Sudoku Solving Algorithm in C

Alexander Barry

CMPE3232

University of New Brunswick

Jan. 2012 – Apr. 2012

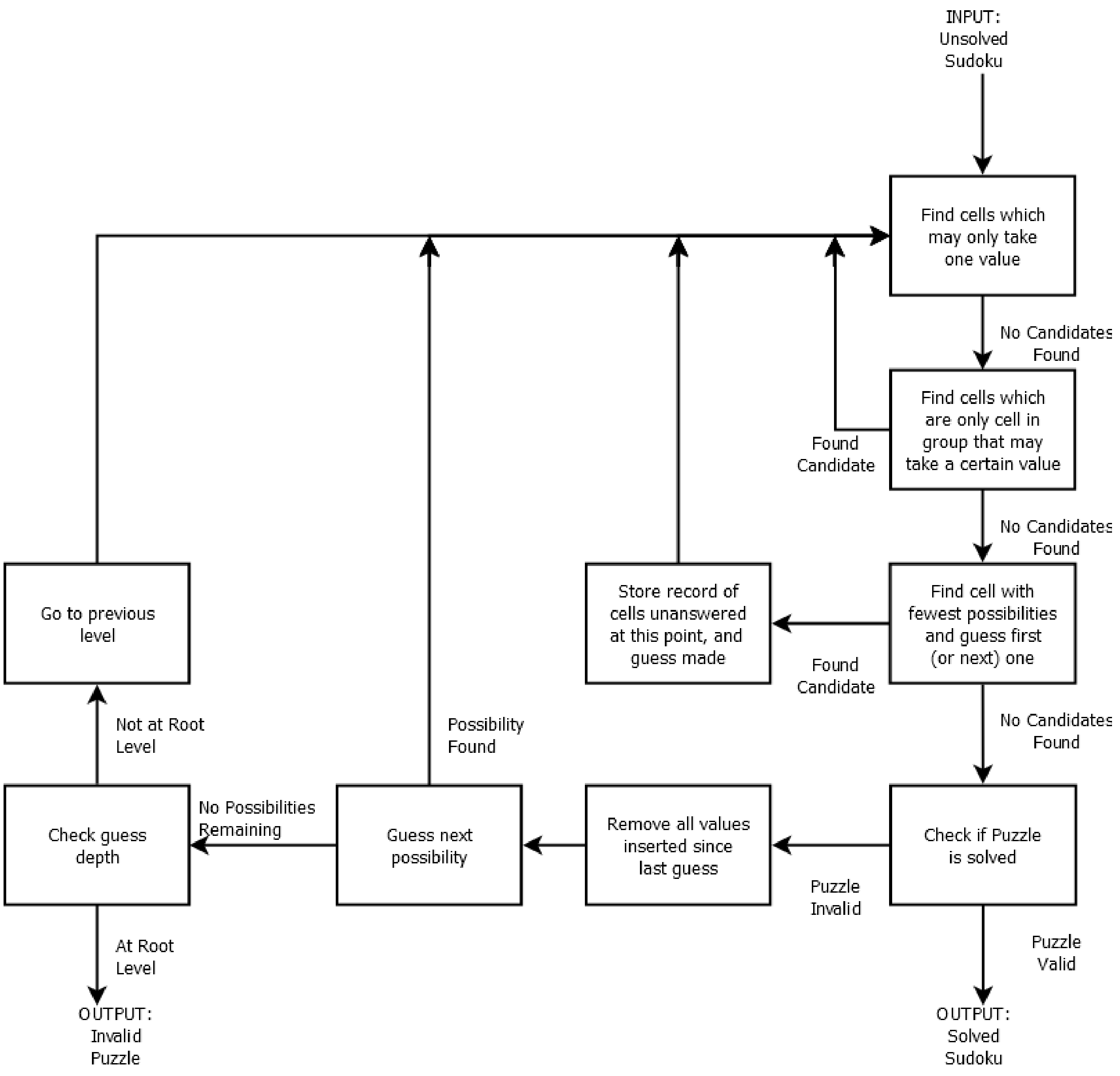
This document is part of a larger document for the rest of the project.

