Previous Next

37. Sudoku Solver

March 20, 2019 | 29.2K views

Average Rating: 4.60 (25 votes)

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

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

Empty cells are indicated by the character `'.'`.

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

A sudoku puzzle...

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

...and its solution numbers marked in red.

Note:

- The given board contain only digits `1-9` and the character `'.'`.
- You may assume that the given Sudoku puzzle will have a single unique solution.
- The given board size is always `9x9`.

Solution

Approach 0: Brute Force

The first idea is to use brut-force to generate all possible ways to fill the cells with numbers from `1` to `9`, and then check them to keep the solution only. That means 9^{81} operations to do, where 9 is a number of available digits and 81 is a number of cells to fill. Hence we're forced to think further how to optimize.

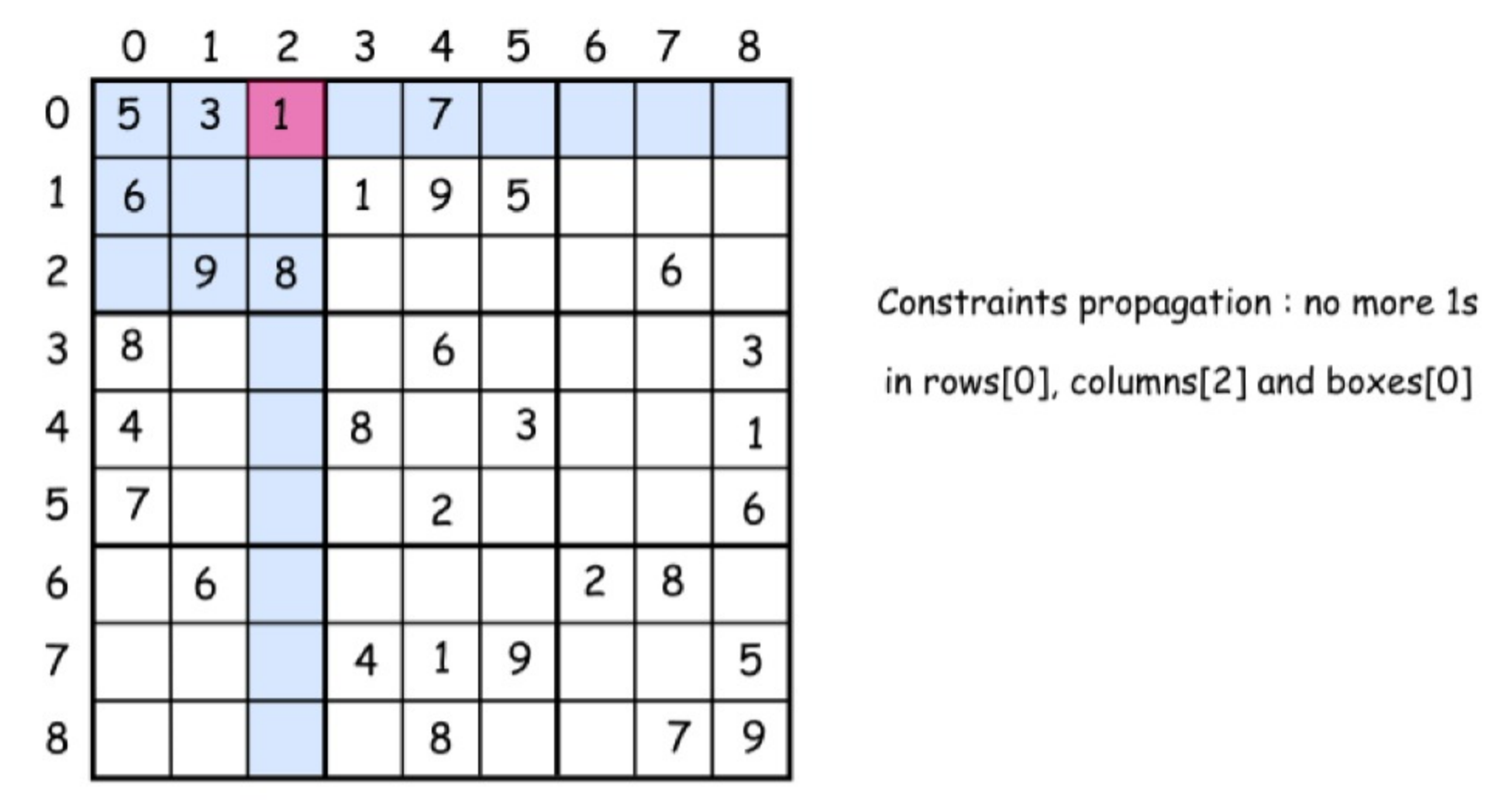
Approach 1: Backtracking

Conceptions to use

There are two programming conceptions here which could help.

