Matrix

1. Valid Sudoku

Pattern: Arrays & Hashing

Problem Statement

Determine if a `9 x 9` Sudoku board is valid. Only the filled cells need to be validated according to the following rules: 1. Each row must contain the digits `1-9` without repetition. 2. Each column must contain the digits `1-9` without repetition. 3. Each of the nine `3 x 3` sub-boxes of the grid must contain the digits `1-9` without repetition.

Note:
- A valid Sudoku board (partially filled) is not necessarily solvable.
- Only filled cells need validation.
- Empty cells are represented by `'.'`.

Sample Input & Output

```
Input: board =
[["5","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
```

```
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: true
Explanation: All rows, columns, and 3x3 boxes satisfy Sudoku rules.
```

```
Input: board =
[["8","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: false
Explanation: There are two 8s in the top-left 3x3 sub-box.
```

```
Input: board =
[[".",".",".",".","5",".",".","1","."]
,[".","4",".","3",".",".",".",".","."]
,[".",".",".",".",".","3",".",".","1"]
,["8",".",".",".",".",".",".","2","."]
,[".",".","2",".","7",".",".",".","."]
,[".","1","5",".",".",".",".",".","."]
,[".",".",".",".",".","2",".",".","."]
,[".","2",".","9",".",".",".",".","."]
,[".",".","4",".",".",".",".",".","."]]
Output: false
Explanation: Column 5 has two 5s (at [0][4] and [8][4] is actually 8 -
but here, two 2s appear in column 1).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        # STEP 1: Initialize structures
        #   - Use sets to track seen digits in rows, cols, and boxes
        rows = [set() for _ in range(9)]
        cols = [set() for _ in range(9)]
        boxes = [set() for _ in range(9)]

        # STEP 2: Main loop / recursion
        #   - Iterate over every cell (i, j)
        for i in range(9):
            for j in range(9):
                val = board[i][j]
                if val == '.':
                    continue  # Skip empty cells

                # STEP 3: Update state / bookkeeping
                #   - Compute box index: (i//3)*3 + j//3 maps 3x3 blocks
                box_idx = (i // 3) * 3 + (j // 3)

                # Check for duplicates in row, col, or box
                if (val in rows[i] or
                    val in cols[j] or
                    val in boxes[box_idx]):
                    return False

                # Add current value to trackers
                rows[i].add(val)
                cols[j].add(val)
                boxes[box_idx].add(val)

        # STEP 4: Return result
        #   - If no duplicates found, board is valid
        return True

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal valid Sudoku
```

```python
board1 = [
    ["5","3",".",".","7",".",".",".","."],
    ["6",".",".","1","9","5",".",".","."],
    [".","9","8",".",".",".",".","6","."],
    ["8",".",".",".","6",".",".",".","3"],
    ["4",".",".","8",".","3",".",".","1"],
    ["7",".",".",".","2",".",".",".","6"],
    [".","6",".",".",".",".","2","8","."],
    [".",".",".","4","1","9",".",".","5"],
    [".",".",".",".","8",".",".","7","9"]
]
print("Test 1:", sol.isValidSudoku(board1))  # Expected: True

#  Test 2: Invalid due to duplicate in 3x3 box
board2 = [
    ["8","3",".",".","7",".",".",".","."],
    ["6",".",".","1","9","5",".",".","."],
    [".","9","8",".",".",".",".","6","."],
    ["8",".",".",".","6",".",".",".","3"],
    ["4",".",".","8",".","3",".",".","1"],
    ["7",".",".",".","2",".",".",".","6"],
    [".","6",".",".",".",".","2","8","."],
    [".",".",".","4","1","9",".",".","5"],
    [".",".",".",".","8",".",".","7","9"]
]
print("Test 2:", sol.isValidSudoku(board2))  # Expected: False

#  Test 3: Invalid due to duplicate in column
board3 = [
    [".",".",".",".","5",".",".","1","."],
    [".","4",".","3",".",".",".",".","."],
    [".",".",".",".",".","3",".",".","1"],
    ["8",".",".",".",".",".",".","2","."],
    [".",".","2",".","7",".",".",".","."],
    [".","1","5",".",".",".",".",".","."],
    [".",".",".",".",".","2",".",".","."],
    [".","2",".","9",".",".",".",".","."],
    [".",".","4",".",".",".",".",".","."]
]
print("Test 3:", sol.isValidSudoku(board3))  # Expected: False
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly →

instant feedback.

Example Walkthrough

We'll walk through **Test 1** step by step.

Initial state: - `rows = [set(), set(), ..., set()]` (9 empty sets) - Same for `cols` and `boxes`.

Step 1: `i=0, j=0`, `val = "5"` - Not `'.'`, so proceed. - `box_idx = (0//3)*3 + (0//3) = 0` - Check: `"5"` not in `rows[0]`, `cols[0]`, or `boxes[0]` → OK. - Add `"5"` to all three → `rows[0] = {"5"}`, etc.

Step 2: `i=0, j=1`, `val = "3"` - `box_idx = 0` - `"3"` not seen in row 0, col 1, or box 0 → add it.

Step 3: `i=0, j=2`, `val = "."` → skip.

…

Step 10: `i=1, j=0`, `val = "6"` - `box_idx = (1//3)*3 + 0 = 0*3 + 0 = 0` - Check box 0: currently has `{"5","3","8","9"}` from previous rows? - Wait! Actually, we haven't processed row 3 yet. At this point (i=1), box 0 only has `"5","3"` from row 0 and now `"6"` from row 1 → no conflict.

Continue until all 81 cells are checked.

Final state: No duplicates found → return `True`.

Key insight: Each digit is checked **exactly once** against its row, column, and box using hash sets for O(1) lookups.

Complexity Analysis

- **Time Complexity**: `O(1)`

  The board is always `9 x 9` → fixed 81 cells. Each cell is processed once with O(1) set operations. So technically constant time. If generalized to `n x n`, it would be `O(n²)`.

- **Space Complexity**: `O(1)`

We use 27 sets (9 rows + 9 cols + 9 boxes), each holding at most 9 digits. Total space is bounded by a constant (27×9). Thus, O(1).

2. Set Matrix Zeroes

Pattern: Arrays & Hashing (In-Place Modification)

Problem Statement

Given an `m x n` integer matrix `matrix`, if an element is `0`, set its entire row and column to `0`'s.
You must do it **in place**.

Sample Input & Output