This document is my largest contribution to the project's final document: an algorithm to solve Sudoku puzzles, written in C. It was implemented on a microcontroller, but it relies on no hardware so it works just as well on any valid C platform.

It uses some `printf` functions because it was written and tested (initially) on a personal computer, but on the microcontroller it handled output in another way (another part of the project displayed the puzzle and its solution on a VGA monitor).

Sudoku Solving Algorithm

This algorithm may be briefly summarized with the below figure:


INPUT: Unsolved Sudoku
Find cells which may only take one value
No Candidates Found
Find cells which are only cell in group that may take a certain value
Found Candidate
No Candidates Found
Find cell with fewest possibilities and guess first (or next) one
Found Candidate
Store record of cells unanswered at this point, and guess made
No Candidates Found
Check if Puzzle is solved
Puzzle Valid
OUTPUT: Solved Sudoku
Puzzle Invalid
Remove all values inserted since last guess
Guess next possibility
Possibility Found
No Possibilities Remaining
Check guess depth
Not at Root Level
Go to previous level
At Root Level
OUTPUT: Invalid Puzzle

How each step is achieved is outlined in detail in this section of the report.

Type definitions and Global variables

```
#define ROW 0
#define COL 1
#define BOX 2
```

```
char OrigSudoku[9][9] = {
{1,2,0,  3,0,0,  0,0,4},
{3,5,0,  0,0,0,  1,0,0},
{0,0,4,  0,0,0,  0,0,0},

{0,0,5,  4,0,0,  2,0,0},
{6,0,0,  0,7,0,  0,0,0},
{0,0,0,  0,0,8,  0,9,0},

{0,0,3,  1,0,0,  5,0,0},
{0,0,0,  0,0,9,  0,7,0},
{0,0,0,  0,6,0,  0,0,8},
};

volatile char Sudoku[9][9] = {
{0,0,0,  0,0,0,  0,0,0},
{0,0,0,  0,0,0,  0,0,0},
{0,0,0,  0,0,0,  0,0,0},

{0,0,0,  0,0,0,  0,0,0},
{0,0,0,  0,0,0,  0,0,0},
{0,0,0,  0,0,0,  0,0,0},

{0,0,0,  0,0,0,  0,0,0},
{0,0,0,  0,0,0,  0,0,0},
{0,0,0,  0,0,0,  0,0,0},
};

int Cols[9]  = {0,0,0, 0,0,0, 0,0,0};
int Rows[9]  = {0,0,0, 0,0,0, 0,0,0};
int Boxes[9] = {0,0,0, 0,0,0, 0,0,0};

struct _point {
    char x;
    char y;
};
typedef struct _point point;
```

The definitions seen here are simply:

- The definitions of `ROW, COL,` and `BOX` are to refer to each group in later functions.
- `volatile char OrigSudoku[9][9], Sudoku[9][9];`
    - These simply store the input, and the puzzle while it is being solved.
- `int Cols[9], Rows[9], Boxes[9];`
    - These are 9 entry arrays corresponding to one for each column/row/box, where each entry is 9 bits corresponding to which digits (1..9) are currently in the Sudoku grid.
- `point;`
    - This is simply a struct with an *x* and *y* coordinate.

There are other global variables and structure definitions, but are not relevant until much later in the algorithm, and will be presented accordingly.

With these definitions alone, many functions for basic data manipulation may be defined.

Basic function definitions

It must first be understood that this algorithm relies heavily on recording values as one bit of 9, rather than numbers 1..9. For example, `0b000001000` corresponds to the value 4, as the 4th bit (counting the least significant bit as the 1st) is on. `0b000001001` would correspond to the values 4 and 1.