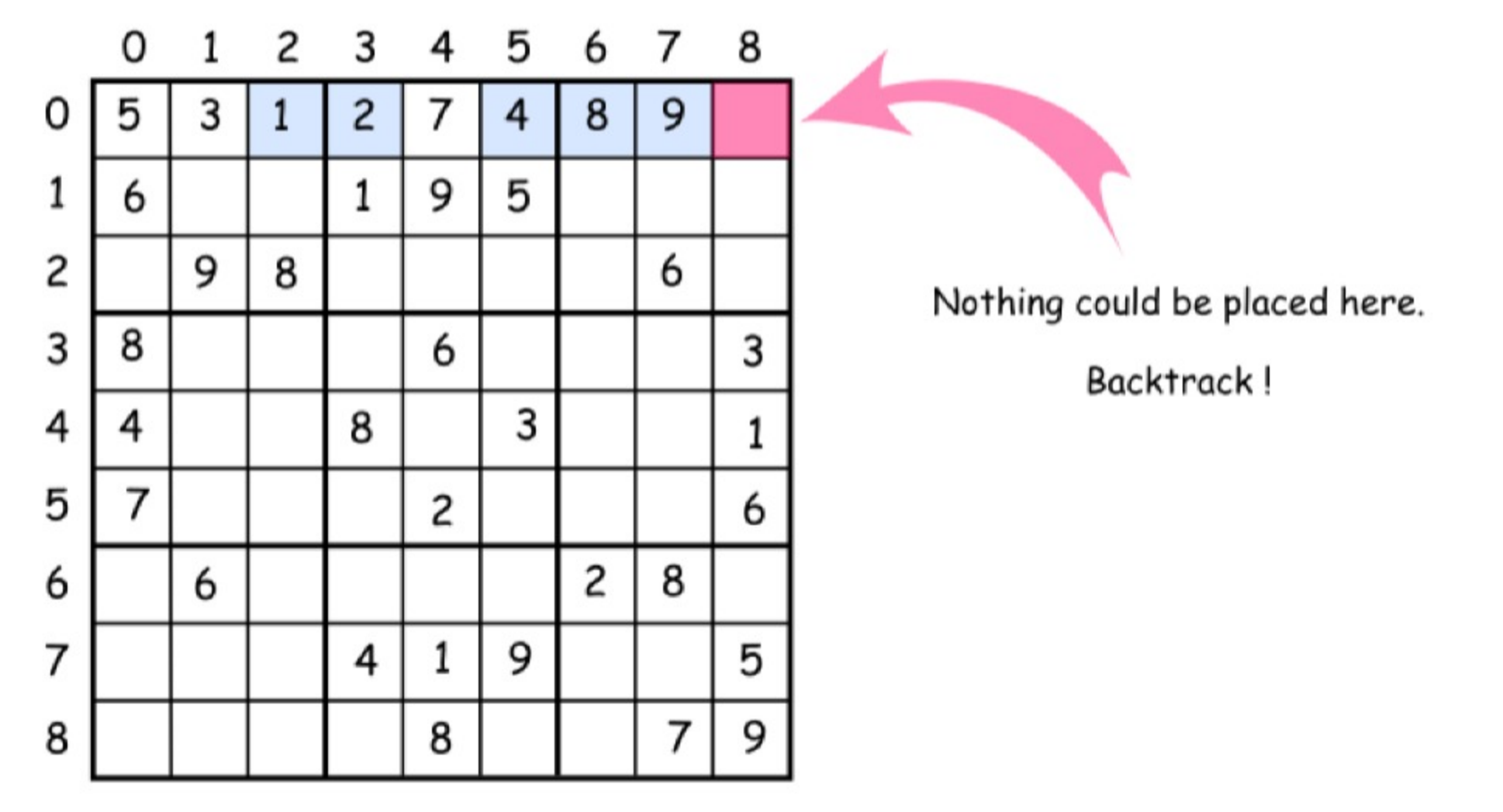
> The first one is called *constrained programming.*

That basically means to put restrictions after each number placement. One puts a number on the board and that immediately excludes this number from further usage in the current *row, column* and *sub-box*. That propagates *constraints* and helps to reduce the number of combinations to consider.



