

```
def solve_sudoku(board):
    if row == 9:
        return True
    try:
        board[row][col] = num
    except ValueError:
        return False
    # row == True
    try:
        board[row][col] = num
    except ValueError:
        return False
```

2022s Pythonコード徹底解説

数独ソルバーのアルゴリズム設計と「例外エラー」の真実

応用情報技術者試験（午後問3）から学ぶ、深さ優先探索とバックトラッキングの極意

```
def solve_sudoku(board):
    for col in board[row]:
        col += 0
        if row == 9:
            return True
        # Backtrack
        return False
    try:
        board[row][col] = num
    except ValueError:
        # Backtrack
        return False
```

完璧な結果の直後に現れる赤い文字

```
[2, 3, 1, 4, 9, 5, 7, 6, 8]
[8, 4, 9, 2, 6, 7, 3, 1, 5]
[5, 6, 7, 1, 3, 8, 4, 2, 9]
[...]
```

An exception has occurred, use %tb to
see the full traceback.

エラーではなく、強制終了の合図

`solve(x)` 関数がゴールに到達
($x > \text{MAX_BOARD} - 1$)

`print_board()` で盤面を出力

`sys.exit()` を実行

Jupyter / Colab 環境の仕様

`SystemExit`例外としてキ
ャッチされ、Tracebackが
表示される

✓ プログラムは100%正常に動作しています

Before: 力技の終了 (sys.exit)

```
if x > MAX_BOARD - 1:  
    print_board()  
    sys.exit() # 探索をここで強制切断
```



簡単だが、**実行環境**によっては**例外エラーのログ**が出る

After: エレガントな終了 (Pythonic)

```
if x > MAX_BOARD - 1:  
    print_board()  
    return True # 解が見つかったことを返す
```



深い再帰の底から
「**解けたよ(True)**」という
シグナルが、呼び出し元へ
ドミノ倒しのように伝播して
安全に終了する

数独ソルバーの3大コンポーネント

データ構造 (Data Structure)

board (長さ81の1次元配列)

盤面の状態管理 (0は空白)

ルールOKなら
盤面を更新

検証エンジン (Rule Checkers)

row_ok, column_ok, frame_ok, check_ok

数独のルール (行・列・3x3枠の重複) チェック

次のマスへ進む

探索エンジン (Solver Engine)

solve(x)

深さ優先探索 (DFS) と
バックトラッキングによる総当たり

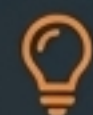
マスに数字を仮置きし、
判定を依頼

データ構造：2次元の盤面を1次元配列へフラット化

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]
[18]	[19]	[20]	[22]	[23]	[24]	[25]	[26]	[27]
[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]
[31]	[32]	[33]	[34]	[35]	[36]	[37]	[38]	[40]
[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[59]
[51]	[52]	[53]	[54]	[55]	[56]	[57]	[58]	[60]
[61]	[62]	[63]	[64]	[65]	[66]	[67]	[68]	[70]
[72]	[73]	[74]	[75]	[76]	[77]	[78]	[79]	[80]

インデックス操作の基本ルール:

- 配列の添字 x は 0 から 80。
- 値が 0 のマスは「空白」を意味する。

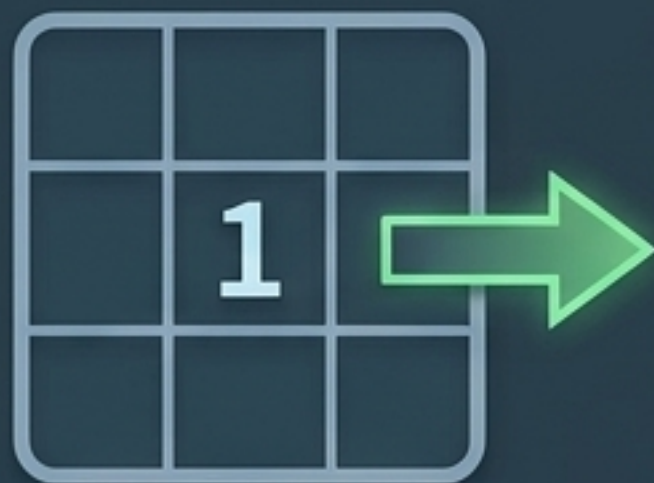


Insight

2次元配列 (`board[row][col]`) ではなく1次元配列 (`board[x]`) を使うことで、再帰関数内で「次のマスへ進む ($x + 1$)」処理が圧倒的にシンプルになる。

