

Forum:[Hack Forums](#)

Thread: [Sudoku Solver\(4x4\)](#)

saltycracker - October 28, 2020, 11:56 pm

Here's a great challenge for the aspiring programming..

Create a sudoku solver(not a sudoku game) that will take the start of a sudoku game and have your app solve it.

Note: I used 4 x 4 because there is way less setting up for the smaller grid.

Here's a 4 x 4 to get you started.

```
0 3 4 0
4 0 0 2
```

```
1 0 0 3
0 2 1 0
```

Try and get your app to solve this sudoku. Please try this without Googling the solution.

Lynx - October 29, 2020, 12:07 am

I'll do this when I'm home. Made a 9x9 sudoku solver once in c++ but it wasn't as efficient as i wanted it to be.

Great challenge!

saltycracker - October 29, 2020, 3:20 am

(Oct 29, 2020 12:07 AM)Lynx Wrote: [\[CITE\]](#) I'll do this when I'm home. Made a 9x9 sudoku solver once in c++ but it wasn't as efficient as i wanted it to be.

Great challenge!

The challenge is in the C/C++ section but a solution in any language will do.

DizzyPope - October 29, 2020, 5:31 am

I actually find this interesting without being that good of a coder, what's a good approach for this problem?

saltycracker - October 29, 2020, 5:43 am

(Oct 29, 2020 05:31 AM)smurfstrike Wrote: [\[CITE\]](#) I actually find this interesting without being that good of a coder, what's a good approach for this problem?

You have to find the rules for each cell. i.e. What is the way you determine that each cell is valid or invalid.

DizzyPope - October 29, 2020, 7:05 am

(Oct 29, 2020 05:43 AM)saltycracker Wrote: [\[CITE\]](#) You have to find the rules for each cell. i.e. What is the way you determine that each cell is valid or invalid.

And then run all cells in a loop with some checks? Sounds like a lot of work

Lynx - October 29, 2020, 7:06 am

(Oct 29, 2020 05:43 AM)saltycracker Wrote: [\[CITE\]](#) You have to find the rules for each cell. i.e. What

is the way you determine that each cell is valid or invalid.
You mean solving it like humans would?

(Oct 29, 2020 05:31 AM)smurfstrike Wrote: [CITE](#) I actually find this interesting without being that good of a coder, what's a good approach for this problem?
I will address the two ways to do this in my solution for this. Got a preference for a programming language?

Rietty - October 29, 2020, 10:45 am

Fairly straight-forward problem. Does backtracking/recursion in order to generate the solution. Works for any nicely defined grid size. Like (9x9, 4x4, 16x16.. etc).

Code /*

Backtracking solver for Sudoku.

The basic idea of the algorithm is to assign numbers to the empty cells one by one.

If the number is safe to assign, check if the assignment results in a solution.

If it does not, then go to the next safe number and solve that.

Instead of using a 2D array, I'll simply use a 1D vector and calculate the index as needed.

*/

```
#include <iostream>
```

```
#include <vector>
```

```
#define EMPTY 0
```

```
class Board {
```

```
public:
```

```
    // Constructor & Destructor.
```

```
    Board(std::vector<int> grid, int size, int group_size) : m_grid {grid}, m_n {size},  
m_group_size{group_size}{};
```

```
    ~Board(){};
```

```
    // Solve the board.
```

```
    bool solve() {
```

```
        int row, col;
```

```
        if(!unassigned(row, col)) {
```

```
            return true; // We solved it!
```

```
        }
```

```
        for(int value = 1; value <= m_n; value++) {
```

```
            // Is it valid?
```

```
            if(safe(row, col, value)) {
```

```
                set(row, col, value);
```

```
                if(solve()) {
```

```
                    return true;
```

```
                }
```

```
                set(row, col, EMPTY);
```

```
            }
```

```
        }
```

```
        return false;
```

```
    }
```

```
    // Overloaded operator so we can simply use cout to print the grid.
```

```
    friend std::ostream &operator<<(std::ostream &output, const Board &B) {
```

```
        for(int x = 0; x < B.m_n; x++) {
```

```

        for(int y = 0; y < B.m_n; y++) {
            output << B.m_grid[y + (B.m_n * x)] << ' ';
        }
        output << "\n";
    }
    return output;
}

private:
    // Internal functions that are helpful for solving.
    int get(int x, int y) {
        return m_grid[(x + (m_n * y))]; // Returns the value at grid location x, y.
    }

    void set(int x, int y, int value) {
        m_grid[(x + (m_n * y))] = value; // Sets the value for the grid location x, y.
    }

    bool unassigned(int& row, int& col) {
        for(row = 0; row < m_n; row++) {
            for(col = 0; col < m_n; col++) {
                if(get(row, col) == EMPTY) return true; // Returns true and stops the function when it
finds an empty square.
            }
        }
        return false; // If we return false, we've essentially solved the problem.
    }

    bool checkRow(int row, int value) {
        for(int col = 0; col < m_n; col++) {
            if(get(row, col) == value) return true; // Return true if we have already used the number in
the row.
        }
        return false;
    }

    bool checkCol(int col, int value) {
        for(int row = 0; row < m_n; row++) {
            if(get(row, col) == value) return true; // Return true if we have already used the number in
the col.
        }
        return false;
    }

    bool checkBox(int row, int col, int value) {
        for(int r = 0; r < m_group_size; r++) {
            for(int c = 0; c < m_group_size; c++) {
                if(get(r + row, c + col) == value) return true; // Return true if the bounded box has
already used the number.
            }
        }
        return false;
    }
}

