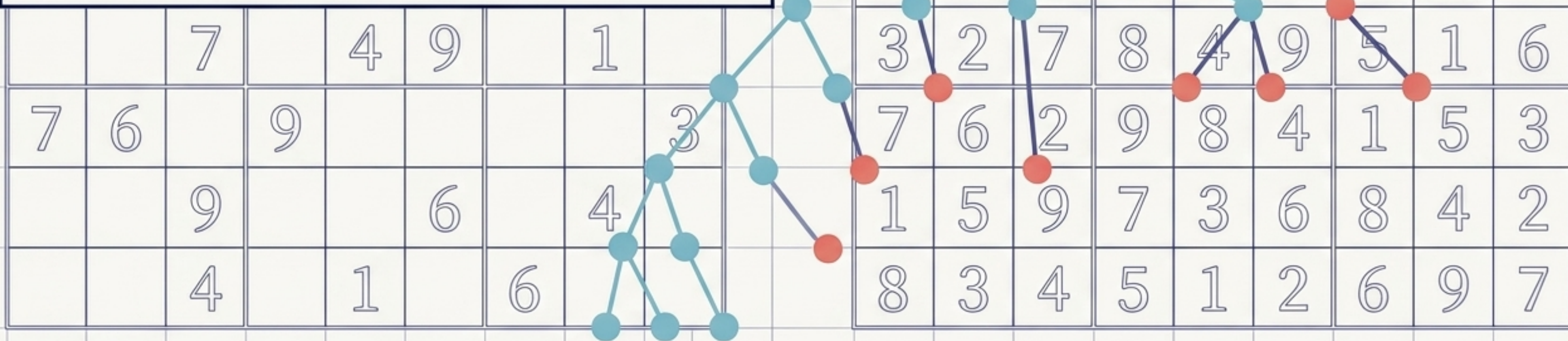


設問3 徹底解説： 再帰アルゴリズムにおける 「状態管理」の極意

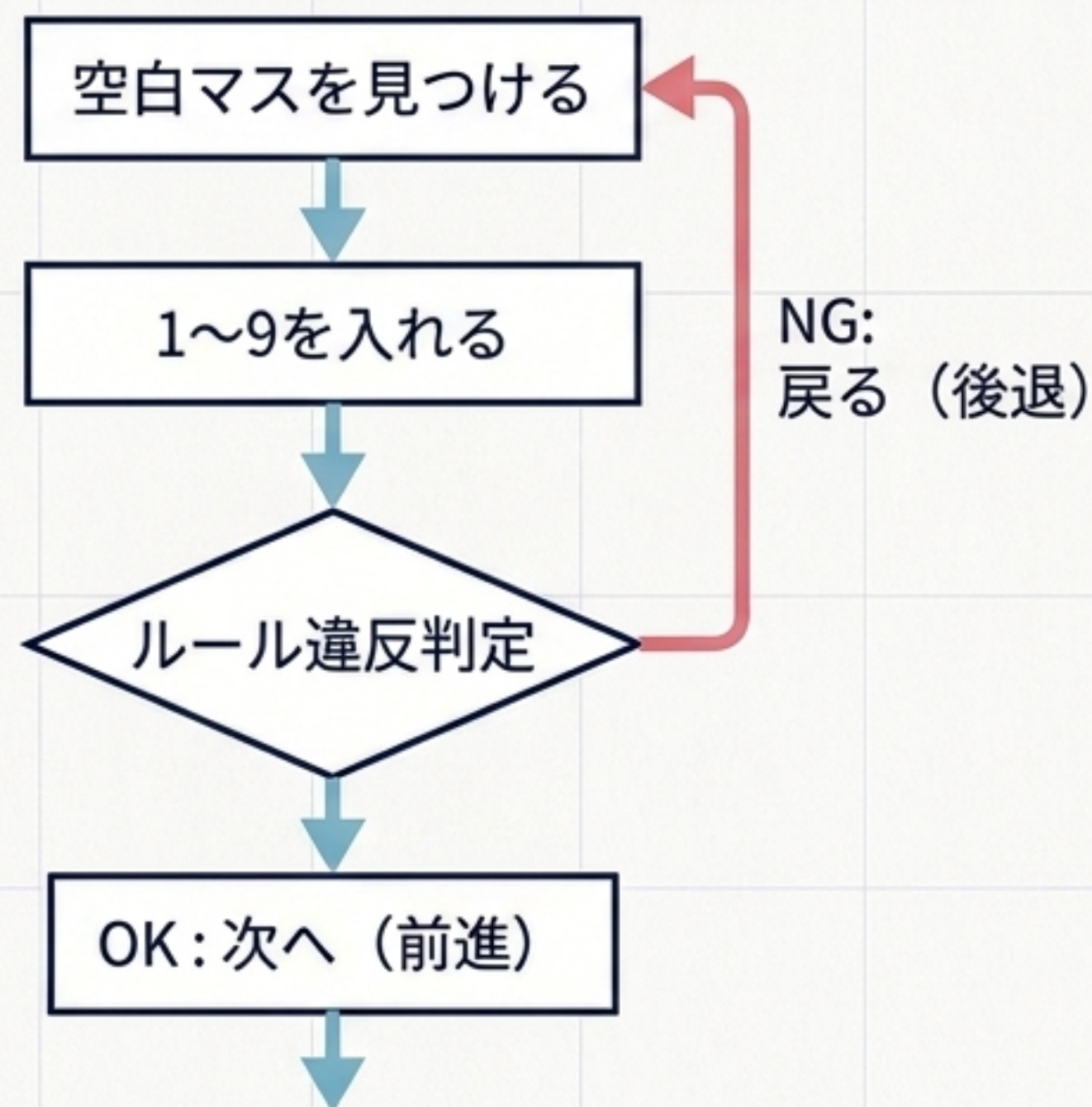
ナンプレ解答プログラムの高速化と、
バックトラッキングに潜む罠



応用情報技術者試験 午後
プログラミング (Explainer Deck)