探索の基本戦略：とりあえず進む、ダメなら戻る

前進



空白マスに1を入れてみる。ルールOKなら次のマスへ。

手詰まり



進んだ先で1~9 どれを入れてもルール違反になる状態。

後戻り（バックトラック）



1つ前のマスに戻り、マスを「空白（0）」に初期化し、次の候補を試す。

Key Concept

この「試行錯誤」のプロセスを自動化する心臓部が再帰呼び出し（Recursion）である。

ルールチェック①：行の判定 (row_ok)

[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]
[36]	[37]	[38]	[39]	[40]	[41]	[42]	[43]	[44]
[45]	[46]	[47]	[48]	[49]	[50]	[51]	[52]	[53]
[54]	[55]	[56]	[57]	[58]	[59]	[60]	[61]	[62]
[63]	[64]	[65]	[66]	[67]	[68]	[69]	[70]	[71]
[72]	[73]	[74]	[75]	[76]	[77]	[78]	[79]	[80]

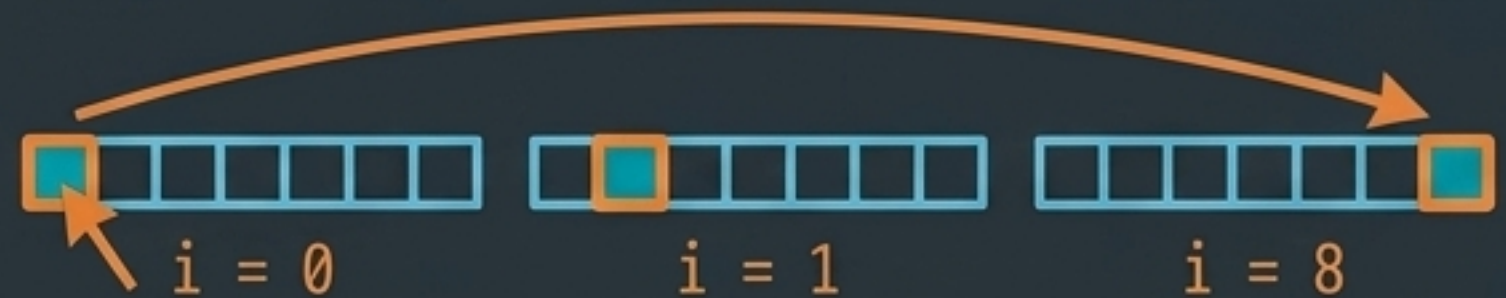
空欄 [オ]：行の左端を求める

$\text{row_top} = \text{div}(x, 9) * 9$

計算例： $12 // 9 = 1$ (1 行目)。
 $1 * 9 = 9$ (左端は 9)。

空欄 [カ]：重複チェック

$\text{board}[\text{row_top} + i] == n$



i が 0~8 まで動くことで、左端から右へ 9 マス分を順番にスキャンする。

ルールチェック②：列の判定 (column_ok)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[9]	[1]	[10]	[11]	[12]	[14]	[15]	[16]	[17]
[18]	[19]	[20]	[22]	[23]	[24]	[25]	[26]	[27]
[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]
[31]	[32]	[33]	[34]	[35]	[36]	[37]	[34]	[40]
[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[59]
[51]	[52]	[53]	[54]	[55]	[56]	[57]	[58]	[60]
[61]	[62]	[63]	[64]	[65]	[66]	[67]	[68]	[70]
[72]	[73]	[74]	[75]	[76]	[77]	[78]	[79]	[80]