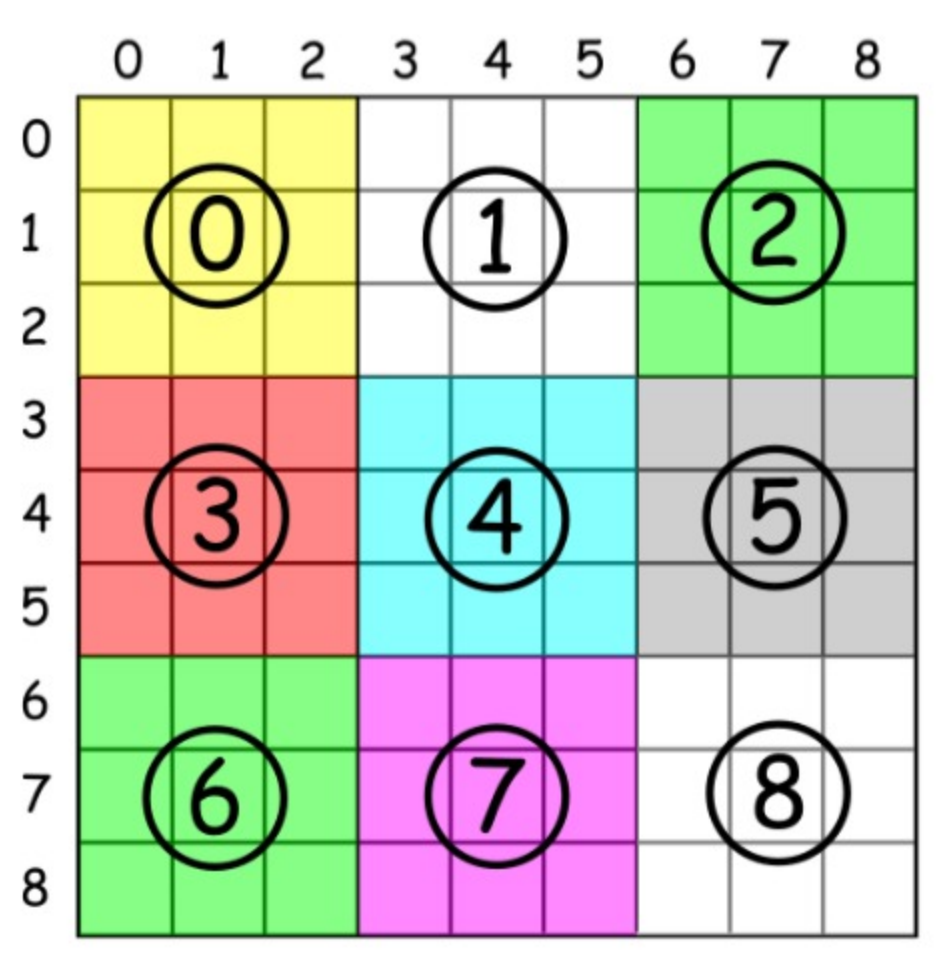
> The second one called *backtracking.*

Let's imagine that one has already managed to put several numbers on the board. But the combination chosen is not the optimal one and there is no way to place the further numbers. What to do? *To backtrack.* That means to come back, to change the previously placed number and try to proceed again. If that would not work either, *backtrack* again.



How to enumerate sub-boxes

> One tip to enumerate sub-boxes: let's use `box_index = (row / 3) * 3 + column / 3` where `/` is an integer division.



Algorithm

Now everything is ready to write down the backtrack function `backtrack(row = 0, col = 0)`.