```
Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
Output: [[1,0,1],[0,0,0],[1,0,1]]
Explanation: The zero at (1,1) zeroes out row 1 and column 1.
```

```
Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]
Explanation: Zeros in first row (col 0 and 3) zero out cols 0 & 3;
             also zero out entire first row.
```

```
Input: matrix = [[1]]
Output: [[1]]
Explanation: No zeros → no change (edge: 1x1 matrix).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        # STEP 1: Initialize structures
        #   - Use first row and first col as markers.
        #   - Track separately if first row/col originally had zeros.
        m, n = len(matrix), len(matrix[0])
        first_row_has_zero = any(matrix[0][j] == 0 for j in range(n))
        first_col_has_zero = any(matrix[i][0] == 0 for i in range(m))

        # STEP 2: Main loop / recursion
        #   - Scan inner matrix (from [1][1] onward).
        #   - If cell is 0, mark its row head and col head as 0.
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][j] == 0:
                    matrix[i][0] = 0
                    matrix[0][j] = 0

        # STEP 3: Update state / bookkeeping
        #   - Use markers in first row/col to zero out inner cells.
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0

        # STEP 4: Return result
        #   - Handle edge cases: zero out first row/col if needed.
        if first_row_has_zero:
            for j in range(n):
                matrix[0][j] = 0
        if first_col_has_zero:
            for i in range(m):
                matrix[i][0] = 0

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    mat1 = [[1,1,1],[1,0,1],[1,1,1]]
```

```python
    sol.setZeroes(mat1)
    print("Test 1:", mat1)
    # Expected: [[1,0,1],[0,0,0],[1,0,1]]

    #  Test 2: Edge case
    mat2 = [[1]]
    sol.setZeroes(mat2)
    print("Test 2:", mat2)
    # Expected: [[1]]

    #  Test 3: Tricky/negative
    mat3 = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
    sol.setZeroes(mat3)
    print("Test 3:", mat3)
    # Expected: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**:
`matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]`

Initial state:
- `m = 3`, `n = 4`
- `first_row_has_zero = True` (because `matrix[0][0] == 0` and `matrix[0][3] == 0`)
- `first_col_has_zero = True` (because `matrix[0][0] == 0`)

Step 1: Mark inner zeros
Loop over `i=1..2`, `j=1..3`:
- At `(1,1)`: 4 → no mark
- At `(1,2)`: 5 → no mark
- At `(1,3)`: 2 → no mark
- At `(2,1)`: 3 → no mark
- At `(2,2)`: 1 → no mark
- At `(2,3)`: 5 → no mark
→ No new markers added (only original zeros in row 0).