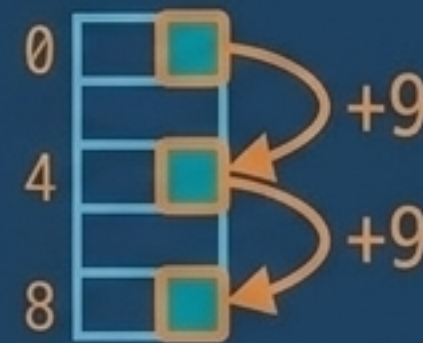
空欄 [キ]：列の上端を求める

$\text{column_top} = \text{mod}(x, 9)$

計算例： $34 \% 9 = 7$ （余りがそのまま最上段のインデックスになる）。

空欄 [ク]：重複チェック

$\text{board}[\text{column_top} + 9 * i] == n$



$+ 9 * i$ によって、1次元配列の中で「真下に1段下がる」動きを実現している。

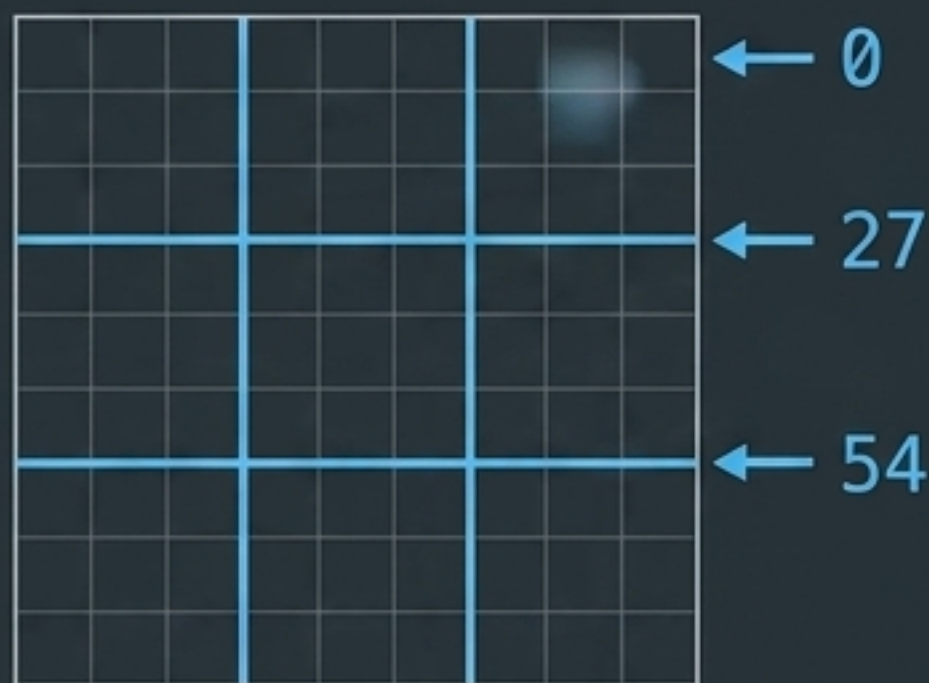
ルールチェック③：最難関の数式 (frame_ok)

