# Question

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 digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

**Example 1:**



**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:** [["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"]]

**Explanation:** The input board is shown above and the only valid solution is shown below:



**Constraints:**

* board.length == 9
* board[i].length == 9
* board[i][j] is a digit or '.'.
* It is **guaranteed** that the input board has only one solution.

# 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.

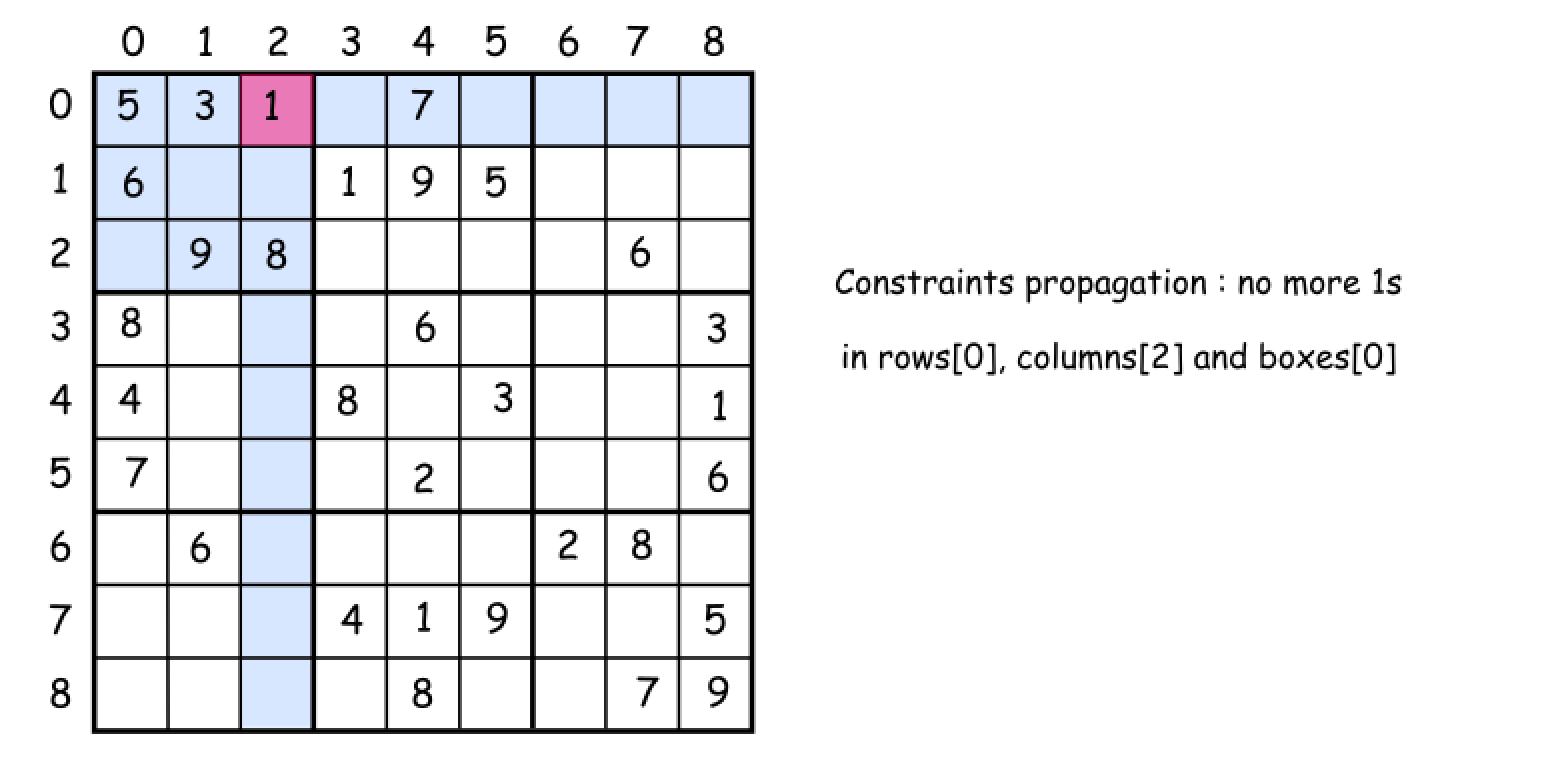
#### **Approach 1: Backtracking**

**Conceptions to use**

There are two programming conceptions here which could help.