```

```

    bool safe(int row, int col, int value) {
        // Check to see if the position is legal/safe to place a value in.
        return !checkRow(row, value) && !checkCol(col, value) && !checkBox(row - row %
m_group_size, col - col % m_group_size, value) && get(row, col) == EMPTY;
    }

    // Variables.
    std::vector<int> m_grid;
    int m_n, m_group_size;
};

int main(int argc, char const *argv[]) {
    Board board({0, 3, 4, 0, 4, 0, 0, 2, 1, 0, 0, 3, 0, 2, 1, 0}, 4, 2);
    if(board.solve()) {
        std::cout << "Solution:" << std::endl << board << std::endl;
    } else {
        std::cout << "Can't solve this!" << std::endl;
    }
    return 0;
}

```

To solve the problem we pass it a board defined as a 1D vector, as well as size of N, and of the smaller sub-box within. (i.e. the 2x2 or 3x3 box.)

saltycracker - October 29, 2020, 1:29 pm

(Oct 29, 2020 07:06 AM)Lynx Wrote: [\[CITE\]](#) You mean solving it like humans would?

There is multi ways of designing this. You could limit the input by generating a subset of the valid input by inspecting the starting grid values.

I'm trying this in Perl with references to a 1d array. Basically defining edges for each node and then use a recursive function to walk up and down checking the subset of inputs.

I got it working for one case and now I'm generalizing it.

Well my Perl version of the 4x4 sudoku solver works(well for the cases I tried). I think I went overboard and made it wayyyy too complicated.

Code #! /usr/bin/env perl

```

use warnings;
use strict;
use utf8;
use feature qw<say>;
use boolean;
use List::Util qw<any uniq>;

```

```

my @cells;

```

```

my %posToElem = (
    11 => 0,
    12 => 1,
    13 => 2,
    14 => 3,

```

```

21 => 4,
22 => 5,
23 => 6,
24 => 7,
31 => 8,
32 => 9,
33 => 10,
34 => 11,
41 => 12,
42 => 13,
43 => 14,
44 => 15
);

```

```

my @sudokuPos = (#edges for each position
[11, [12, 13, 14, 21, 22, 31, 41]],
[12, [11, 13, 14, 21, 22, 32, 42]],
[13, [11, 12, 14, 23, 24, 33, 43]],
[14, [11, 12, 13, 23, 24, 34, 44]],
[21, [22, 23, 24, 11, 12, 31, 41]],
[22, [21, 23, 24, 11, 12, 32, 42]],
[23, [21, 22, 24, 13, 14, 33, 43]],
[24, [21, 22, 23, 13, 14, 34, 44]],
[31, [32, 33, 34, 41, 42, 11, 21]],
[32, [31, 33, 34, 41, 42, 12, 22]],
[33, [31, 32, 34, 43, 44, 13, 23]],
[34, [31, 32, 33, 43, 44, 14, 24]],
[41, [42, 43, 44, 31, 32, 11, 21]],
[42, [41, 43, 44, 31, 32, 12, 22]],
[43, [41, 42, 44, 33, 34, 13, 23]],
[44, [41, 42, 43, 33, 34, 14, 24]]
);

```

```

sub getSudokuCells {
    my $sudokuStr;

    {
        local $/ = undef;
        chop($sudokuStr = <DATA>);
    }

    map {int($_)} (split(/, $sudokuStr));
}