Step 2: Zero out inner cells using markers
Check each inner cell:

- For `i=1, j=1`: `matrix[1][0]=3  0`, `matrix[0][1]=1  0` → keep 4
- `j=2`: same → keep 5
- `j=3`: `matrix[0][3]=0` → set `matrix[1][3] = 0`
- For `i=2, j=1`: `matrix[0][1]=1`, `matrix[2][0]=1` → keep 3
- `j=2`: keep 1
- `j=3`: `matrix[0][3]=0` → set `matrix[2][3] = 0`

Now matrix looks like:
`[[0,1,2,0], [3,4,5,0], [1,3,1,0]]`

Step 3: Zero out first row and first column
- `first_row_has_zero = True` → set entire row 0 to 0
- `first_col_has_zero = True` → set `matrix[0][0]`, `matrix[1][0]`, `matrix[2][0]` to 0

Final matrix:
`[[0,0,0,0], [0,4,5,0], [0,3,1,0]]`

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  We scan the matrix a constant number of times (3 full passes):
  – once to check first row/col,
  – once to mark,
  – once to apply zeros,
  – and two partial passes for first row/col cleanup.
  All are linear in total elements.

- **Space Complexity**: `O(1)`

  We use only a few boolean flags (`first_row_has_zero`, `first_col_has_zero`). No extra arrays or hash maps — all marking is done **in-place** using the matrix's own first row and column.

3. Spiral Matrix

Pattern: Matrix Traversal (Simulation)

Problem Statement

Given an `m x n` matrix, return all elements of the matrix in spiral order.

Sample Input & Output

```
Input: [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]
Explanation: Traverse top row → right column → bottom row (rev)
→ left column (rev), then repeat inward.
```

```
Input: [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]
Explanation: Spiral continues layer by layer until all cells visited.
```

```
Input: [[1]]
Output: [1]
Explanation: Single-element edge case.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        # STEP 1: Initialize boundaries and result list
        #   - top, bottom, left, right define current layer
        #   - result collects elements in spiral order
        if not matrix or not matrix[0]:
            return []

        top, bottom = 0, len(matrix) - 1
        left, right = 0, len(matrix[0]) - 1
```

```python
        result = []

        # STEP 2: Main loop - traverse while boundaries valid
        # - Invariant: [top, bottom] and [left, right] form a valid submatrix
        while top <= bottom and left <= right:
            # Traverse top row (left → right)
            for col in range(left, right + 1):
                result.append(matrix[top][col])
            top += 1  # Move top boundary down

            # Traverse right column (top → bottom)
            for row in range(top, bottom + 1):
                result.append(matrix[row][right])
            right -= 1  # Move right boundary left

            # Traverse bottom row (right → left), if row exists
            if top <= bottom:
                for col in range(right, left - 1, -1):
                    result.append(matrix[bottom][col])
                bottom -= 1  # Move bottom boundary up

            # Traverse left column (bottom → top), if column exists
            if left <= right:
                for row in range(bottom, top - 1, -1):
                    result.append(matrix[row][left])
                left += 1  # Move left boundary right

        # STEP 4: Return result
        #   - All elements collected in spiral order
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.spiralOrder([[1,2,3],[4,5,6],[7,8,9]]) == [1,2,3,6,9,8,7,4,5]

    #   Test 2: Edge case - single element
    assert sol.spiralOrder([[1]]) == [1]

    #   Test 3: Tricky/negative - wide rectangle
```

```python
assert (sol.spiralOrder([[1,2,3,4],[5,6,7,8],[9,10,11,12]]) ==
        [1,2,3,4,8,12,11,10,9,5,6,7])

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `spiralOrder([[1,2,3],[4,5,6],[7,8,9]])` step by step.

Initial state:
- `matrix = [[1,2,3],[4,5,6],[7,8,9]]`
- `top = 0`, `bottom = 2`
- `left = 0`, `right = 2`
- `result = []`

Step 1: Top row (left → right)
Loop: `col` from 0 to 2
- Append `matrix[0][0] = 1` → `result = [1]`
- Append `matrix[0][1] = 2` → `result = [1,2]`
- Append `matrix[0][2] = 3` → `result = [1,2,3]`
Then: `top += 1` → `top = 1`

State: `top=1`, `bottom=2`, `left=0`, `right=2`, `result=[1,2,3]`

Step 2: Right column (top → bottom)
Loop: `row` from 1 to 2
- Append `matrix[1][2] = 6` → `result = [1,2,3,6]`
- Append `matrix[2][2] = 9` → `result = [1,2,3,6,9]`
Then: `right -= 1` → `right = 1`

State: `top=1`, `bottom=2`, `left=0`, `right=1`, `result=[1,2,3,6,9]`

Step 3: Bottom row (right → left)
Check: `top (1) <= bottom (2)` →
Loop: `col` from 1 down to 0
- Append `matrix[2][1] = 8` → `result = [1,2,3,6,9,8]`
- Append `matrix[2][0] = 7` → `result = [1,2,3,6,9,8,7]`
Then: `bottom -= 1` → `bottom = 1`

State: `top=1`, `bottom=1`, `left=0`, `right=1`, `result=[1,2,3,6,9,8,7]`

Step 4: Left column (bottom → top)
Check: `left (0) <= right (1)` →
Loop: `row` from 1 down to 1 (only one iteration)
- Append `matrix[1][0] = 4` → `result = [1,2,3,6,9,8,7,4]`
Then: `left += 1` → `left = 1`

State: `top=1`, `bottom=1`, `left=1`, `right=1`, `result=[1,2,3,6,9,8,7,4]`

Next loop iteration:
`top (1) <= bottom (1)` and `left (1) <= right (1)` → continue

Step 5: Top row again
Loop: `col` from 1 to 1
- Append `matrix[1][1] = 5` → `result = [1,2,3,6,9,8,7,4,5]`
Then: `top += 1` → `top = 2`

Now: `top (2) > bottom (1)` → loop ends.

Final result: `[1,2,3,6,9,8,7,4,5]`

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  Every element is visited exactly once. Total elements = `m * n`.

- **Space Complexity**: `O(1)` (excluding output)

  Only a few boundary variables (`top`, `bottom`, `left`, `right`) are used. The output list is not counted toward auxiliary space.

4. Rotate Image

Pattern: Matrix Manipulation (In-Place Transformation)

Problem Statement

You are given an `n x n` 2D matrix representing an image. Rotate the image by **90 degrees (clockwise)**.
You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Sample Input & Output

