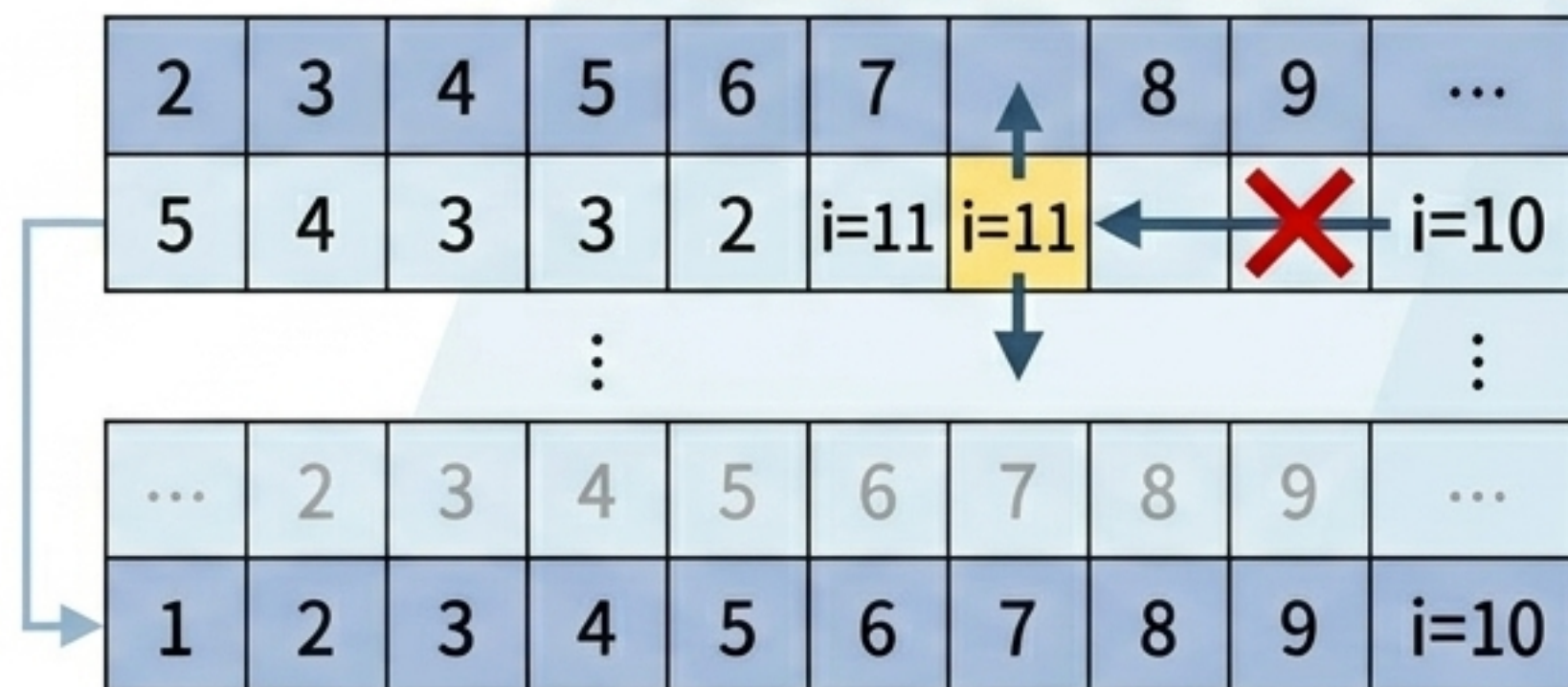
ブロック2：1D配列の弱点を塞ぐ「境界の安全装置」

CODE

```
if (i - 1) % N == 0: // 左端の場合
    a = 0
    b = 1
elif i % N == 0:    // 右端の場合
    a = -1
    b = 0
else:               // 通常
    a = -1
    b = 1
```

