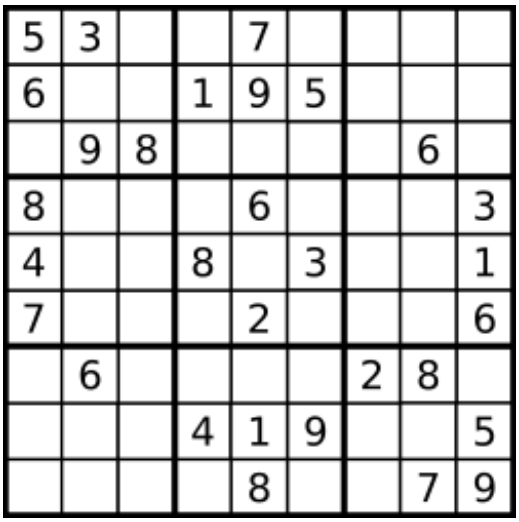
Sudoku Solver

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

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

You may assume that there will be only one unique solution.



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.

Solution 1

Try 1 through 9 for each cell. Details see comments inside code. Let me know your suggestions.

```
public class Solution {
    public void solveSudoku(char[][] board) {
        if(board == null || board.length == 0)
            return;
        solve(board);
    }

    public boolean solve(char[][] board){
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(board[i][j] == '.'){
                    for(char c = '1'; c <= '9'; c++){//trial. Try 1 through 9 for
each cell
                        if(isValid(board, i, j, c)){
                            board[i][j] = c; //Put c for this cell

                            if(solve(board))
                                return true; //If it's the solution return true
                            else
                                board[i][j] = '.'; //Otherwise go back
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }

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

        //Check row
        for(int col = 0; col < 9; col++)
            if(board[i][col] == c)
                return false;

        //Check 3 x 3 block
        for(int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++)
            for(int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++)
                if(board[row][col] == c)
                    return false;
        return true;
    }
}
```

written by kun12 original link here

Solution 2

Update: there's a follow-up oms solution which is even more optimized

This is one of the fastest Sudoku solvers I've ever written. It is compact enough - just 150 lines of C++ code with comments. I thought it'd be interesting to share it, since it combines several techniques like reactive network update propagation and backtracking with very aggressive pruning.

The algorithm is online - it starts with an empty board and as you add numbers to it, it starts solving the Sudoku.

Unlike in other solutions where you have bitmasks of allowed/disallowed values per row/column/square, this solution track bitmask for every(!) cell, forming a set of constraints for the allowed values for each particular cell. Once a value is written into a cell, new constraints are immediately propagated to row, column and 3x3 square of the cell. If during this process a value of other cell can be unambiguously deduced - then the value is set, new constraints are propagated, so on.... You can think about this as an implicit reactive network of cells.

If we're lucky (and we'll be lucky for 19 of 20 of Sudokus published in magazines) then Sudoku is solved at the end (or even before!) processing of the input.

Otherwise, there will be empty cells which have to be resolved. Algorithm uses backtracking for this purpose. To optimize it, algorithm starts with the cell with the smallest ambiguity. This could be improved even further by using priority queue (but it's not implemented here). Backtracking is more or less standard, however, at each step we guess the number, the reactive update propagation comes back into play and it either quickly proves that the guess is unfeasible or significantly prunes the remaining search space.