The main benefit of this method is that the algorithm is constantly checking which rows/columns/boxes contain certain values. Storing them like this allows the calculation to be performed by simply looking at the right bit, rather than having to loop through part of the grid every time, effectively calculating the same thing.

For clarity, functions can be used to make this conversion (note they are essentially 2^x and $\log_2 x$):

```
    /* int numToBits(char num)
     *  Converts a number from 1 to 9 to its
     *  corresponding bit (0b1 to 0b1,0000,0000)
     */
int numToBits(char num) {
    if( num == 0){ return 0; }
    else { return 1<<(num-1); }
}

    /* char bitsToNumMin(int bits, char nextNum);
     *  Returns first value of bit greater than nextNum
     *  Returns 0 if none.
     */
char bitsToNumMin(int bits, char min) {
    char n = 1;
    while(bits > 0) {
        if( (bits & 1) == 1 && (n > min) ) { return n; }
        bits >>= 1;
        n++;
    }
    return 0;
}

    /* char bitsToNum(int bits);
     *  Returns first value of bits
     *  Returns 0 if none.
     */
char bitsToNum(int bits) {
    return bitsToNumMin(bits, 0);
}
```

As this program relies heavily on the number of possibilities a value has, another function may easily be implemented to count the number of bits on in a value:

```
    /* char countBits(int bits);
     *  Returns number of bits in input.
     */
char countBits(int bits) {
    char count = 0;
    while( bits > 0 ) {
        if( bits & 1 ) { count++; }
        bits >>= 1;
    }
    return count;
}
```

Sudoku specific data manipulation

Next, Sudoku specific function definitions may be defined:

A Sudoku grid is split into Rows, Columns, and Boxes. The latter is the least easy to compute from its point, so a function would be beneficial:

```
    /* char boxNum(point pt);
     *  Returns points corresponding box number.
     */
char  boxNum(point pt) {
    return (pt.x - 1 )/3 + (( pt.y - 1)/3)*3;
}
```

Other point specific functions may be defined as well: returning the first point in any Row/Column/Box (referred to as ‘type’ or ‘group’ later on), and incrementing the point to the next position in the group specified:

```
    /* point first(char type, char pos)
     *  Returns first point in number pos group 'type'
     *  e.g. first(COL, 3) returns point (3,1)
     */
point first(char type, char pos) {
    point pt;
    if( type == ROW ) {
        pt.x = 1;
        pt.y = pos;
    } else if ( type == COL ) {
        pt.x = pos;
        pt.y = 1;
    } else if ( type == BOX ) {
        pos -=1;
        pt.x = 3*(pos%3) + 1;
        pt.y = 3*(pos/3) + 1;
    }

    return pt;
}

    /* bool nextPt(point,type)
     * Goes to next point in type
     * Returns TRUE when complete (FALSE when not)
     */
char nextPt(point* pt, char type) {
    if( type == ROW ) {
        if( pt->x >= 9) {    return FALSE; }
        else {           pt->x++; return TRUE; }
    }
    if (type == COL ) {
        if( pt->y >= 9) {    return FALSE; }
        else {           pt->y++; return TRUE; }
    }
    if (type == BOX ) {
        if ( pt->y % 3 == 0 && pt->x % 3 == 0) { return FALSE; }
        if ( pt->x % 3 == 0 ) { pt->x -= 2; pt->y += 1; return TRUE; }
        else { pt->x++; return TRUE; }
    } else {
        return FALSE;
    }
    return TRUE;
}
```