基本戦略：深さ優先探索（DFS）による総当たり

左上から順に数字を当てはめ、ルール違反があれば「直前のマス」へ戻る(バックトラッキング)。
この単純な「前進」と「後退」の反復がパズルを解く基本原則。



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]
[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]	[26]
[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]
[72]	[73]	[74]	[75]	[76]	[77]	[78]	[79]	[80]

図3 配列board による盤面の表現

課題：再帰呼び出しと重複チェックの「指数関数的爆発」

探索範囲が広がるほど、毎回盤面全体を走査する重複チェックの負荷が劇的に増大する。
このボトルネックを解消しなければ、実用的な速度で解答に辿り着けない。

計算負荷 (チェック回数)

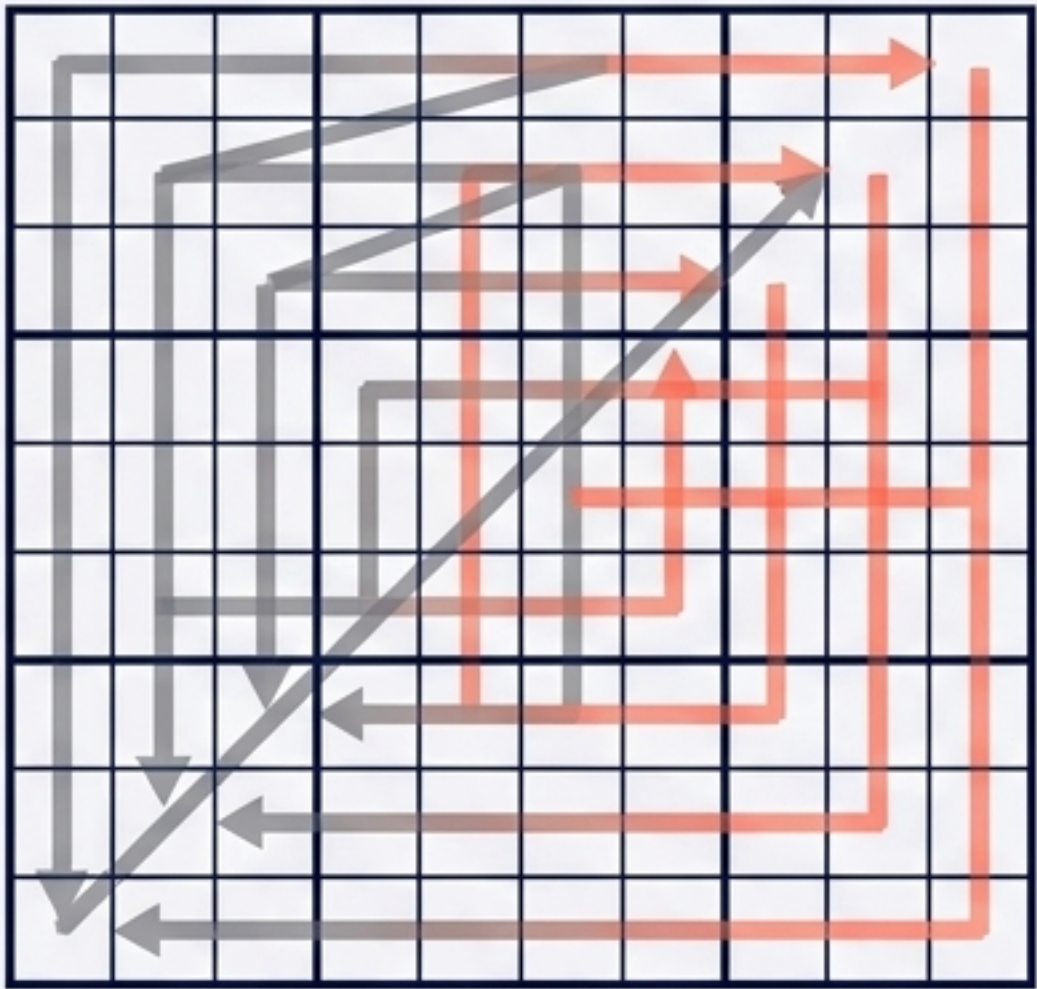
空白マスの数

row_ok
column_ok
row_ok
column_ok
frame_ok
frame_ok
check_ok
div
mod
column_ok
mod
check_ok
row_top
div
column_top
frame_top
row_top
frame_top
print_board
MAX_BOARD
row_ok
frame_top
column_ok

解決策：「リアルタイムのメモ帳」 データ構造Zの導入

毎回チェックするのをやめ、「各マスにどの数字が入れるか」をあらかじめ **1** と **0** で記録しておく大域変数（グローバル変数） **Z** を用意する。