```
Input: [[1,2,3],
        [4,5,6],
        [7,8,9]]
Output: [[7,4,1],
         [8,5,2],
         [9,6,3]]
Explanation: The matrix is rotated 90° clockwise in place.
```

```
Input: [[5,1,9,11],
        [2,4,8,10],
        [13,3,6,7],
        [15,14,12,16]]
Output: [[15,13,2,5],
         [14,3,4,1],
         [12,6,8,9],
         [16,7,10,11]]
Explanation: 4x4 matrix rotated correctly.
```

```
Input: [[1]]
Output: [[1]]
Explanation: Single-element matrix remains unchanged.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        # STEP 1: Initialize structures
        #   - n is the size of the square matrix
        #   - We'll perform in-place rotation using layer-by-layer
        #     swaps (like peeling an onion)
        n = len(matrix)

        # STEP 2: Main loop / recursion
        #   - Loop over layers from outer to inner
        #   - For an n x n matrix, there are n // 2 layers
        for layer in range(n // 2):
            # Define first and last index of current layer
            first = layer
            last = n - 1 - layer

            # STEP 3: Update state / bookkeeping
            #   - For each element in the current layer's top row
            #     (excluding the last, which is handled by rotation),
```

```
            #     perform a 4-way swap
            for i in range(first, last):
                offset = i - first

                # Save top element
                top = matrix[first][i]

                # Move left → top
                matrix[first][i] = matrix[last - offset][first]

                # Move bottom → left
                matrix[last - offset][first] = \
                    matrix[last][last - offset]

                # Move right → bottom
                matrix[last][last - offset] = \
                    matrix[i][last]

                # Move saved top → right
                matrix[i][last] = top

        # STEP 4: Return result
        #   - Nothing to return; matrix is modified in-place

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case (3x3)
    mat1 = [[1,2,3],[4,5,6],[7,8,9]]
    sol.rotate(mat1)
    expected1 = [[7,4,1],[8,5,2],[9,6,3]]
    assert mat1 == expected1, f"Test 1 failed: got {mat1}"
    print("  Test 1 passed")

    #   Test 2: Edge case (1x1)
    mat2 = [[1]]
    sol.rotate(mat2)
    expected2 = [[1]]
    assert mat2 == expected2, f"Test 2 failed: got {mat2}"
    print("  Test 2 passed")
```

```python
#   Test 3: Tricky case (4x4)
mat3 = [[5,1,9,11],
        [2,4,8,10],
        [13,3,6,7],
        [15,14,12,16]]
sol.rotate(mat3)
expected3 = [[15,13,2,5],
             [14,3,4,1],
             [12,6,8,9],
             [16,7,10,11]]
assert mat3 == expected3, f"Test 3 failed: got {mat3}"
print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `[[1,2,3],[4,5,6],[7,8,9]]`.

