Sudoku Solver

Problem Statement:

Given a 9x9 incomplete sudoku, solve it such that it becomes valid sudoku. Valid sudoku has the following properties.

1. All the rows should be filled with numbers(1 - 9) exactly once.
2. All the columns should be filled with numbers(1 - 9) exactly once.
3. Each 3x3 submatrix should be filled with numbers(1 - 9) exactly once.

Note: Character **'.'** indicates empty cell.

Example:

`Input:`

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

`Output:`

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

`Explanation:`
`The empty cells are filled with the possible numbers. There can exist many such arrangements of numbers. The above solution is one of them. Let’s see how we can fill the cells below.`

Solution

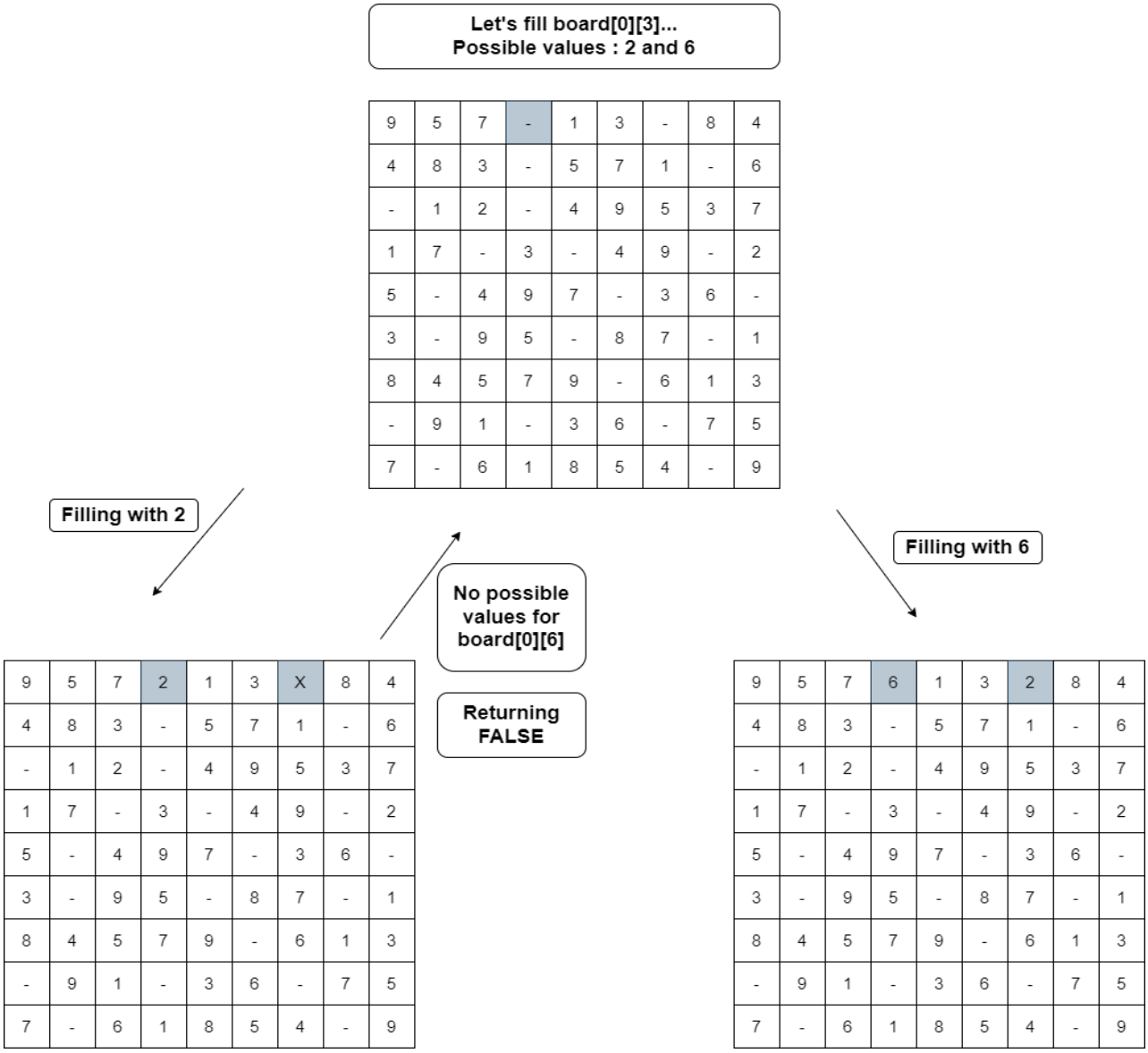
Intuition:

Since we have to fill the empty cells with available possible numbers and we can also have multiple solutions, the main intuition is to try every possible way of filling the empty cells. And the more correct way to try all possible solutions is to use recursion. In each call to the recursive function, we just try all the possible numbers for a particular cell and transfer the updated board to the next recursive call.

Approach:

- Let’s see the step by step approach. Our main recursive function(solve()) is going to just do a plain matrix traversal of the sudoku board. When we find an empty cell, we pause and try to put all available numbers(1 - 9) in that particular empty cell.

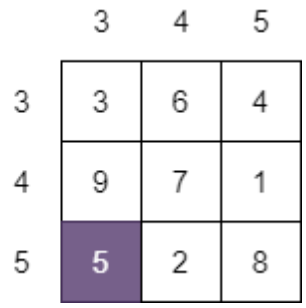
- We need another loop to do that. But wait, we forgot one thing - the board has to satisfy all the conditions, right? So, for that we have another function(isValid()) which will check whether the number we have inserted into that empty cell will not violate any conditions.
- If it is violating, we try with the next number. If it is not, we call the same function recursively, but this time with the updated state of the board. Now, as usual it tries to fill the remaining cells in the board in the same way.
- Now we'll come to the returning values. If at any point we cannot insert any numbers from 1 - 9 in a particular cell, it means the current state of the board is wrong and we need to backtrack. An important point to follow is, we need to return false to let the parent function(which is called this function) know that we cannot fill this way. This will serve as a hint to that function, that it needs to try with the next possible number. Refer to the picture below.



- If a recursive call returns true, we can assume that we found one possible way of filling and we simply do an **early return**.

Validating Board

- Now, let's see how we are validating the sudoku board. After determining a number for a cell(at i'th row, j'th col), we try to check the validity. As we know, a valid sudoku needs to satisfy 3 conditions, we can use three loops. But we can do within a single loop itself. Let's try to understand that.
- We loop from 0 to 8 and check the values - board[i][col](1st condition) and board[row][i] (2nd condition), whether the number is already included. For the 3rd condition - the expression (3 * (row / 3) + i / 3) evaluates to the row numbers of that 3x3 submatrix and the expression (3 * (col / 3) + i % 3) evaluates to the column numbers.
- For eg, if row= 5 and col= 3, the cells visited are



It covers all the cells in the sub-matrix.

Code:

```
import java.util.*;

class Solution {

  public static boolean solveSudoku(char[][] board) {
    for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++) {
        if (board[i][j] == '.') {

          for (char c = '1'; c <= '9'; c++) {
            if (isValid(board, i, j, c)) {
              board[i][j] = c;

              if (solveSudoku(board))
                return true;
              else
                board[i][j] = '.';
            }
          }

          return false;
        }
      }
    }
    return true;
  }

  public static boolean isValid(char[][] board, int row, int col, char c) {
    for (int i = 0; i < 9; i++) {
      if (board[i][col] == c)
        return false;

      if (board[row][i] == c)
        return false;

      if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
```

```
        return false;
    }
    return true;
  }

  public static void main(String[] args) {

   char[][] board= {
                    {'9', '5', '7', '.', '1', '3', '.', '8', '4'},
                    {'4', '8', '3', '.', '5', '7', '1', '.', '6'},
                    {'.', '1', '2', '.', '4', '9', '5', '3', '7'},
                    {'1', '7', '.', '3', '.', '4', '9', '.', '2'},
                    {'5', '.', '4', '9', '7', '.', '3', '6', '.'},
                    {'3', '.', '9', '5', '.', '8', '7', '.', '1'},
                    {'8', '4', '5', '7', '9', '.', '6', '1', '3'},
                    {'.', '9', '1', '.', '3', '6', '.', '7', '5'},
                    {'7', '.', '6', '1', '8', '5', '4', '.', '9'}
                    };
    solveSudoku(board);

    for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++)
        System.out.print(board[i][j] + " ");
      System.out.println();
    }
  }
}
```

Output:

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

Time Complexity: O(9(n ^ 2)), in the worst case, for each cell in the n2 board, we have 9 possible numbers.