Before: 重い全スキャン



After: **Z**による即時判定

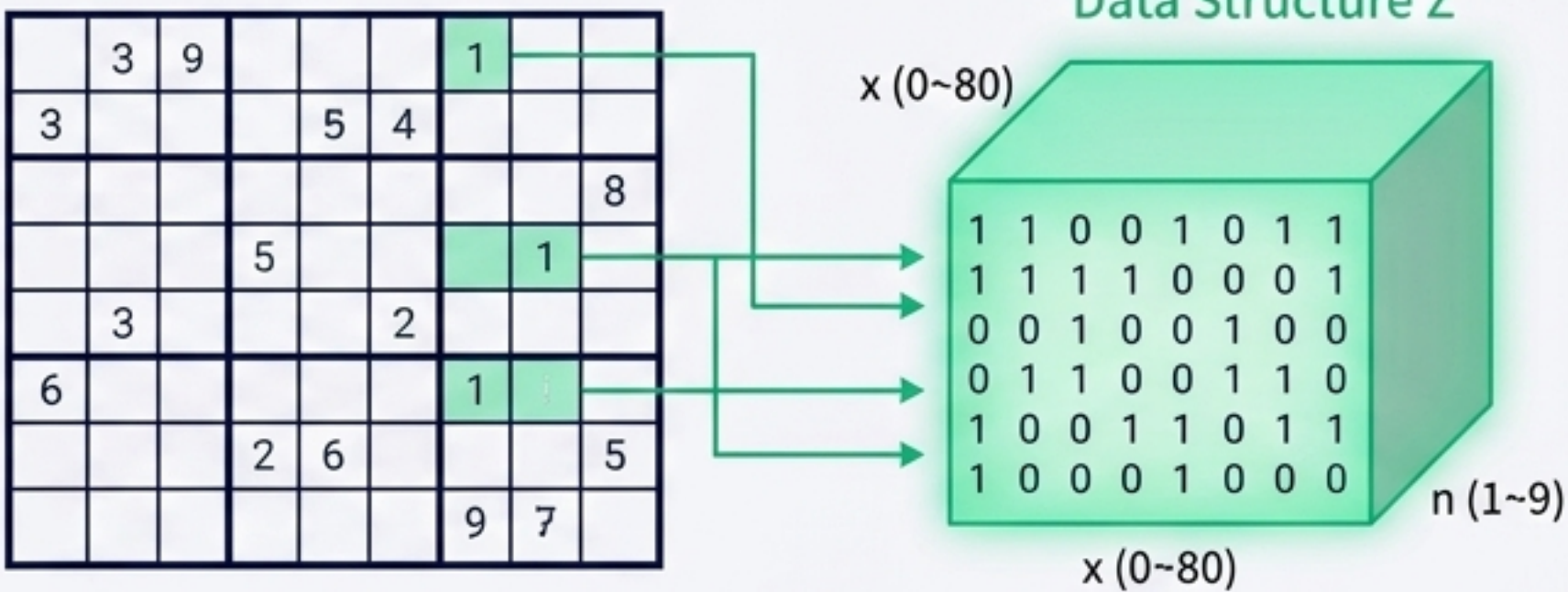
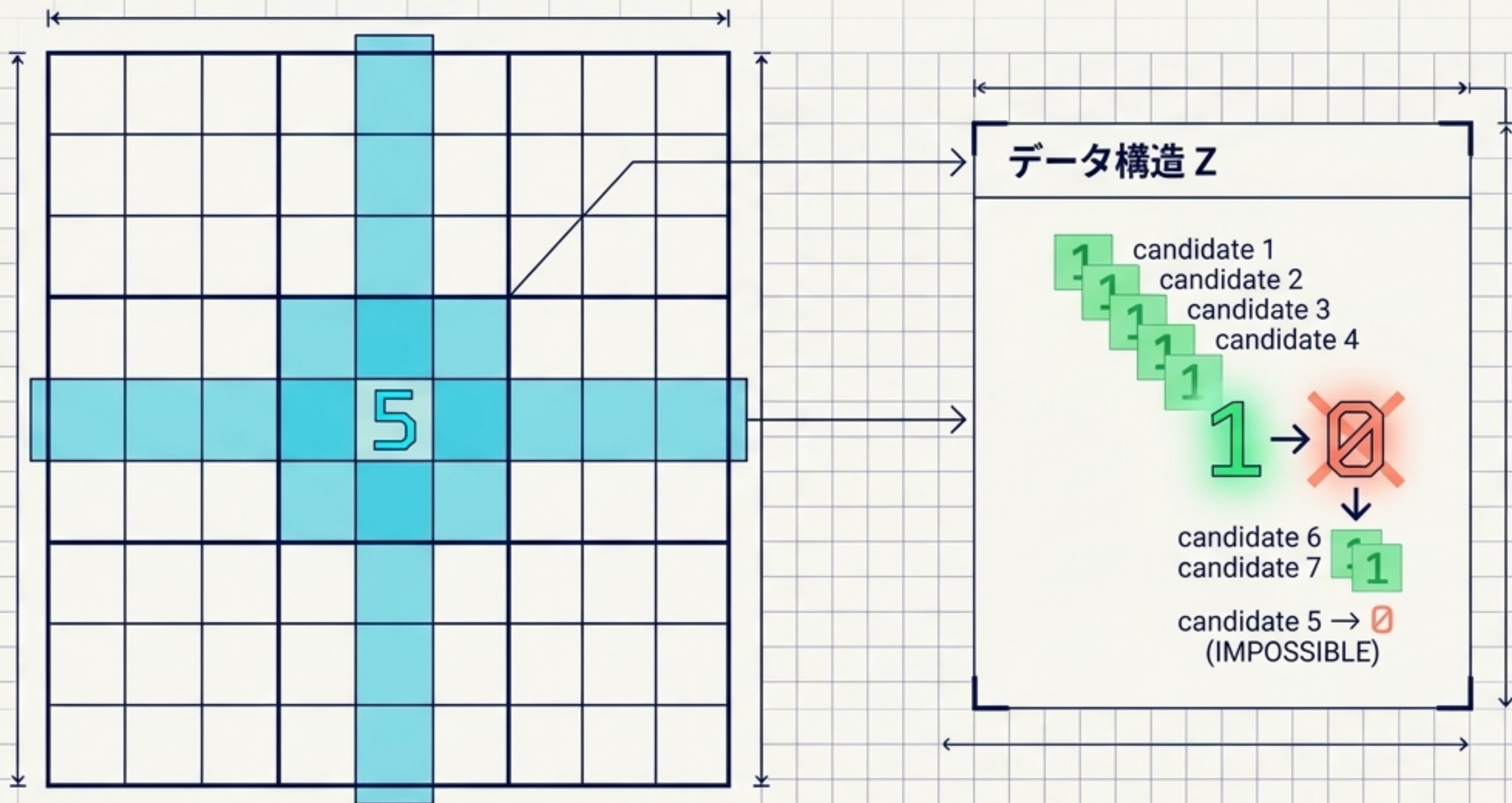