Initial matrix:

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

Step-by-step execution:

1. `n = 3` → `n // 2 = 1`, so **1 layer** (`layer = 0`).
2. `first = 0`, `last = 2`.
3. Loop `i` from `0` to `1` (since `range(0, 2)`).

First iteration (`i = 0`): - `offset = 0 - 0 = 0` - `top = matrix[0][0] = 1`

Now perform 4-way swap:

- **Left → Top**:
  `matrix[0][0] = matrix[2 - 0][0] = matrix[2][0] = 7`
  → Row 0 becomes `[7, 2, 3]`
- **Bottom → Left**:
  `matrix[2][0] = matrix[2][2 - 0] = matrix[2][2] = 9`
  → Row 2 becomes `[9, 8, 9]`
- **Right → Bottom**:
  `matrix[2][2] = matrix[0][2] = 3`
  → Row 2 becomes `[9, 8, 3]`
- **Top (saved) → Right**:
  `matrix[0][2] = top = 1`
  → Row 0 becomes `[7, 2, 1]`

Matrix now:

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

Second iteration (`i = 1`): - `offset = 1 - 0 = 1` - `top = matrix[0][1] = 2`

Swaps:

- **Left → Top**:
  `matrix[0][1] = matrix[2 - 1][0] = matrix[1][0] = 4`
  → Row 0: `[7, 4, 1]`
- **Bottom → Left**:
  `matrix[1][0] = matrix[2][2 - 1] = matrix[2][1] = 8`
  → Row 1: `[8, 5, 6]`
- **Right → Bottom**:
  `matrix[2][1] = matrix[1][2] = 6`
  → Row 2: `[9, 6, 3]`
- **Top → Right**:
  `matrix[1][2] = top = 2`
  → Row 1: `[8, 5, 2]`

Final matrix:

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

Matches expected output!

Key insight: Each layer is rotated by moving elements in groups of 4 — top ← left ← bottom ← right ← top.

Complexity Analysis

- **Time Complexity**: $O(n^2)$

  We visit each element exactly once. The outer loop runs `n // 2` times, and the inner loop runs up to `n - 1` times per layer. Total operations $n^2 / 4 * 4 = n^2$.

- **Space Complexity**: `O(1)`

  Only a constant amount of extra space is used (`top`, `offset`, loop indices). The rotation is done **in-place**.

5. Sudoku Solver

Pattern: Backtracking

Problem Statement

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:
- Each of the digits `1-9` must occur exactly once in each row.
- Each of the digits `1-9` must occur exactly once in each column.
- Each of the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

The `'.'` character indicates empty cells.

The input board is guaranteed to be solvable. Modify the board **in-place**.