It's interesting to note, that in this case taking and restoring snapshots of the compact representation of the state is faster than doing backtracking rollback by "undoing the moves".

```
class Solution {
    struct cell // encapsulates a single cell on a Sudoku board
    {
        uint8_t value; // cell value 1..9 or 0 if unset
        // number of possible (unconstrained) values for the cell
        uint8_t numPossibilities;
        // if bitset[v] is 1 then value can't be v
        bitset<10> constraints;
        cell() : value(0), numPossibilities(9),constraints() {};
    };
    array<array<cell,9>,9> cells;

    // sets the value of the cell to [v]
    // the function also propagates constraints to other cells and deduce new val
ues where possible
    bool set(int i, int j, int v)
    {
        // updating state of the cell
        cell& c = cells[i][j];
```

```
        cell& c = cells[i][j];
        if (c.value == v)
            return true;
        if (c.constraints[v])
            return false;
        c.constraints = bitset<10>(0x3FE); // all 1s
        c.constraints.reset(v);
        c.numPossibilities = 1;
        c.value = v;

        // propagating constraints
        for (int k = 0; k<9; k++) {
            // to the row:
            if (i != k && !updateConstraints(k, j, v))
                return false;
            // to the column:
            if (j != k && !updateConstraints(i, k, v))
                return false;
            // to the 3x3 square:
            int ix = (i / 3) * 3 + k / 3;
            int jx = (j / 3) * 3 + k % 3;
            if (ix != i && jx != j && !updateConstraints(ix, jx, v))
                return false;
        }
        return true;
    }
    // update constraints of the cell i,j by excluding possibility of 'excludedVa
lue'
    // once there's one possibility left the function recurses back into set()
    bool updateConstraints(int i, int j, int excludedValue)
    {
        cell& c = cells[i][j];
        if (c.constraints[excludedValue]) {
            return true;
        }
        if (c.value == excludedValue) {
            return false;
        }
        c.constraints.set(excludedValue);
        if (--c.numPossibilities > 1)
            return true;
        for (int v = 1; v <= 9; v++) {
            if (!c.constraints[v]) {
                return set(i, j, v);
            }
        }
        assert(false);
    }

    // backtracking state - list of empty cells
    vector<pair<int, int>> bt;

    // find values for empty cells
    bool findValuesForEmptyCells()
    {
        // collecting all empty cells
        bt.clear();
```

```
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (!cells[i][j].value)
                    bt.push_back(make_pair(i, j));
            }
        }
        // making backtracking efficient by pre-sorting empty cells by numPossibi
lities
        sort(bt.begin(), bt.end(), [this](const pair<int, int>&a, const pair<int,
int>&b) {
            return cells[a.first][a.second].numPossibilities < cells[b.first][b.s
econd].numPossibilities; });
        return backtrack(0);
    }

    // Finds value for all empty cells with index >=k
    bool backtrack(int k)
    {
        if (k >= bt.size())
            return true;
        int i = bt[k].first;
        int j = bt[k].second;
        // fast path - only 1 possibility
        if (cells[i][j].value)
            return backtrack(k + 1);
        auto constraints = cells[i][j].constraints;
        // slow path >1 possibility.
        // making snapshot of the state
        array<array<cell,9>,9> snapshot(cells);
        for (int v = 1; v <= 9; v++) {
            if (!constraints[v]) {
                if (set(i, j, v)) {
                    if (backtrack(k + 1))
                        return true;
                }
                // restoring from snapshot,
                // note: computationally this is cheaper
                // than alternative implementation with undoing the changes
                cells = snapshot;
            }
        }
        return false;
    }
public:
    void solveSudoku(vector<vector<char>> &board) {
        cells = array<array<cell,9>,9>(); // clear array
        // Decoding input board into the internal cell matrix.
        // As we do it - constraints are propagated and even additional values ar
e set as we go
        // (in the case if it is possible to unambiguously deduce them).
        for (int i = 0; i < 9; i++)
        {
            for (int j = 0; j < 9; j++) {
                if (board[i][j] != '.' && !set(i, j, board[i][j] - '0'))
                    return; // sudoku is either incorrect or unsolvable
            }
        }
```

```
        // if we're lucky we've already got a solution,
        // however, if we have empty cells we need to use backtracking to fill the
m
        if (!findValuesForEmptyCells())
            return; // sudoku is unsolvable

        // copying the solution back to the board
        for (int i = 0; i < 9; i++)
        {
            for (int j = 0; j < 9; j++) {
                if (cells[i][j].value)
                    board[i][j] = cells[i][j].value + '0';
            }
        }
    }
};
```

written by 0xF4 original link here

Solution 3

Singapore's prime minister Lee Hsien Loong showcased his Sudoku Solver C code. You can read his original Facebook post here and another news reporting it here.