表2 データ構造 **Z** の初期化処理と更新処理

処理の名称	処理の内容
初期化処理	初期化時の盤面に対し、個々の空白のマスについて1～9の数字を入れた場合の重複チェックを行う。重複チェックの結果によって、初期化時の盤面状態で個々の空白のマスに入れることができない数字は、データ構造 Z の該当する数字の要素に 0 を設定する。それ以外の要素には 1 を設定する。
更新処理	空白のマスに数字を入れたとき、そのマスが含まれる横1行、縦1列、3×3の枠内の全てのマスを対象に、データ構造 Z の該当する数字の要素を 0 に更新する。

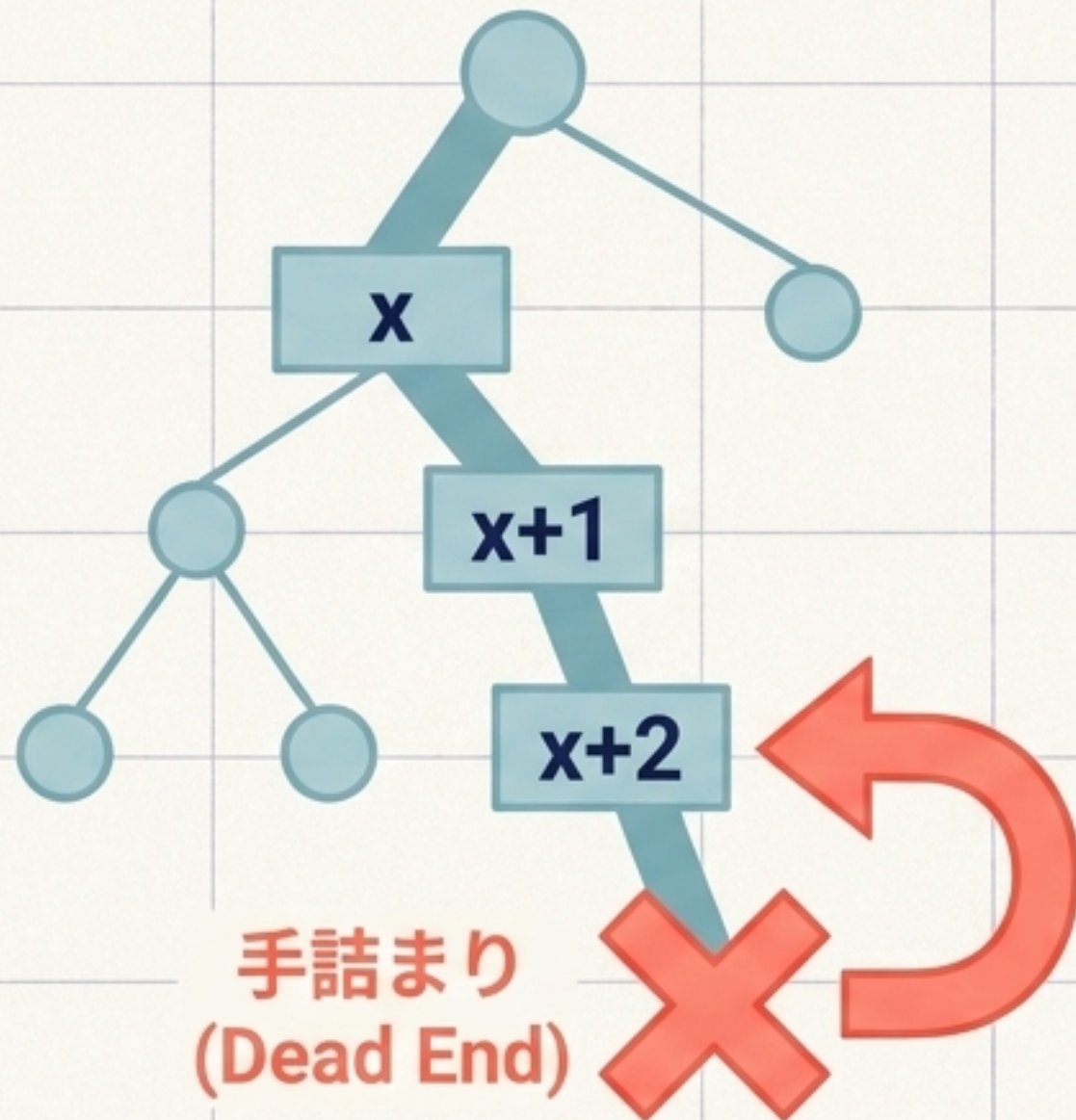
Zの更新メカニズム：波及する制限

空白マスに数字を入れた瞬間、影響範囲（縦・横・枠）にあるマスに対し、その数字の候補フラグを一斉に「0（不可能）」へ更新する。これにより次回以降のチェックが即座に完了する。



探索の壁：バックトラッキング（後戻り）の発生