```

```

sub fixedCell {
    my ($num) = @_ ;
    $num == 0 ? false : true;
}

```

```

sub calValidSet {
    my($cells, $edges) = @_ ;
    my @ans;
    for my $elem (@$cells) {

```

```

    if (any{$_ == $elem} (@$edges)) {
        push(@ans, $elem);
    }
}
\@ans;
}

```

```

sub buildSudokuStructure {
    my ($r_cells, $p_to_e) = @_;
    my @ans;
    while (my ($index, $elem) = each (@$r_cells)) {
        my @edges;
        for my $e (@{$sudokuPos[$index]->[1]}) {
            push(@edges, \ $r_cells->[$p_to_e->{$e}]);
        }
        push(
            @ans,
            {
                fixed => fixedCell($elem),
                elem => $elem,
                pos => $sudokuPos[$index]->[0],
                edges => \@edges
            });
    }
    \@ans;
}

```

```

sub startingNumbers {
    my ($edges) = @_;
    my @nums;
    my @ans;
    for my $n (@$edges) {#ref to cell value
        push(@nums, $$n) if $$n;
    }
    @nums = uniq @nums;
    for my $elem (1..4) {
        push(@ans, $elem) unless any {$elem == $_} @nums;
    }
    \@ans;
}

```

```

sub generateStartingNums {
    my ($sudoku) = @_;
    for my $elem (@$sudoku) {
        if ($elem->{fixed}) {
            $elem->{startingNums} = [];
        } else {
            $elem->{startingNums} = startingNumbers($elem->{edges});
        }
    }
}

```

```

sub validateTry {

```

```

my ($edges, $value) = @_;
for my $elem (@$edges) {
    return false if ($$elem == $value);
}
true;
}

```

```

sub solveSudoku {
    my ($sudoku) = @_;
    if ($sudoku->[0]->{fixed}) {
        solveSudokuAux($sudoku, 1);
    }else {
        for my $elem (@{$sudoku->[0]->{startingNums}}) {
            if (validateTry($sudoku->[0]->{edges}, $elem)) {
                my $old_value = $cells[0];
                @cells[0] = $elem;
                solveSudokuAux($sudoku, 1);
                @cells[0] = $old_value;
            }
        }
    }
}

```

```

say "The solving is all done";
}

```

```

sub solveSudokuAux {
    my ($sudoku, $pos) = @_;
    if ($pos < 16) {
        if ($sudoku->[$pos]->{fixed}) {
            solveSudokuAux($sudoku, ++$pos);
        }else {
            for my $elem (@{$sudoku->[$pos]->{startingNums}}) {
                if (validateTry($sudoku->[$pos]->{edges}, $elem)) {
                    my $old_value = $cells[$pos];
                    my $old_pos = $pos;
                    @cells[$pos] = $elem;
                    solveSudokuAux($sudoku, ++$pos);
                    @cells[$old_pos] = $old_value;
                    $pos = $old_pos;
                }
            }
        }
    }
    }else {
        my $g_counter = 0;
        my $l_counter = 0;
        my $s_counter = 0;
        for my $elem (@cells) {
            ++$g_counter;
            ++$l_counter;
            ++$s_counter;
            print $elem, " ";
            if ($g_counter > 1) {
                print "\n";
            }
        }
    }
}

```

```

    $g_counter = 0;
}
if ($l_counter > 3) {
    print "\n";
    $l_counter = 0;
}
if ($s_counter > 7) {
    print "\n";
    $s_counter = 0;
}
}
}
}
}
}

```

```
@cells = getSudokuCells();
```

```
my $sudokuStr = buildSudokuStructure(\@cells, \%posToElem);
```

```
generateStartingNums($sudokuStr);
```

```
solveSudoku($sudokuStr);
```

```
exit(0);
```

```
#0340400210030210
```

```
#0000430000420000-
```

```
#00104000000020300
```

```
#2000003004000001-
```

```
#10400000000000102
```

```
__DATA__
```

```
00104000000020300
```

The above produces:

```
3 2 1 4
```

```
4 1 2 3
```

```
1 4 3 2
```

```
2 3 4 1
```

