sudoku

Usage

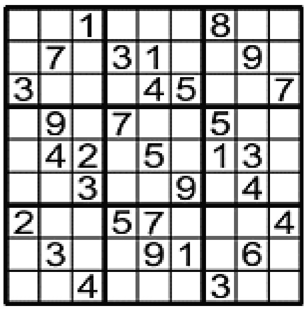
`sudoku` *`puzzlefile`*

`sudoku_c` *`puzzlefile`*

Description

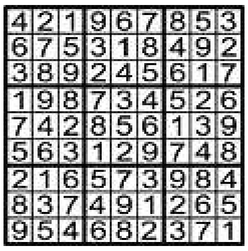
A type of puzzle called "sudoku" has recently gained popularity. I first heard of it in a Slate article (here). There's a good Wikipedia article here.

Briefly: you are given a 9x9 grid of squares; some of the squares have digits. For example:



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

Your task is to fill in the empty squares so that each row, column, and 3x3 "region" (outlined in bold) contains the digits 1-9 exactly once. For example, the solution to the above puzzle is:



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

This struck me as an easy and fun problem to solve on the computer. I have two programs here, written in C for Unix-like systems. Both use the same solution algorithm, but one simply outputs the solution (if any), the other uses curses terminal positioning to animate the solution process.

Data Structures

… such as they are. The sensible thing is to use a 2-d array to hold the grid:

```
  #define GRIDSIZE 9

  unsigned int grid[GRIDSIZE][GRIDSIZE];
```

A zero element means the grid square is empty; otherwise it holds a value between 1 and 9.

I used unsigned values to hold "bitmaps" to keep track of the different values currently held in each row, column, and 3x3 region.

```
  unsigned int col[GRIDSIZE];
  unsigned int row[GRIDSIZE];
  unsigned int region[GRIDSIZE];
```

Columns, rows, and regions are numbered C-like from 0 to 8. Columns and rows are easy enough: grid square `grid[i][j]` corresponds to the column bitmap `col[j]` and the row bitmap `row[i]`. But what about regions? It turns out there's a simple expression that will correctly figure out a region index given `i` and `j`:

```
  3 * (i/3) + (j/3)
```

… where integer division is used, of course.

While solving the puzzle, placing a digit `n` in a square turns on bit `n-1` of the appropriate row, column, and region value. Coded:

```
    grid[i][j] = n;
    row[i] ^= (1 << (n-1));
    col[j] ^= (1 << (n-1));
    region[3 * (i/3) + (j/3)] ^= (1 << (n-1));
```

Removing the value, if we need to, is the *same* operation, just setting `grid[i][j]` to zero instead. (The exclusive-or operation simply toggles the appropriate bit on or off.)

Testing whether it's "legal" to put a digit `n` in a grid square is similarly straightforward: check if (a) the grid square is empty; (b) the row doesn't have `n` in it already; (c) the column doesn't have `n` in it already; and (d) the region doesn't have `n` in it already. Translating that into a C expression:

```
    (grid[i][j] == 0) &&
    ((row[i] & (1 << (n-1))) == 0) &&
    ((col[j] & (1 << (n-1))) == 0) &&
    ((region[3 * (i/3) + (j/3)] & (1 << (n-1))) == 0);
```

Finally, to keep track of the empty squares, I implemented a stack:

```
   unsigned int istack[GRIDSIZE*GRIDSIZE];
   unsigned int jstack[GRIDSIZE*GRIDSIZE];
   unsigned int stackp;
```

… one array for the empty-square row numbers (`istack[]`) another for empty-square column numbers (`jstack[]`), and a stack pointer (`stackp`) to keep track of how many empty squares there are. Shades of FORTRAN!

Algorithm

I wrote a recursive function `solve()` to crank out a brute-force solution. Given the grid, bitmaps, and empty-square stack described above, it attempts to solve the puzzle; it returns a success/failure (1/0) flag.

Pseudo-code:

```
   if there are no empty grid squares
       we're done, the puzzle is solved
   otherwise, get the next empty square off the stack;
   for each digit 1..9:
       if it's legal to put that digit in the empty square, do so
       if there's a solution from this point
           we're done, return success
       otherwise remove the digit
   if we've tried all the digits with no solution
       then there's no solution from this point;
       put the empty square back on the stack and
       return failure.
```

You can check the source to see how this translates into C.