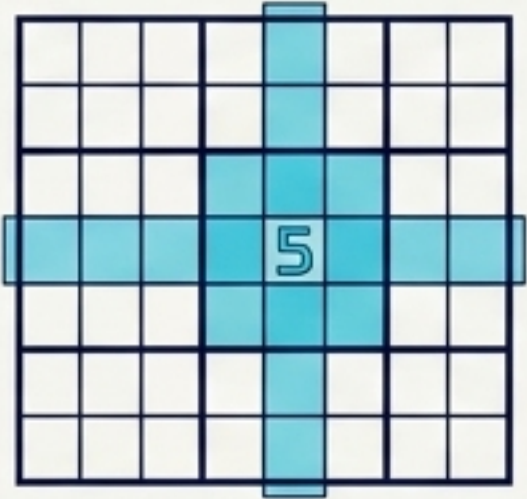
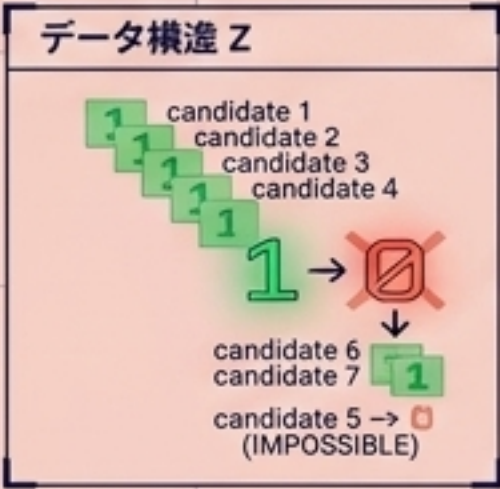
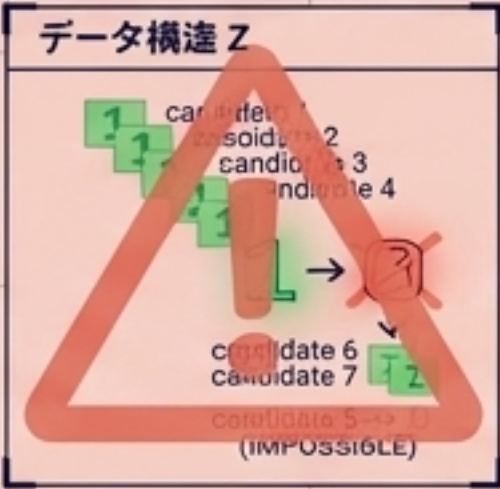
どの数字を入れてもルール違反になる「手詰まり」が発生した場合、マスを手詰まりの状態に戻し、一つ前のマスへ遡って別の可能性を探らなければならない。



- (3) データ構造 Z を参照し、(2) で見つけた空白のマスに入れることができる数字のリストを取得し、リストの数字を順番に入れる。
 - (3-1) 入れる数字がある場合、①処理 A を行った後、マスに数字を入れる。
その後、データ構造 Z の更新処理を行い、(2) に進んで次の空白のマスを見つける。
 - (3-2) $\backslash[48;2;255;127;80;179]$ 入れる数字がない場合、マスを空白に戻し、②処理 B を行った後、一つ前に数字を入れたマスに戻り、戻ったマスで取得したリストの次の数字から再開する。