The solving is all done

saltycracker - December 1, 2020, 10:50 pm

Decided to post my (9x9) solution in OCaml

```
Code let sudokuRowSize = 9
```

```
let sudokuBoxSize = 3
```

```
let sudokuValidCharacters = [1;2;3;4;5;6;7;8;9;]
```

```
let arr =
```

```
[[
  0;0;0; 0;0;8; 0;0;0;
```

```
  2;0;0; 0;5;9; 0;0;0;
```

```
  1;0;0; 0;0;0; 2;5;0;
```

```

0;0;8; 0;0;1; 4;6;0;
3;0;0; 0;0;0; 1;0;0;
9;0;4; 0;8;3; 0;0;0;

0;0;0; 5;0;0; 0;8;4;
0;0;0; 0;0;4; 9;0;0;
0;2;3; 0;0;0; 0;0;0;
[]

```

```

let buildRow p =
  let r = p / sudokuRowSize in
  let rp = p mod sudokuRowSize in
  let rec buildRowAux cnt acc =
    match cnt with
    | -1 -> acc
    | x when x = rp -> buildRowAux (cnt - 1) acc
    | x -> buildRowAux (cnt - 1) ((x + (r * sudokuRowSize))::acc) in
  buildRowAux (sudokuRowSize - 1) []

```

```

let buildCol p =
  let r = p / sudokuRowSize in
  let rp = p mod sudokuRowSize in
  let rec buildColAux cnt acc =
    match cnt with
    | -1 -> acc
    | x when x = r -> buildColAux (cnt - 1) acc
    | x -> buildColAux (cnt - 1) ((rp + (x * sudokuRowSize))::acc) in
  buildColAux (sudokuRowSize - 1) []

```

```

let buildBoxArr p =
  if p = 0
  then
    [1; 2; ]
  else if p = 1
  then
    [-1; 1; ]
  else
    [-2; -1; ]

```

```

let buildBox p =
  let r = (p / sudokuRowSize) mod sudokuBoxSize in
  let c = p mod sudokuBoxSize in
  let ra = buildBoxArr r in
  let ca = buildBoxArr c in
  List.fold_left
    (
      fun ar dr ->
        List.fold_left
          (
            fun ac dc -> (p + dc + (dr * sudokuRowSize))::ac
          ) ar ca
    ) [] ra

```

```

let buildStartingValues e =
  List.fold_left
    (
      fun a d ->
        if (List.exists (fun ed -> arr.(ed) = d) e)
        then
          a
        else
          d::a
    ) [] sudokuValidCharacters

let sudokuArr =
  Array.mapi
    (
      fun p v ->
        let fixed =
          if v = 0
          then
            false
          else
            true in
        let edges =
          if fixed
          then
            []
          else
            (buildRow p) @ (buildCol p) @ (buildBox p) in
        let startingValues =
          if fixed
          then
            []
          else
            buildStartingValues edges in
        (
          p,
          v,
          fixed,
          edges,
          startingValues
        )
    ) arr

let lst =
  Array.fold_left (fun a d -> d::a) [] sudokuArr

let rec solveIt l =
  match l with
  | [] ->
    (
      print_endline("Done");
      Array.iter (fun x -> Printf.printf "%d, " x) arr
    )

```

```
| (p, v, f, e, sv)::tl ->
  if f
  then
    solvelt tl
  else
    List.iter
      (
        fun s ->
          if (List.exists (fun ed -> arr.(ed) = s) e)
          then
            ()
          else
            (
              arr.(p) <- s;
              solvelt tl;
              arr.(p) <- 0;
            )
      ) sv
```

let () = solvelt lst

saltycracker - December 7, 2020, 7:48 am

I tried a solution in Emacs Lisp.. My apologies to any Lisp programmers.

<https://pastebin.com/S54edcLj>

FW1738 - December 7, 2020, 8:34 am

(Oct 29, 2020 01:29 PM)saltycracker Wrote: [\[CITE\]](#) The above produces:

```
3 2 1 4
4 1 2 3
```

```
1 4 3 2
2 3 4 1
```

The solving is all done

The diagonal needs to be solved too right? So the answer needs to be:

```
3 2 4 1
4 1 3 2
```

```
1 4 2 3
2 3 1 4
```

saltycracker - December 7, 2020, 12:24 pm

(Dec 7, 2020 09:34 AM)FW1738 Wrote: [\[CITE\]](#) The diagonal needs to be solved too right? So the answer needs to be:

```
3 2 4 1
4 1 3 2
```

```
1 4 2 3
2 3 1 4
```

That's not part of the sudoku's rules.

saltcracker - *December 14, 2020, 8:35 am*

Here's a similar solution but in Ruby.

<https://pastebin.com/BRwGE9gH>