It is also convenient to define functions making the appropriate updates to each array when inserting a new value, or removing a value:

```
    /* void insertVal(point pt, char num)
     *  Inserts num into sudoku grid at point pt,
     *      as well as into col/row/box arrays.
     */
void insertVal(point pt, char num) {
    int bits = numToBits(num);

    if( Sudoku[pt.y - 1][pt.x - 1] != 0 ) { return; }
    if( num == 0 ) { return; }
//  printf("Inserting value %d into point ", num); printPt(pt); printf("\n");

    Rows[pt.y-1] |= bits;
    Cols[pt.x-1] |= bits;
    Boxes[boxNum(pt)] |= bits;
    Sudoku[pt.y-1][pt.x-1] = num;
}

    /* void removeVal(point pt);
     *  Removes value in point pt from
     *  all necessary locations
     */
void removeVal(point pt) {
    // complemented
    int inv_bits = 0x1FF ^ numToBits(Sudoku[pt.y-1][pt.x-1]);
    Sudoku[pt.y-1][pt.x-1] = 0;

    Rows[pt.y-1] &= inv_bits;
    Cols[pt.x-1] &= inv_bits;
    Boxes[boxNum(pt)] &= inv_bits;
}
```

Main initialization

The first initial step of the main function is defined here to more clearly outline the important logic later:

```
void main(void) {
    point pt;
    char activity, type, num, valBits;
    int exclusiveBits, bits;
    guess* tmpGuess;

    init();


    // Initialize rows/columns
    for(pt.y=1; pt.y<=9; pt.y++) {
        for(pt.x=1; pt.x<=9; pt.x++) {
            insertVal( pt, OrigSudoku[pt.y-1][pt.x-1] );
        }
    }

    while(activity) {
        activity = FALSE;
    // code omitted and explained later for clarity,
    // note this loop is far from complete
```

Note this loop is not complete, and each step (not shown) corresponds to the steps shown in the block diagram shown earlier, which are defined in detail next.

Step 1 Solving functions: Cells which have only one possibility

The most essential function may be seen next, which takes advantage of the `Cols[], Rows[], Boxes[]` arrays defined previously to efficiently compute which values are legal to insert in a cell. The function returns a 9 bit value corresponding to the 9 digits which are valid in Sudoku (1..9):

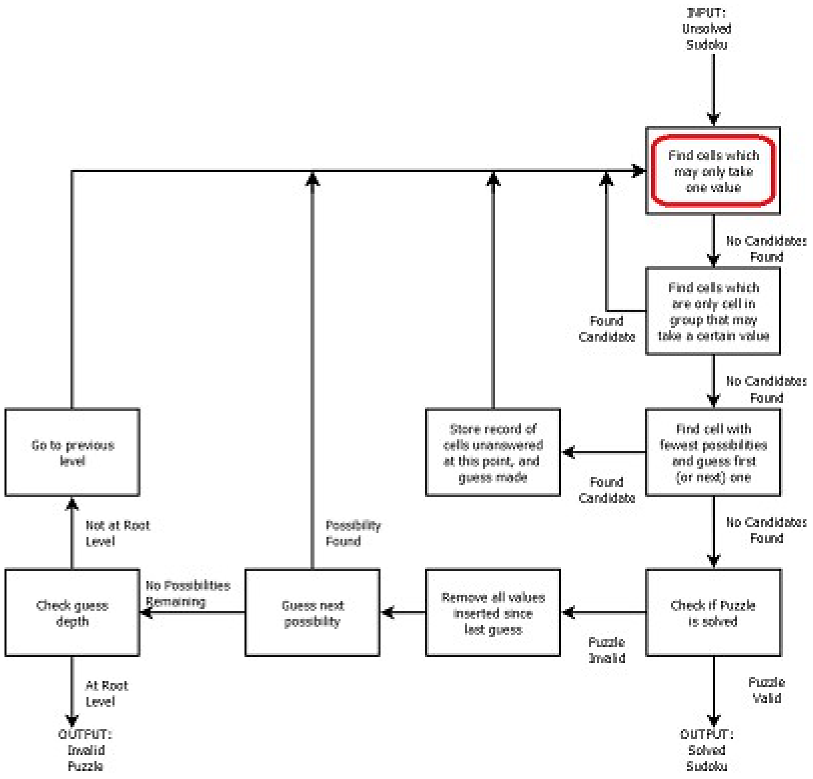
```

    /* int cellPossibBits(point pt);
     *  Returns 9 bit value of values which are not in cell's
     *      row, box, or column.
     *  Returns 0 for filled cells.
     */
int cellPossibBits(point pt) {
    if (Sudoku[pt.y-1][pt.x-1] != 0) { return 0;}
    return 0x1FF ^ (Cols[pt.x-1] | Rows[pt.y-1] | Boxes[ boxNum(pt) ] );
}
```