Sample Input & Output

```
Input: board = [
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: board filled with valid digits (in-place)
Explanation: The puzzle has a unique valid solution that
satisfies all Sudoku rules.
```

```
Input: board = [["."]*9 for _ in range(9)]
Output: A fully filled valid Sudoku grid
Explanation: Even an empty board is solvable; backtracking will
find one valid configuration.
```

```
Input: board = [
  ["1",".",".",".",".",".",".",".","."],
  [".","2",".",".",".",".",".",".","."],
  [".",".","3",".",".",".",".",".","."],
  [".",".",".","4",".",".",".",".","."],
  [".",".",".",".","5",".",".",".","."],
  [".",".",".",".",".","6",".",".","."],
  [".",".",".",".",".",".","7",".","."],
  [".",".",".",".",".",".",".","8","."],
  [".",".",".",".",".",".",".",".","9"]
]
Output: Completed valid Sudoku
Explanation: Diagonal initial values still allow a unique solution.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place.
        """
        # STEP 1: Initialize structures
        #   - rows[i]: set of digits in row i
        #   - cols[j]: set of digits in col j
        #   - boxes[box_id]: set of digits in 3x3 box
        rows = [set() for _ in range(9)]
        cols = [set() for _ in range(9)]
        boxes = [set() for _ in range(9)]

        # Pre-fill known digits
        for i in range(9):
            for j in range(9):
                if board[i][j] != '.':
                    num = board[i][j]
                    rows[i].add(num)
                    cols[j].add(num)
                    box_id = (i // 3) * 3 + (j // 3)
                    boxes[box_id].add(num)

        # STEP 2: Main backtracking function
        #   - Tries digits 1-9 in empty cells
        #   - Backtracks if conflict arises
        def backtrack(i, j):
            # Base: reached end of board
            if i == 9:
                return True
            # Move to next cell
            next_i, next_j = (i, j + 1) if j < 8 else (i + 1, 0)

            # Skip filled cells
            if board[i][j] != '.':
                return backtrack(next_i, next_j)

            # Try digits '1' to '9'
```

```python
            box_id = (i // 3) * 3 + (j // 3)
            for d in '123456789':
                if d in rows[i] or d in cols[j] or d in boxes[box_id]:
                    continue  # conflict → skip

                # STEP 3: Update state / bookkeeping
                board[i][j] = d
                rows[i].add(d)
                cols[j].add(d)
                boxes[box_id].add(d)

                # Recurse to next cell
                if backtrack(next_i, next_j):
                    return True

                # Undo changes (backtrack)
                board[i][j] = '.'
                rows[i].remove(d)
                cols[j].remove(d)
                boxes[box_id].remove(d)

            # STEP 4: Return result
            #   - No digit worked → signal failure to caller
            return False

        # Start backtracking from top-left
        backtrack(0, 0)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    board1 = [
        ["5","3",".",".","7",".",".",".","."],
        ["6",".",".","1","9","5",".",".","."],
        [".","9","8",".",".",".",".","6","."],
        ["8",".",".",".","6",".",".",".","3"],
        ["4",".",".","8",".","3",".",".","1"],
        ["7",".",".",".","2",".",".",".","6"],
        [".","6",".",".",".",".","2","8","."],
        [".",".",".","4","1","9",".",".","5"],
```

```
        [".",".",".",".","8",".",".","7","9"]
    ]
    sol.solveSudoku(board1)
    assert all('.' not in row for row in board1), "Test 1 failed"
    print("  Test 1 passed: Normal Sudoku solved")

    #   Test 2: Edge case - empty board
    board2 = [["."]*9 for _ in range(9)]
    sol.solveSudoku(board2)
    assert all('.' not in row for row in board2), "Test 2 failed"
    print("  Test 2 passed: Empty board solved")

    #   Test 3: Tricky/negative - diagonal start
    board3 = [
        ["1",".",".",".",".",".",".",".","."],
        [".","2",".",".",".",".",".",".","."],
        [".",".","3",".",".",".",".",".","."],
        [".",".",".","4",".",".",".",".","."],
        [".",".",".",".","5",".",".",".","."],
        [".",".",".",".",".","6",".",".","."],
        [".",".",".",".",".",".","7",".","."],
        [".",".",".",".",".",".",".","8","."],
        [".",".",".",".",".",".",".",".","9"]
    ]
    sol.solveSudoku(board3)
    assert all('.' not in row for row in board3), "Test 3 failed"
    print("  Test 3 passed: Diagonal-start Sudoku solved")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** at a high level (full step-by-step would be thousands of steps due to backtracking):

1. **Initialization**:
    - `rows[0] = {'5','3','7'}`, `cols[0] = {'5','6','8','4','7'}`, etc.

  - All pre-filled digits are recorded in `rows`, `cols`, `boxes`.

2. **Start at (0,2)** — first empty cell.
   - Tries `'1'`: not in row 0, col 2, or box 0 → place it.
   - Proceeds to next empty cell.
3. **Later, a conflict arises** (e.g., no digit fits at some cell):
   - Backtrack: undo last placement, try next digit.
   - This repeats until a valid path fills the board.
4. **Eventually**, a full assignment satisfies all constraints.
   - `backtrack` returns `True` up the call stack.
   - Original `board` is modified in-place with solution.

Key Insight: Backtracking explores possibilities **depth-first**, pruning invalid paths early using the sets (`rows`, `cols`, `boxes`) for O(1) conflict checks.

Complexity Analysis

- **Time Complexity**: `O(9^(n))` where `n` = number of empty cells (worst case)

  > In worst case, each empty cell tries up to 9 digits. With ~50–60 empties, this is exponential — but pruning via constraint sets makes it feasible for standard Sudoku.

- **Space Complexity**: `O(1)`

  > We use 3 fixed-size structures (`rows`, `cols`, `boxes`) of size 9 each. Recursion depth 81 (cells), so stack space is bounded by constant. Input board is modified in-place.