3x3ブロックの左上端を特定する

$$(\text{div}(x, 9) // 3) * 27 + (\text{mod}(x, 9) // 3) * 3$$

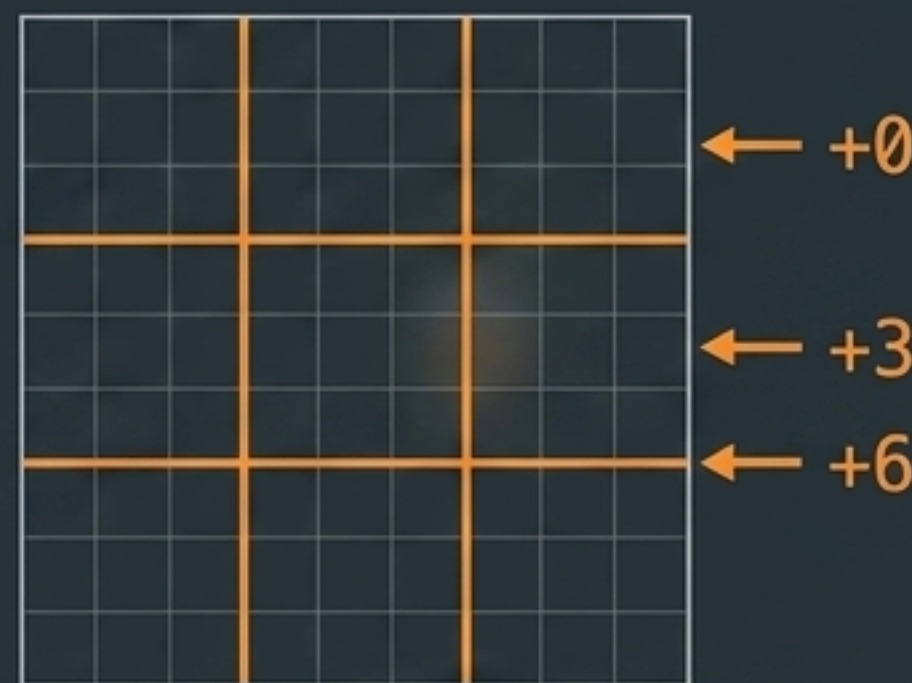
Left / 行のオフセット：

どの「横の3行ブロック」にいるか



Orange / 列のオフセット：

どの「縦の3列ブロック」にいるか



1次元配列で2次元のブロックを扱う際の数学的な美しさ

探索エンジン solve(x) の解剖：前進

[ア] board[x] != 0
対象マスが空白でない場合

```
if ([ ア ]):  
    solve([ イ ])  
else:  
    for n in range(1, 10):  
        if [ ウ ] == True):
```

[イ] x + 1

数字が入っているなら、
そのまま次のマスへ前進



[ウ] check_ok(n, x)

行・列・枠のすべてで
重複がないかチェック



探索エンジン solve(x) の解剖：バックトラック

要（かなめ）となる「証拠隠滅」の初期化处理

```
board[x] = n # 仮置き  
solve(x + 1) # 再帰で次へ  
board[x] = [ 工 ]
```

[工] 0

もし 0 に戻さなかったら？



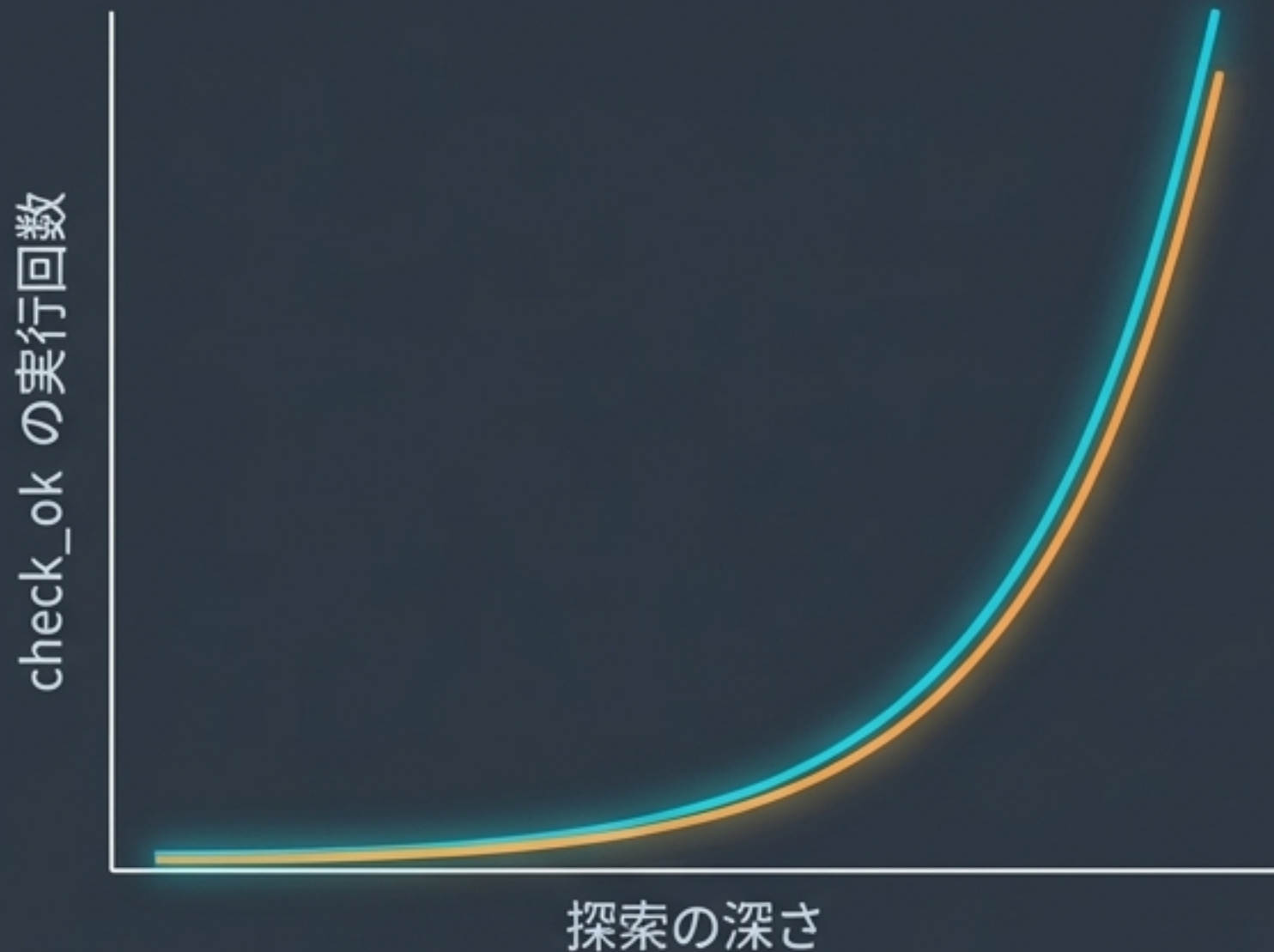
失敗したルート of 数字が盤面に残り、別ルートの探索時に「重複エラー」を誤発火させる汚染状態。