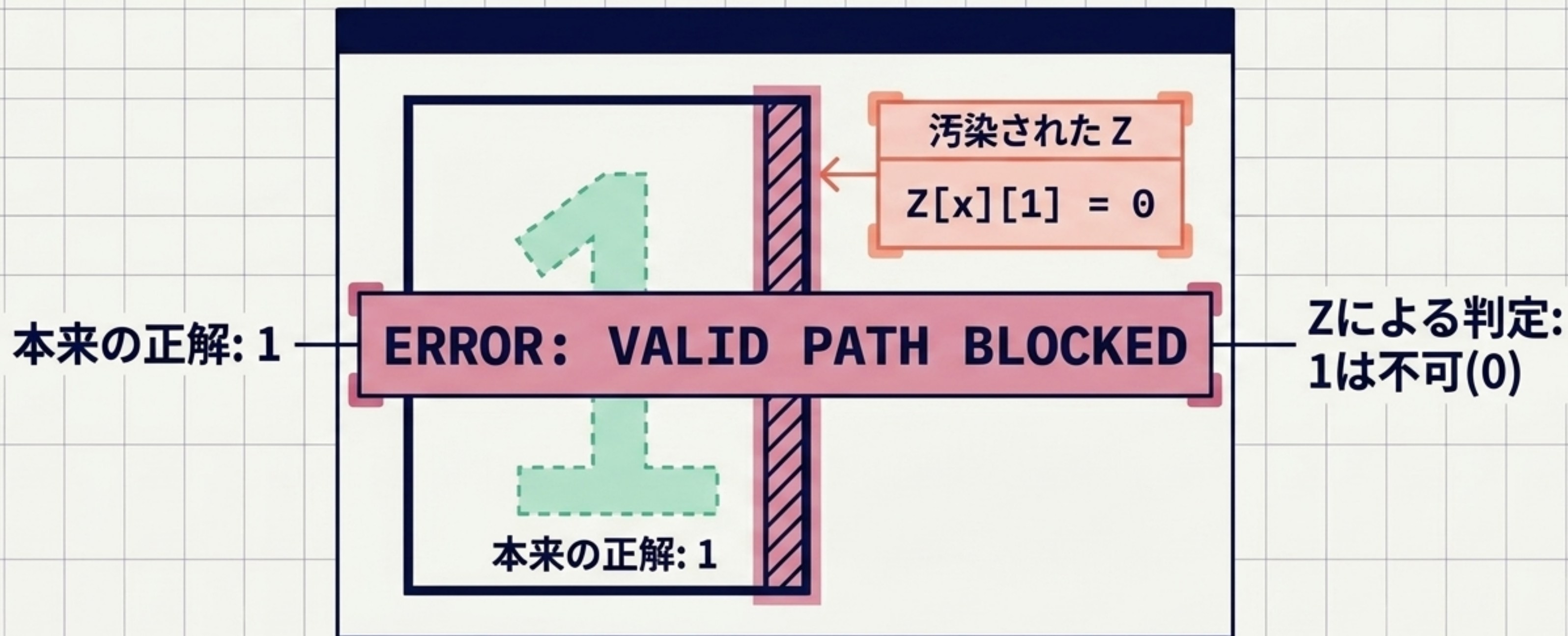
根本問題：グローバル変数 Z の「状態汚染」

盤面 board は元に戻せても、大域変数である Z は自動的にには戻らない。
失敗した未来の制限 (0) が、過去に持ち越されたまま残ってしまう。

Board (ローカルな巻き戻し)	正常: 空白(0)に戻る
	
Z (グローバルな汚染)	異常: 不当な制限(0)が残留
	

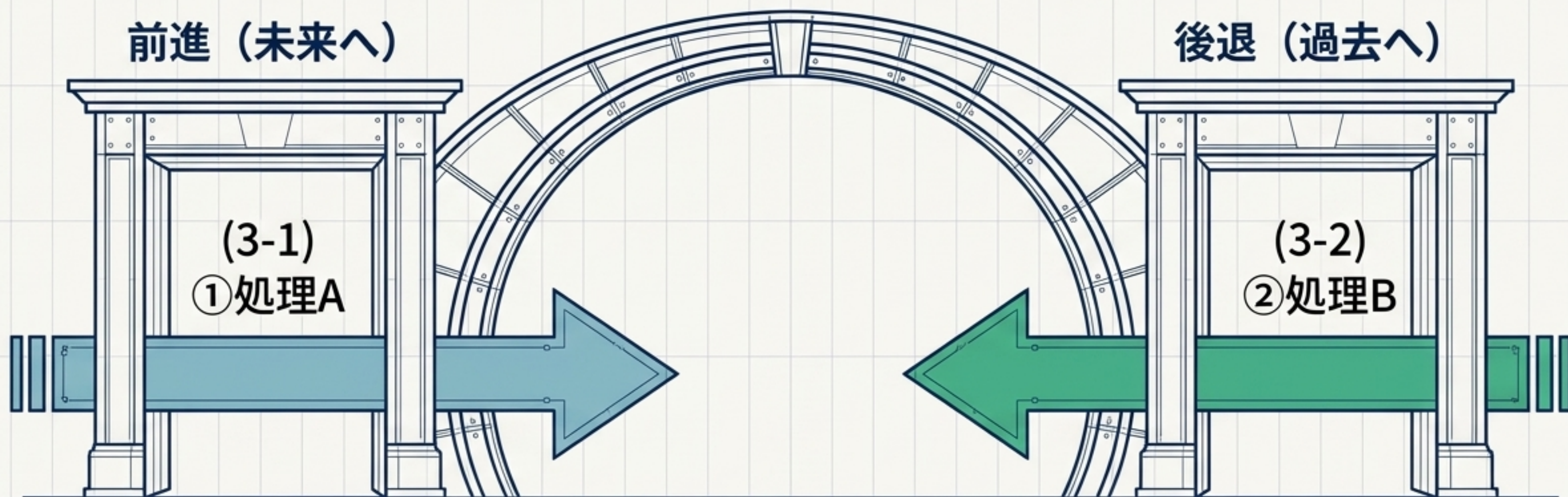
思考実験：もしそのまま探索を再開したら？

誤った制限が残っているため、本来入れられるはずの「正解の数字」がルール違反と誤判定される。このままでは永遠に正しい解答に辿り着けない。