Space Complexity: O(1), since we are refilling the given board itself, there is no extra space required, so constant space complexity.

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.

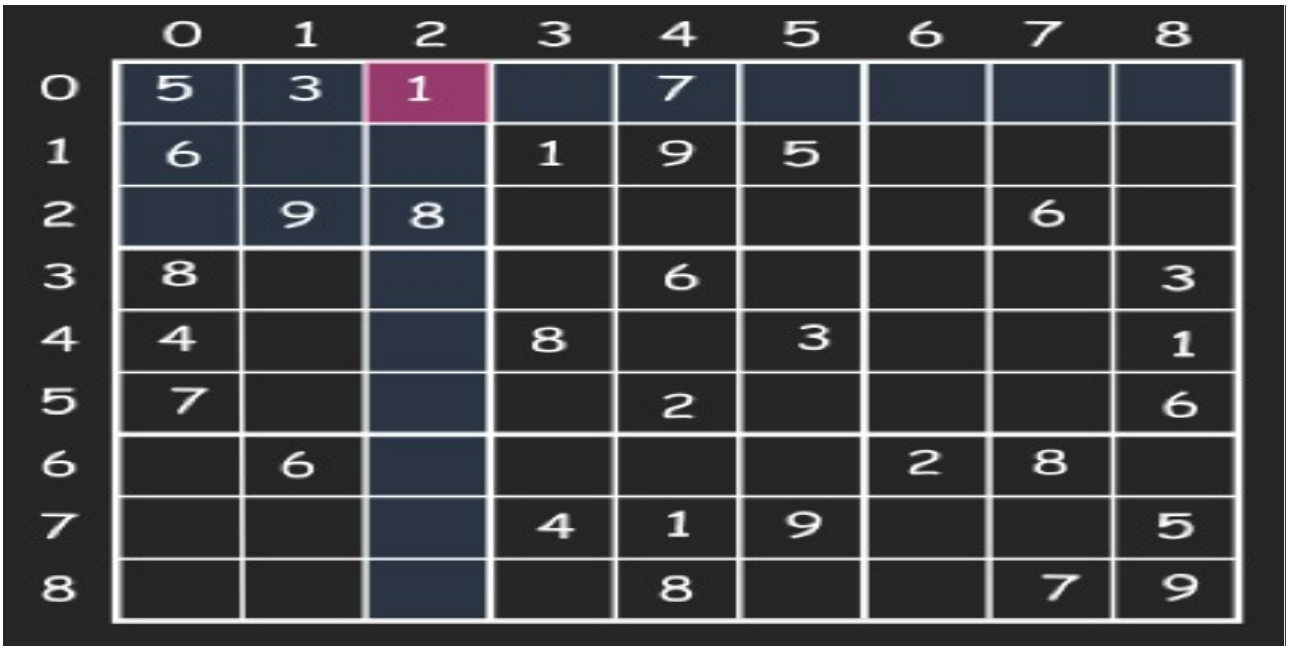
Approach 1: Backtracking

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.