DIAGRAM

ワープ現象（隣の行の末尾を参照してしまう）



② Why? なぜ境界が必要か？

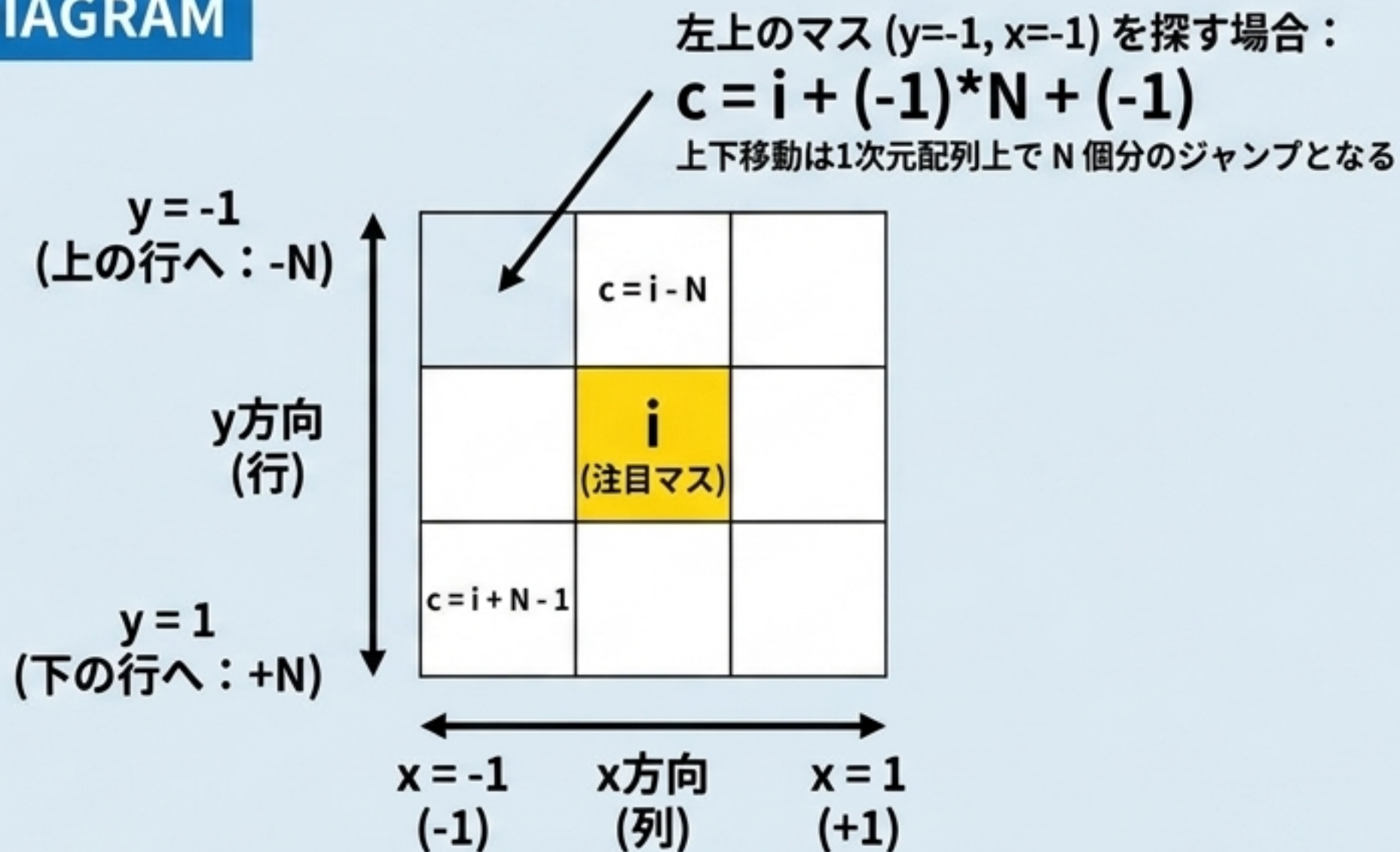
変数 a, b は左右（X 方向）の探索範囲。端のセルでは、はみ出す方向の探索値（-1 または 1）を 0 にすることで、異次元へのワープを防ぐガードレールとなる。

ブロック3：隣接マスのインデックス計算

CODE

```
for y = -1 to 1:
  for x = a to b:
    if y != 0 or x != 0:
      c = i + y * N + x
```