I have made some slight modification to adapt it so it can be tested on LeetCode OJ. It passed all 6/6 test cases with a runtime of **1 ms**. Pretty impressive for a prime minister, huh?

```
// Original author: Hsien Loong Lee (http://bit.ly/1zfIGMc)
// Slight modification by @1337c0d3r to adapt to run on LeetCode OJ.
// https://leetcode.com/problems/sudoku-solver/
int InBlock[81], InRow[81], InCol[81];

const int BLANK = 0;
const int ONES = 0x3fe;     // Binary 1111111110

int Entry[81];  // Records entries 1-9 in the grid, as the corresponding bit set
to 1
int Block[9], Row[9], Col[9];   // Each int is a 9-bit array

int SeqPtr = 0;
int Sequence[81];



void SwapSeqEntries(int S1, int S2)
{
     int temp = Sequence[S2];
     Sequence[S2] = Sequence[S1];
     Sequence[S1] = temp;
}


void InitEntry(int i, int j, int val)
{
     int Square = 9 * i + j;
     int valbit = 1 << val;
     int SeqPtr2;

     // add suitable checks for data consistency

     Entry[Square] = valbit;
     Block[InBlock[Square]] &= ~valbit;
     Col[InCol[Square]] &= ~valbit; // Simpler Col[j] &= ~valbit;
     Row[InRow[Square]] &= ~valbit; // Simpler Row[i] &= ~valbit;

     SeqPtr2 = SeqPtr;
     while (SeqPtr2 < 81 && Sequence[SeqPtr2] != Square)
           SeqPtr2++ ;

     SwapSeqEntries(SeqPtr, SeqPtr2);
     SeqPtr++;
}
```

```
void PrintArray(char **board)
{
     int i, j, valbit, val, Square;
     char ch;

     Square = 0;

     for (i = 0; i < 9; i++) {
         for (j = 0; j < 9; j++) {
             valbit = Entry[Square++];
             if (valbit == 0) ch = '-';
             else {
                 for (val = 1; val <= 9; val++)
                     if (valbit == (1 << val)) {
                        ch = '0' + val;
                        break;
                     }
             }
             board[i][j] = ch;
         }
     }
}


int NextSeq(int S)
{
    int S2, Square, Possibles, BitCount;
    int T, MinBitCount = 100;

    for (T = S; T < 81; T++) {
        Square = Sequence[T];
        Possibles = Block[InBlock[Square]] & Row[InRow[Square]] & Col[InCol[Squar
e]];
        BitCount = 0;
        while (Possibles) {
           Possibles &= ~(Possibles & -Possibles);
           BitCount++;
        }

        if (BitCount < MinBitCount) {
           MinBitCount = BitCount;
           S2 = T;
        }
    }

    return S2;
}


void Place(int S, char** board)
{
    if (S >= 81) {
        PrintArray(board);
        return;
    }

    int S2 = NextSeq(S);
```

```
    SwapSeqEntries(S, S2);

    int Square = Sequence[S];

    int     BlockIndex = InBlock[Square],
            RowIndex = InRow[Square],
            ColIndex = InCol[Square];

    int     Possibles = Block[BlockIndex] & Row[RowIndex] & Col[ColIndex];
    while (Possibles) {
          int valbit = Possibles & (-Possibles); // Lowest 1 bit in Possibles
          Possibles &= ~valbit;
          Entry[Square] = valbit;
          Block[BlockIndex] &= ~valbit;
          Row[RowIndex] &= ~valbit;
          Col[ColIndex] &= ~valbit;

          Place(S + 1, board);

          Entry[Square] = BLANK; // Could be moved out of the loop
          Block[BlockIndex] |= valbit;
          Row[RowIndex] |= valbit;
          Col[ColIndex] |= valbit;
    }

    SwapSeqEntries(S, S2);
}

void solveSudoku(char **board, int m, int n) {
    SeqPtr = 0;
    int i, j, Square;

    for (i = 0; i < 9; i++)
        for (j = 0; j < 9; j++) {
            Square = 9 * i + j;
            InRow[Square] = i;
            InCol[Square] = j;
            InBlock[Square] = (i / 3) * 3 + ( j / 3);
        }


    for (Square = 0; Square < 81; Square++) {
        Sequence[Square] = Square;
        Entry[Square] = BLANK;
    }

    for (i = 0; i < 9; i++)
        Block[i] = Row[i] = Col[i] = ONES;

    for (int i = 0; i < 9; ++i)
       for (int j = 0; j < 9; ++j) {
           if ('.' != board[i][j])
                InitEntry(i, j, board[i][j] - '0');
       }

    Place(SeqPtr, board);
}
```

```
}
```

written by 1337c0d3r original link here

From Leetcoder.