正しい処理



次の候補を試す前に、盤面をまっさらな 0 に戻す。

アルゴリズムの課題と改善（設問3）



「とりあえず入れる → 都度チェックする」素朴なDFS方式は、後半になるほど無駄な探索（どうせダメな数字のチェック）が天文学的に増える。

データ構造Zの導入

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、初期化時の盤面の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。

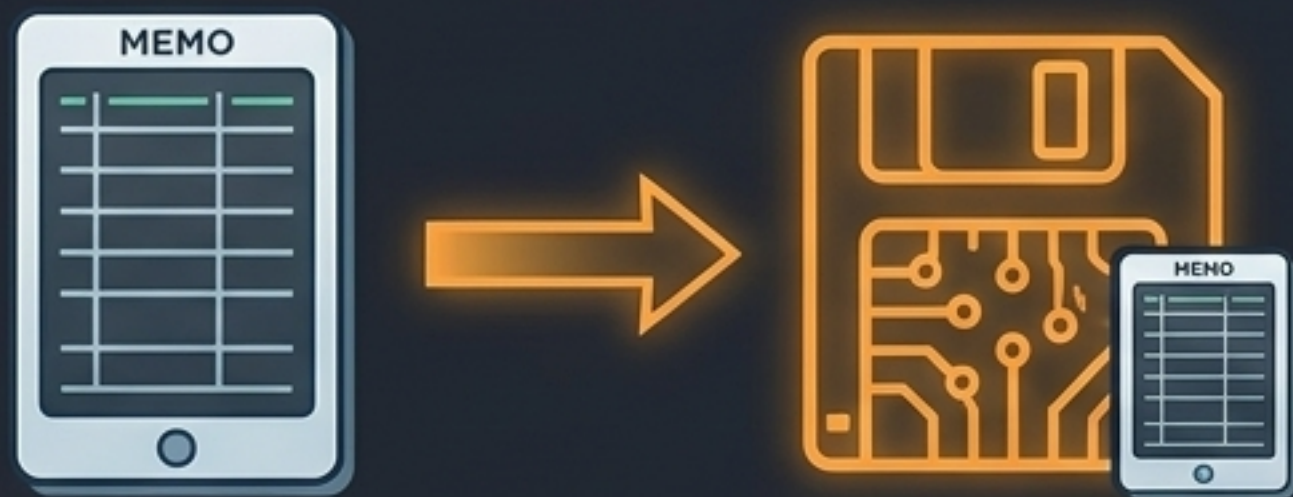
毎回チェックするのではなく、最初から「入れられる数字リスト」を持っておき、置くたびに候補を絞っていく方式（制約伝播）への進化。

データ構造Zの「退避」と「復元」

大域変数（グローバル変数）におけるタイムトラベル

前進：処理A

マスに数字を入れる直前



候補リスト (Z)

セーブポイント (退避)