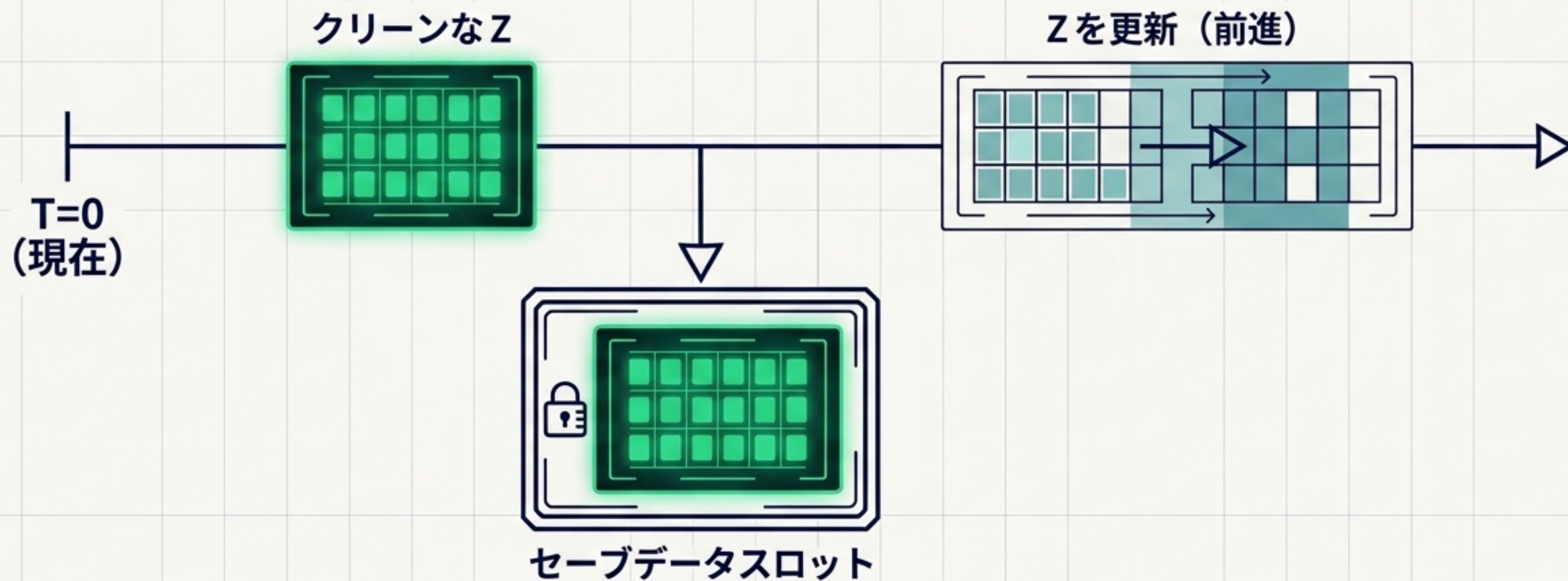
解決のフレームワーク：時を戻す2つの「処理」

設問3の核心。後戻りに対応するためには、データ構造Zの「更新前の姿」を保証する仕組みを、プログラムの行きと帰りに組み込む必要がある。



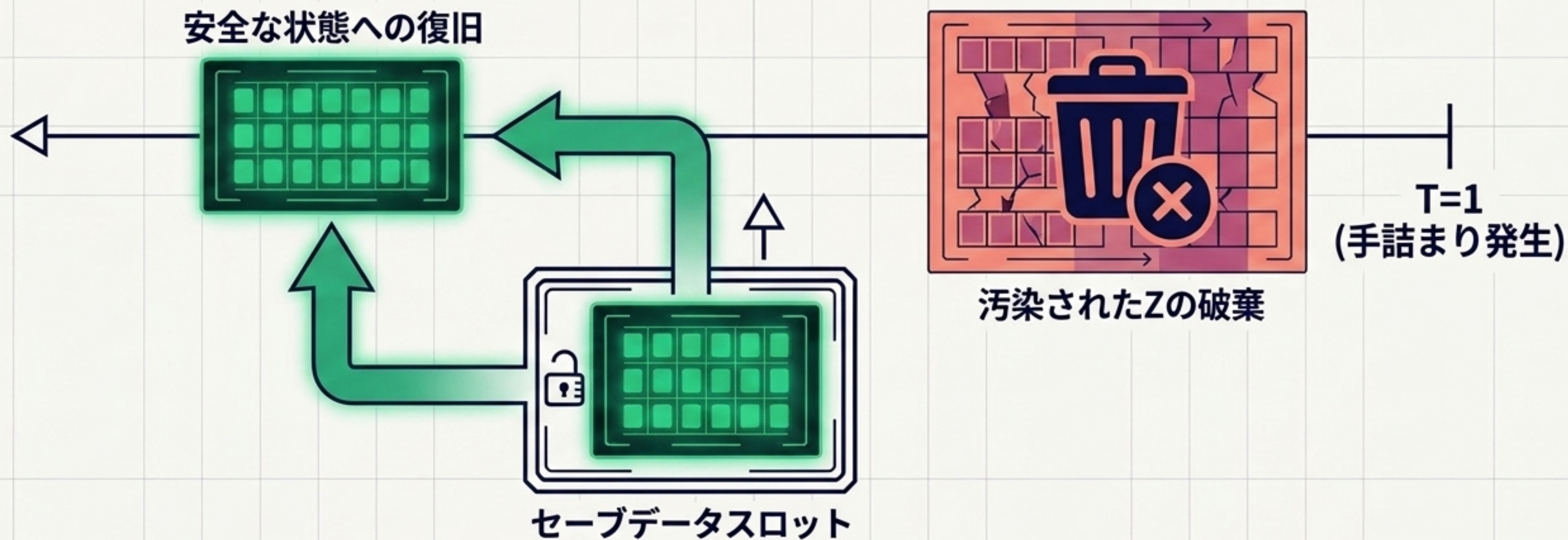
深掘り：処理A（前進時）の役割

未知のマスの数字を入れてZを更新する「直前」。万が一手詰まりになった時に備え、現在の安全なZの状態を、別の安全な場所へ確保しておく。



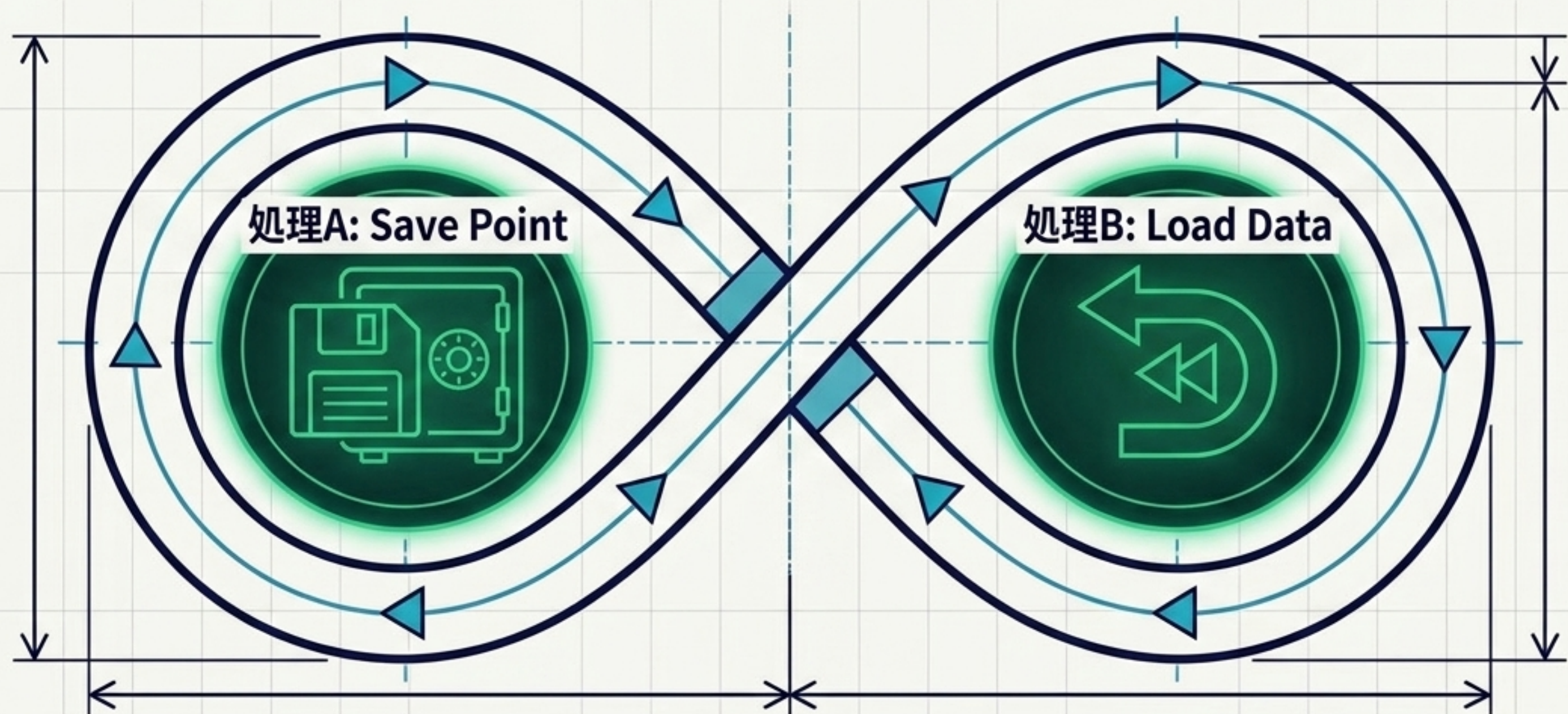
深掘り：処理B（後退時）の役割

手詰まりで一つ前のマスへ戻る「直後」。大域変数の Z を、処理Aで見込んでおいた「確保済みの安全な状態」で上書きし、完全に時を巻き戻す。



メタファーへの変換：ゲームの「セーブ&ロード」

処理Aは「安全な状態のセーブ」、処理Bは「ゲームオーバー時のロード」。この対の操作によって初めて、グローバル変数を用いた再帰処理の整合性が保たれる。



結論：アルゴリズムの思考プロセスを言語化する

再帰処理において局所変数（ローカル）は自動で巻き戻るが、大域変数（グローバル）であるZは自力で巻き戻す必要がある。

「データ構造Z」をどう「退避」し、どう「復元」するべきか？ この思考プロセスを20文字以内で言語化すれば、それが設問3の完璧な正解となる。