I/O

Input is very rudimentary and user-hostile. An input filename is specified by a single command line argument; the specified file is expected to have 9 lines of 9 characters each. Each non-zero digit in the line corresponds to a initially filled-in grid square, any other character sets the square initially empty. For the example grid shown above, the input file could like this:

```
--1---8--
-7-31--9-
3---45--7
-9-7--5--
-42-5-13-
--3--9-4-
2--57---4
-3--91-6-
--4---3--
```

The first (non-curses) program produces the following output when that input file is specified:

```
+---+---+---+---+---+---+---+---+---+
|   |   | 1 |   |   |   | 8 |   |   |
+---+---+---+---+---+---+---+---+---+
|   | 7 |   | 3 | 1 |   |   | 9 |   |
+---+---+---+---+---+---+---+---+---+
| 3 |   |   |   | 4 | 5 |   |   | 7 |
+---+---+---+---+---+---+---+---+---+
|   | 9 |   | 7 |   |   | 5 |   |   |
+---+---+---+---+---+---+---+---+---+
|   | 4 | 2 |   | 5 |   | 1 | 3 |   |
+---+---+---+---+---+---+---+---+---+
|   |   | 3 |   |   | 9 |   | 4 |   |
+---+---+---+---+---+---+---+---+---+
| 2 |   |   | 5 | 7 |   |   |   | 4 |
+---+---+---+---+---+---+---+---+---+
|   | 3 |   |   | 9 | 1 |   | 6 |   |
+---+---+---+---+---+---+---+---+---+
|   |   | 4 |   |   |   | 3 |   |   |
+---+---+---+---+---+---+---+---+---+
Solution found!
+---+---+---+---+---+---+---+---+---+
| 4 | 2 | 1 | 9 | 6 | 7 | 8 | 5 | 3 |
+---+---+---+---+---+---+---+---+---+
| 6 | 7 | 5 | 3 | 1 | 8 | 4 | 9 | 2 |
+---+---+---+---+---+---+---+---+---+
| 3 | 8 | 9 | 2 | 4 | 5 | 6 | 1 | 7 |
+---+---+---+---+---+---+---+---+---+
| 1 | 9 | 8 | 7 | 3 | 4 | 5 | 2 | 6 |
+---+---+---+---+---+---+---+---+---+
| 7 | 4 | 2 | 8 | 5 | 6 | 1 | 3 | 9 |
+---+---+---+---+---+---+---+---+---+
| 5 | 6 | 3 | 1 | 2 | 9 | 7 | 4 | 8 |
+---+---+---+---+---+---+---+---+---+
| 2 | 1 | 6 | 5 | 7 | 3 | 9 | 8 | 4 |
+---+---+---+---+---+---+---+---+---+
| 8 | 3 | 7 | 4 | 9 | 1 | 2 | 6 | 5 |
+---+---+---+---+---+---+---+---+---+
| 9 | 5 | 4 | 6 | 8 | 2 | 3 | 7 | 1 |
+---+---+---+---+---+---+---+---+---+
```

The curses version is slightly prettier, and asks you to "press any key" to load the puzzle, and to solve it. Animation shows how various numbers are tried in empty squares. A call to `sleep(1)` at the top of the `solve()` function slows it down so viewers can get the basic idea of what the recursive algorithm is doing as it tries things and backs off if they don't work out.

Notes

- Input puzzles can have no solution, many solutions, or might be internally inconsistent. No effort is made to detect, let alone distinguish, these cases.
- As I said, it's brute-force. If I had to make it "smarter", the best way might be to investigate empty squares in a better order. For example, if the current grid state implies only one possible value be put into an empty square, do it and move on from there. Left as an exercise for the reader. Since it's "fast enough" as is, this is truly an academic exercise.
- Similarly, a GUI might be nice.
- As usual for hacks, no claims are made for the software-engineering quality of the code.

Non-Curses Source

Curses Source

Back to Hacks

Last modified: July 15 2005 08:33 EDT

Paul A. Sand, *pas@unh.edu*