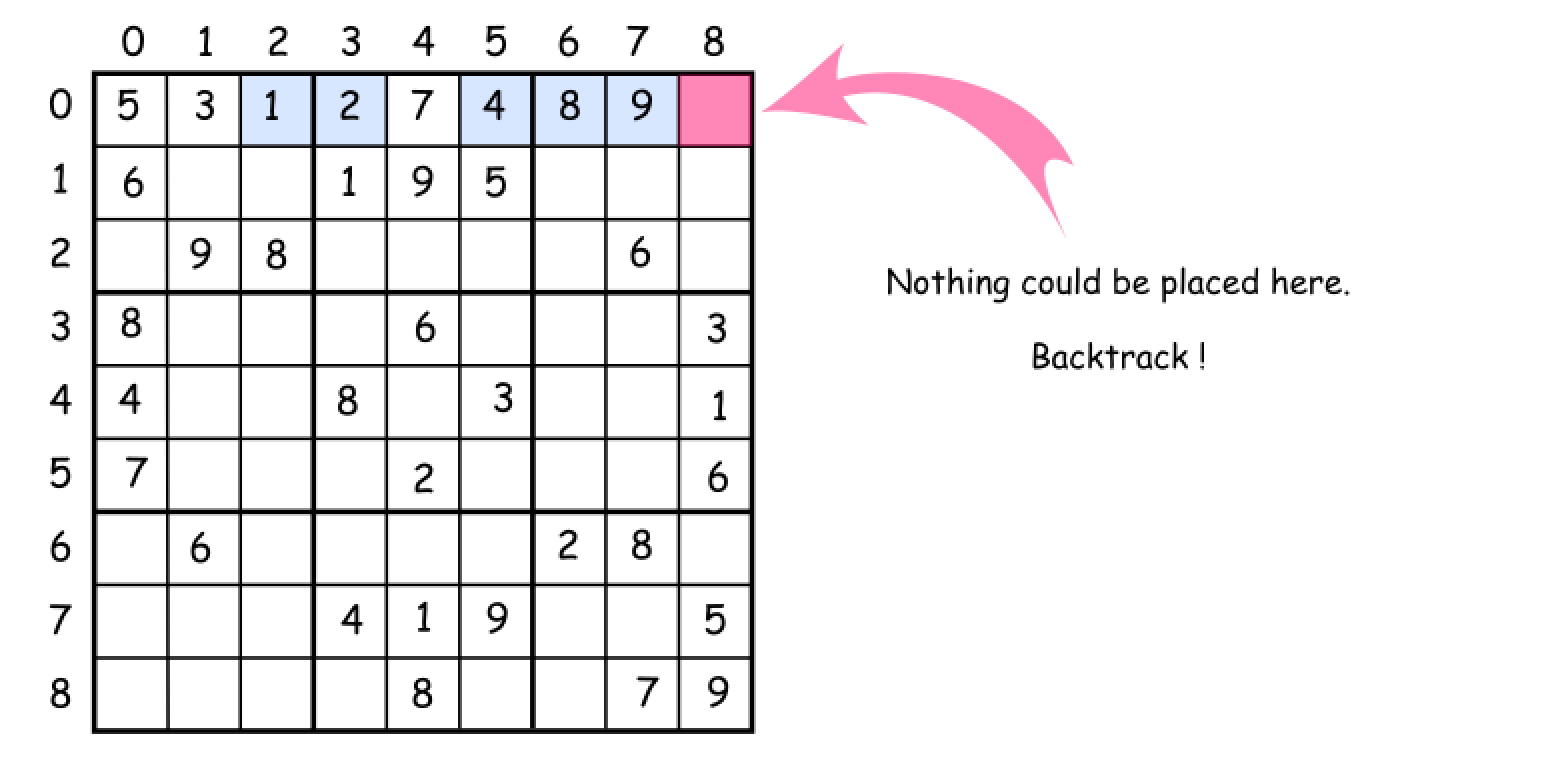
The first one is called constrained programming.

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.