DIAGRAM



? Why? なぜインデックス計算が必要か？

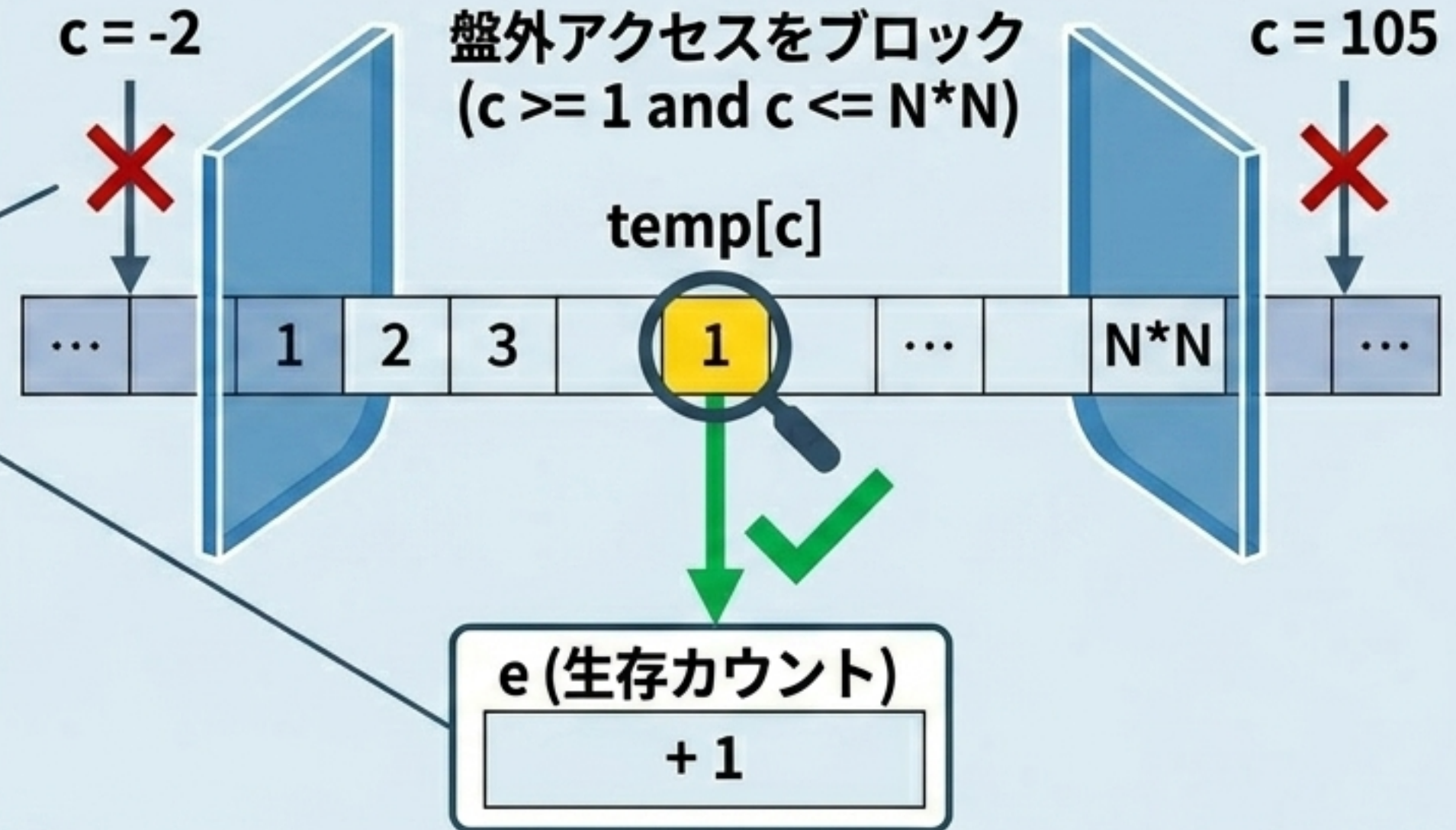
2次元的な位置関係（上下左右）を1次元配列上のインデックスに変換するため。Nが1行の要素数であるため、y方向の移動はNの加減算となる。

ブロック4：生存マスのカウントと盤外除外

CODE

```
if (c >= 1 and c <= N*N):  
    if temp[c] == 1:  
        e = e + 1
```

DIAGRAM



生死の判定は、クリアされた `m` ではなく、必ず退避しておいた現世代 `temp` の配列を参照する。

ブロック5：ルールに基づく次世代への状態更新

CODE

```
// 生死を判定する
if temp[i] == 0 and e == 3:
    m[i] = 1 // 誕生
elif temp[i] == 1 and (e == 2 or e == 3):
    m[i] = 1 // 生存
```

過疎・過密の記述がない理由

ブロック1でclear(m)によって全マスをすでに「死(0)」で初期化しているため、誕生と生存の条件を満たしたマスだけを「生(1)」に上書きすればよい。極めて無駄のないエレガントな設計。

DIAGRAM

次世代の状態

誕生 (Birth)	死 (Over)
生存 (Survival)	死 (Under)