- Start from the upper left cell `row = 0, col = 0`. Proceed till the first free cell.
- Iterate over the numbers from `1` to `9` and try to put each number `d` in the `(row, col)` cell.
  - If number `d` is not yet in the current row, column and box :
    - Place the `d` in a `(row, col)` cell.
    - Write down that `d` is now present in the current row, column and box.
    - If we're on the last cell `row == 8, col == 8` :
      - That means that we've solved the sudoku.
    - Else
      - Proceed to place further numbers.
    - Backtrack if the solution is not yet here : remove the last number from the `(row, col)` cell.

Implementation

Java Python Copy

```
from collections import defaultdict
class Solution:
    def solveSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: void Do not return anything, modify board in-place instead.
        """
        def could_place(d, row, col):
            """
            Check if one could place a number d in (row, col) cell
            """
            return not (d in rows[row] or d in columns[col] or \
                    d in boxes[box_index(row, col)])

        def place_number(d, row, col):
            """
            Place a number d in (row, col) cell
            """
            rows[row][d] += 1
            columns[col][d] += 1
            boxes[box_index(row, col)][d] += 1
            board[row][col] = str(d)

        def remove_number(d, row, col):
            """
            Remove a number which didn't lead
            to a solution
```

Complexity Analysis

- Time complexity is constant here since the board size is fixed and there is no N-parameter to measure. Though let's discuss the number of operations needed : $(9!)^9$. Let's consider one row, i.e. not more than 9 cells to fill. There are not more than 9 possibilities for the first number to put, not more than 9×8 for the second one, not more than $9 \times 8 \times 7$ for the third one etc. In total that results in not more than $9!$ possibilities for a just one row, that means not more than $(9!)^9$ operations in total. Let's compare:
  - $9^{81} = 196627050475552913618075908526912116283103450944214766927315415537966391196809$ for the brute force,
  - and $(9!)^9 = 109110688415571316480344899355894085582848000000000$ for the standard backtracking, i.e. the number of operations is reduced in 10^{27} times !
- Space complexity : the board size is fixed, and the space is used to store board, rows, columns and boxes structures, each contains `81` elements.

Rate this article:

Previous Next

Comments: 11 Sort By

Type comment here... (Markdown is supported)

Preview Post

haoyangfan ★ 897 November 13, 2019 11:42 AM

Share a simplified version of backtrack solution:

```
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
```

Read More

6 Share Reply

SHOW 1 REPLY

LPEO ★ 3 May 31, 2020 11:27 AM Report

For simplicity during an interview

Time Complexity: O(9^(m * n))
Space Complexity: O(m*n)

3 Share Reply

ShaneTsui ★ 109 April 8, 2020 12:38 AM

There's no need creating extra data structure as rows, columns and boxes. It's good for engineering, but not practical for interview. Here's a succint version of backtracing

```
class Solution:
```

Read More

1 Share Reply

lenchen1112 ★ 972 January 18, 2020 3:50 PM Report

Clean Python3, dfs by row-based.

```
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
```

Read More

1 Share Reply

zhang-peter ★ 26 March 25, 2019 7:50 PM

i have three times faster java code(3 ms), what i do is first fill the cells which can only fill one number:

```
class Solution {
    // box size
```

Read More

1 Share Reply

SHOW 1 REPLY

brokechigga ★ 4 June 15, 2019 4:02 AM Report

why this code only work in python 3 not 2?

1 Share Reply

SHOW 2 REPLIES

zhang-peter ★ 26 March 25, 2019 6:49 PM

i have three times faster python code(80 ms), what i do is first fill the cells which can only fill one number:

```
from collections import defaultdict
```

Read More

0 Share Reply

SHOW 1 REPLY

hjy111 ★ 8 March 2, 2020 8:17 AM Report

Why we need backtrack(0,0) at the end of function solveSudoku ?

0 Share Reply

SHOW 1 REPLY

rextor ★ 0 February 29, 2020 9:42 PM

```
/**
 Simpler implementation
```

Read More

0 Share Reply

ywt2018 ★ 0 August 30, 2019 11:14 PM Report

it seems that placeNumber can change the orignal placed chars. agree?

0 Share Reply

SHOW 1 REPLY

‹ 1 2 ›