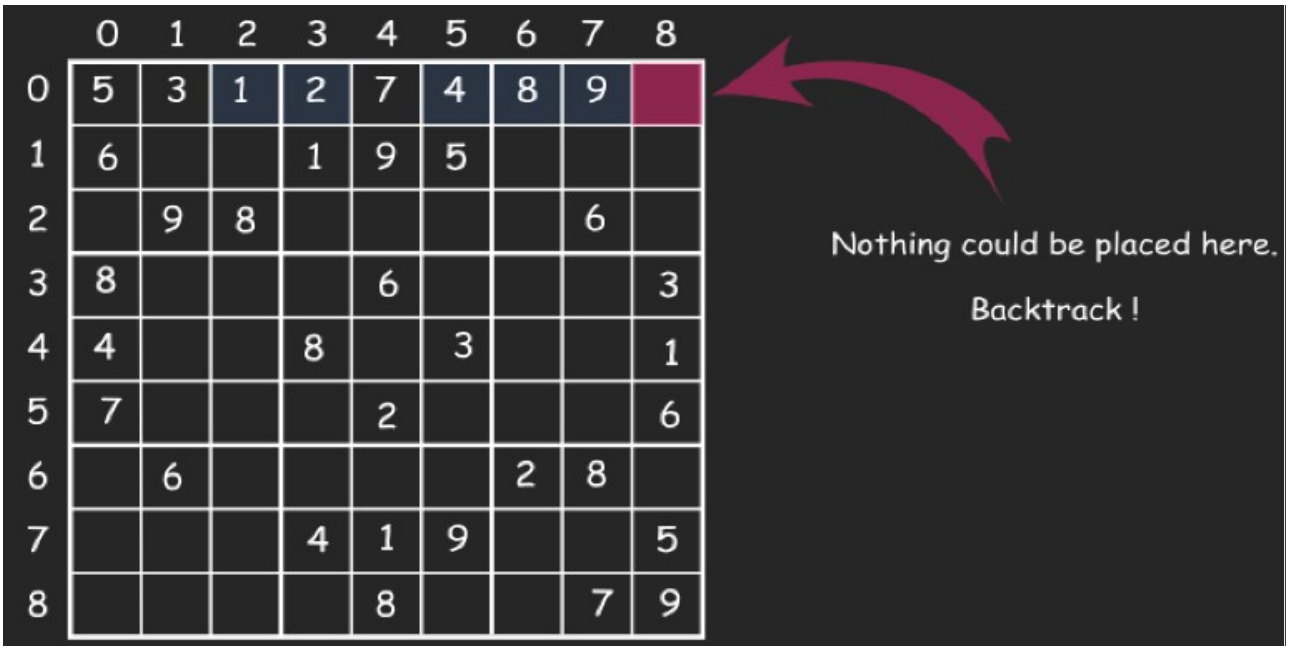


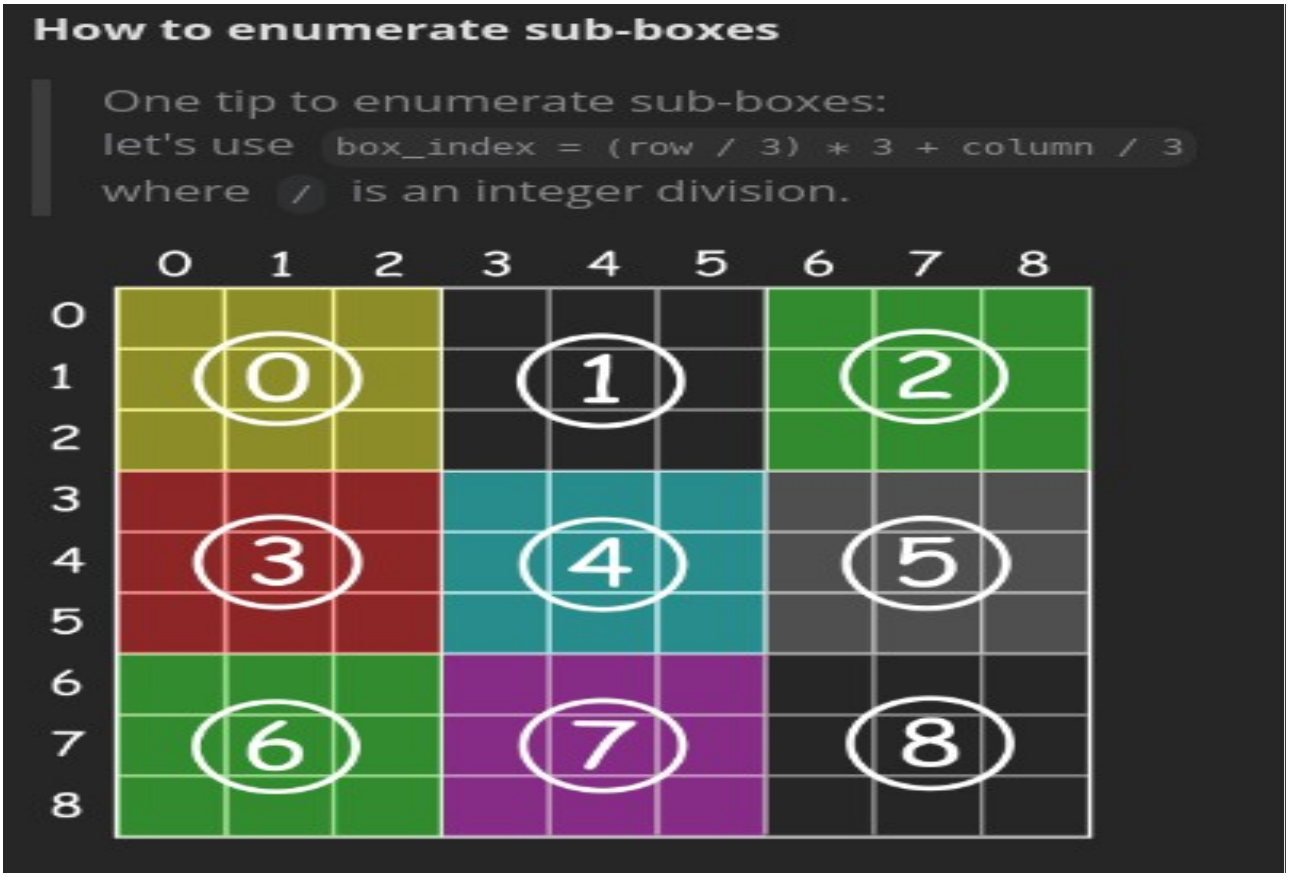
	0	1	2	3	4	5	6	7	8
0	5	3	1		7				
1	6			1	9	5			
2		9	8					6	
3	8				6				3
4	4			8		3			1
5	7				2				6
6		6					2	8	
7				4	1	9			5
8					8			7	9