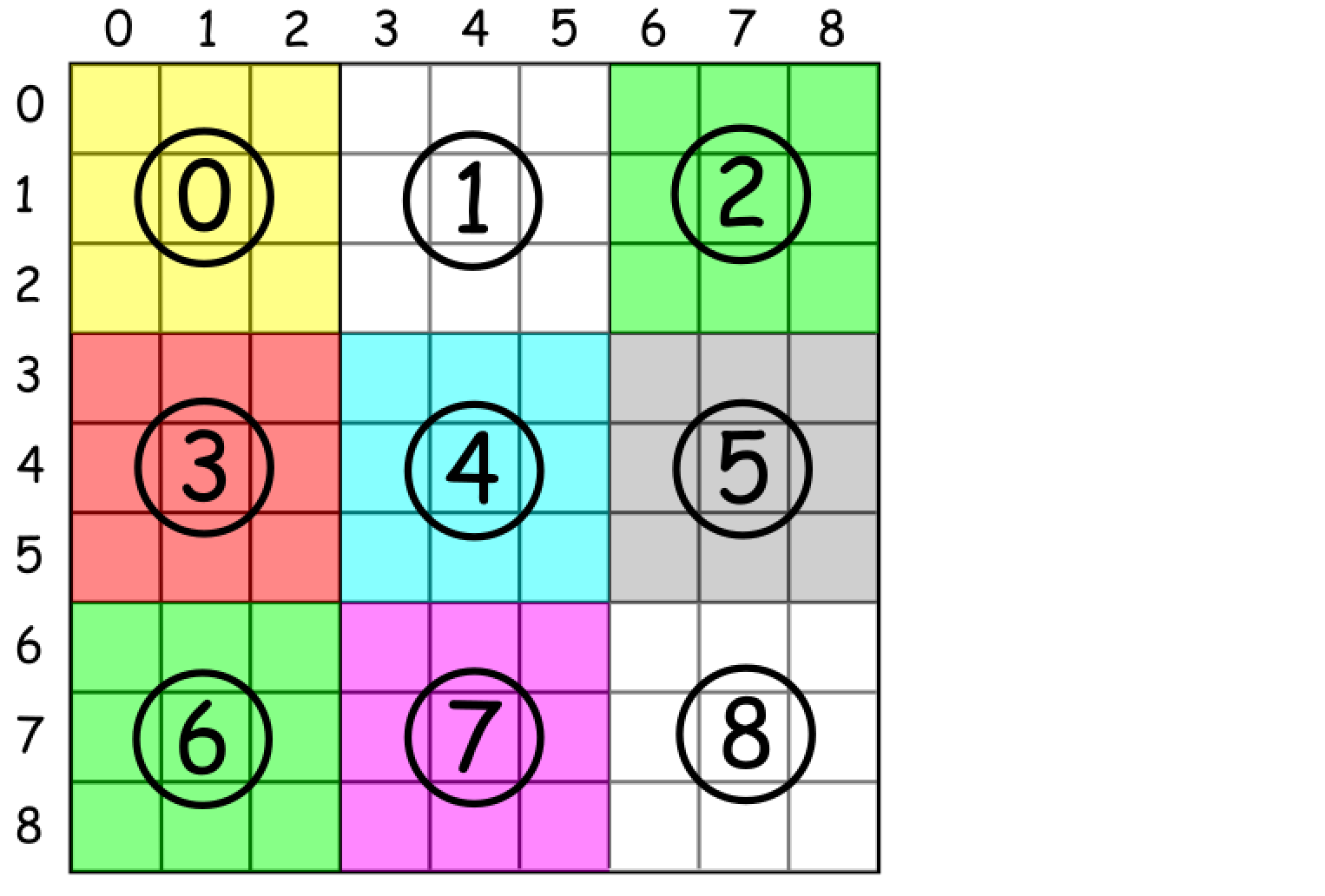
The second one called backtracking.

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**

|  |
| --- |
| class Solution {  // box size  int n = 3;  // row size  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) {  /\*  Check if one could place a number d in (row, col) cell  \*/  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) {  /\*  Place a number d in (row, col) cell  \*/  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) {  /\*  Remove a number which didn't lead to a solution  \*/  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) {  /\*  Call backtrack function in recursion  to continue to place numbers  till the moment we have a solution  \*/  // if we're in the last cell  // that means we have the solution  if ((col == N - 1) && (row == N - 1)) {  sudokuSolved = true;  }  // if not yet  else {  // if we're in the end of the row  // go to the next row  if (col == N - 1) backtrack(row + 1, 0);  // go to the next column  else backtrack(row, col + 1);  }  }  public void backtrack(int row, int col) {  /\*  Backtracking  \*/  // if the cell is empty  if (board[row][col] == '.') {  // iterate over all numbers from 1 to 9  for (int d = 1; d < 10; d++) {  if (couldPlace(d, row, col)) {  placeNumber(d, row, col);  placeNextNumbers(row, col);  // if sudoku is solved, there is no need to backtrack  // since the single unique solution is promised  if (!sudokuSolved) removeNumber(d, row, col);  }  }  }  else placeNextNumbers(row, col);  }  public void solveSudoku(char[][] board) {  this.board = board;  // init rows, columns and boxes  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  char num = board[i][j];  if (num != '.') {  int d = Character.getNumericValue(num);  placeNumber(d, i, j);  }  }  }  backtrack(0, 0);  }  } |

**Complexity Analysis**