テストとデバッグ：世代進行のループ回数の罠

CODE COMPARISON

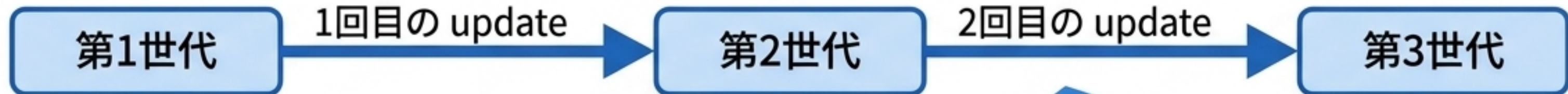
バグ発生コード

```
for (i を 1 から p まで 1 ずつ増やす)  
    update(m)
```

修正後コード

```
for (i を 2 から p まで 1 ずつ増やす)  
    // または (i を 1 から p-1 まで)
```

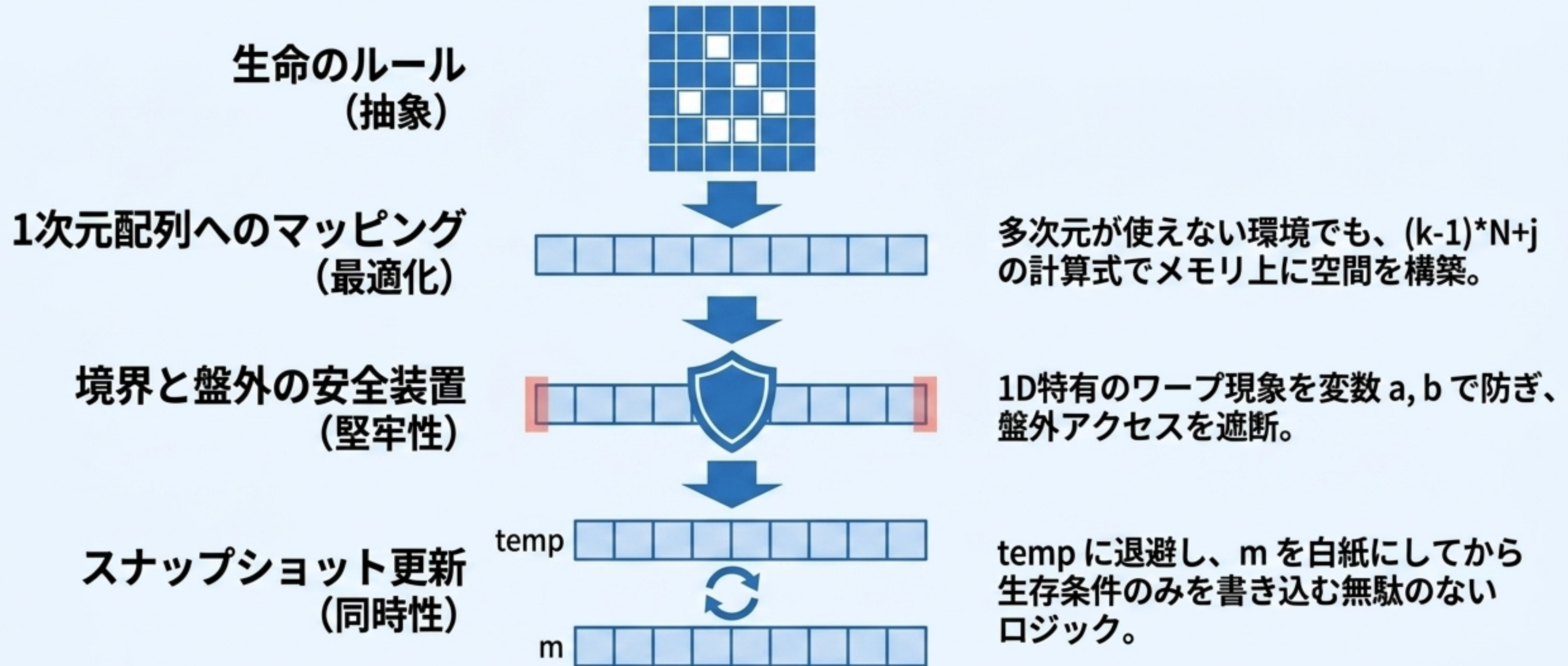
TIMELINE DIAGRAM



p = 2 (第2世代) をテストしたい場合、ループが1から2まで「2回」回ってしまうと、行き過ぎて第3世代になってしまう。

ノード数（世代）に到達するためには、遷移数（updateの回数）は「世代 - 1 回」であるべき。

Summary：アルゴリズムが命を吹き込む構造



複雑な生命の振る舞いは、丁寧な下準備とシンプルなルールの
ループによってコード上に美しく再現される。