Constraints propagation: no more `1`'s in `row[0]`, `columns[2]`, and `boxes[0]`

The second one is called `bactracking`

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.


0 1 2 3 4 5 6 7 8
0 5 3 1 2 7 4 8 9
1 6 1 9 5
2 9 8 6
3 8 6 3
4 4 8 3 1
5 7 2 6
6 6 2 8
7 4 1 9 5
8 8 7 9
Nothing could be placed here.
Backtrack !


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.
0 1 2 3 4 5 6 7 8
0
1
2
3
4
5
6
7
8
0
1
2
3
4
5
6
7
8

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:

```
class Solution {
    int n = 3;
    int N = n * n;
    int[][] rows = new int[N][N + 1];
    int[][] columns = new int[N][N + 1];
    int[][] boxes = new int[N][N + 1];
    char[][] board;
    boolean sudokuSolved = false;

    public boolean couldPlace(int d, int row, int col) {
        int idx = (row / n) * n + col / n;
        return rows[row][d] + columns[col][d] + boxes[idx][d] == 0;
    }

    public void placeNumber(int d, int row, int col) {
        int idx = (row / n) * n + col / n;
        rows[row][d]++;
        columns[col][d]++;
        boxes[idx][d]++;
        board[row][col] = (char)(d + '0');
    }

    public void removeNumber(int d, int row, int col) {
        int idx = (row / n) * n + col / n;
        rows[row][d]--;
        columns[col][d]--;
```

```
        boxes[idx][d]--;
        board[row][col] = '.';
    }

    public void placeNextNumbers(int row, int col) {
        if (row == N - 1 && col == N - 1) sudokuSolved = true;
        else if (col == N - 1) backtrack(row + 1, 0);
        else backtrack(row, col + 1);
    }

    public void backtrack(int row, int col) {
        if (board[row][col] == '.') {
            for (int d = 1; d <= 9; d++) {
                if (couldPlace(d, row, col)) {
                    placeNumber(d, row, col);
                    placeNextNumbers(row, col);
                    if (!sudokuSolved) removeNumber(d, row, col);
                }
            }
        } else placeNextNumbers(row, col);
    }

    public void solveSudoku(char[][] board) {
        this.board = board;
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                if (board[i][j] != '.')
placeNumber(Character.getNumericValue(board[i][j]), i, j);
        backtrack(0, 0);
    }
}
```

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:

- $9^{81} = 196627050475552913618075908526912116283103450944214766927315415537$
  for the brute force,
- and $(9!)^9 = 1091106884155713164803448993558940855828480000000$
  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.