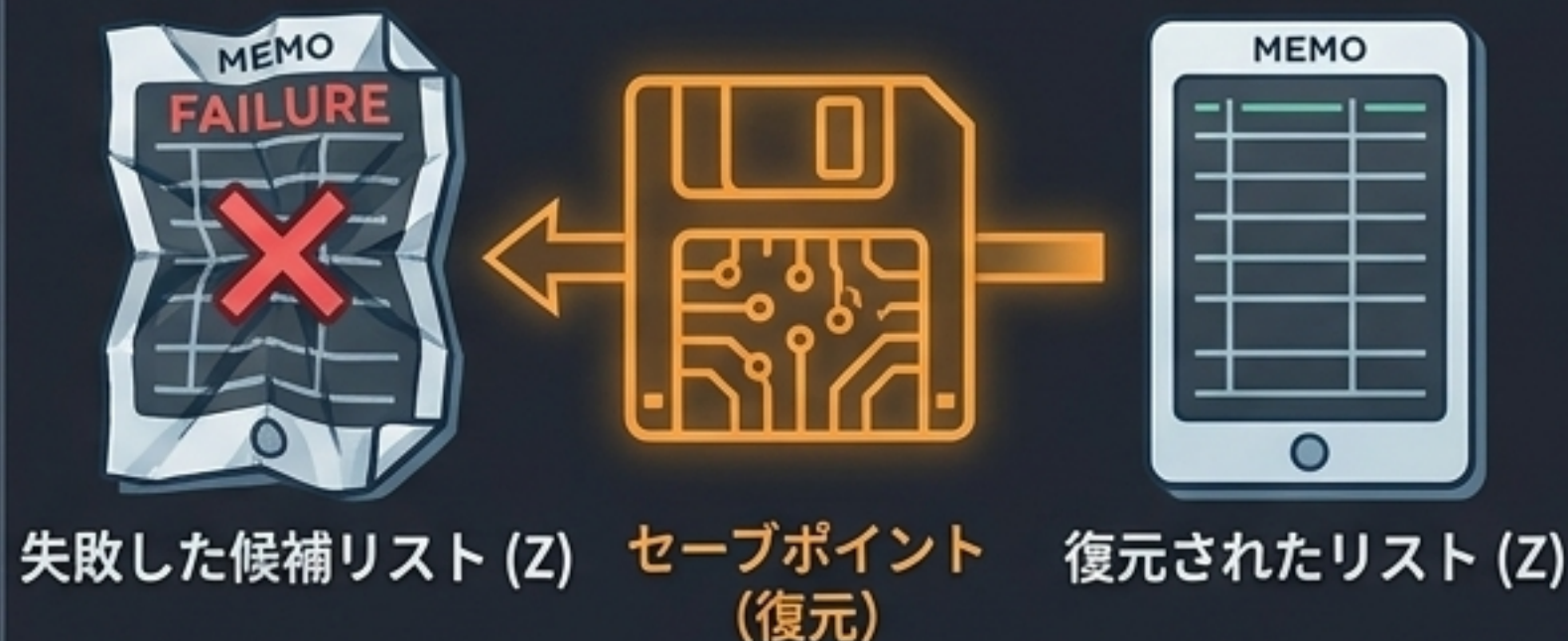
処理Aの内容

データ構造Zの更新前の状態を保存する

現在の候補リストをセーブポイントにコピー（退避）する。

後戻り：処理B

手詰まりになり、前のマスに戻る時



失敗した候補リスト (Z)

セーブポイント (復元)

復元されたリスト (Z)

処理Bの内容

データ構造Zを処理Aで保存した状態に復元

失敗によって狭まった候補リストを綺麗に巻き戻す（復元）。

Algorithmic Synthesis

Pythonコードから学ぶアルゴリズム設計の要点

1. データ構造の抽象化

2次元の盤面を1次元配列（長さ81）にマッピングする数理的アプローチ（divとmodの活用）が、コードを極限までシンプルにする。

2. 再帰と状態管理

バックトラッキングの本質は「仮説（前進）」と「初期化（後戻り）」。状態を元に戻す（`board[x] = 0`）ことではじめて全探索が成立する。

3. 空間と時間のトレードオフ

DFSの計算量爆発を防ぐため、メモリ（データ構造Z）を消費して処理速度を劇的に向上させる高度な最適化テクニック。

エラーの正体は、完璧に計算を終えたプログラムが発した「探索完了のサイン」でした。