With only this simple function, the first step in the block diagram may be implemented (note this code is in the previously incompletely defined main function, but omitted there and shown here for clarity.

This part of the main function simply goes through each point and checks for having only 1 possibility. If there is only one possibility, it is inserted. If this is achieved, an 'activity' flag is set to restart the loop to check if new possibilities open up after old ones get filled in:

```
        // Check for columns with only one possibility
        for(pt.y=1; pt.y<=9; pt.y++) {
            for(pt.x=1; pt.x<=9; pt.x++) {
                bits = cellPossibBits(pt);
                if( countBits(bits) == 1) {
                    insertVal(pt, bitsToNum(bits));
                    activity = TRUE;
                }
            }
        }
//      break;
        if( sudokuComplete() ) { break; }
        // Keep doing the easy part until can't do anymore
        // TODO is this actually more efficient?
        if(activity) { continue;}
```

Again, this corresponds to the first step of the block diagram:

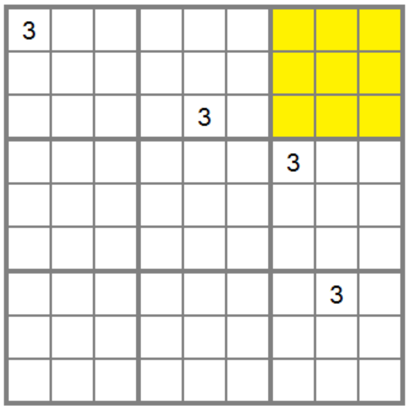

INPUT: Unsolved Sudoku
Find cells which may only take one value
No Candidates Found
Find cells which are only cell in group that may take a certain value
Found Candidate
No Candidates Found
Find cell with fewest possibilities and guess first (or next) one
Found Candidate
Store record of cells unanswered at this point, and guess made
No Candidates Found
Check if Puzzle is solved
Puzzle Valid
OUTPUT: Solved Sudoku
Puzzle Invalid
Remove all values inserted since last guess
Guess next possibility
Possibility Found
No Possibilities Remaining
Check guess depth
Not at Root Level
Go to previous level
At Root Level
OUTPUT: Invalid Puzzle

Now that this step is explained, the functions necessary for the next step may be defined.

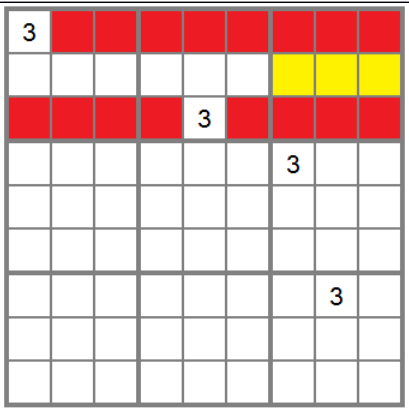
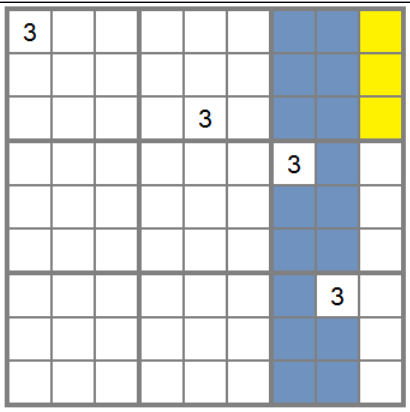
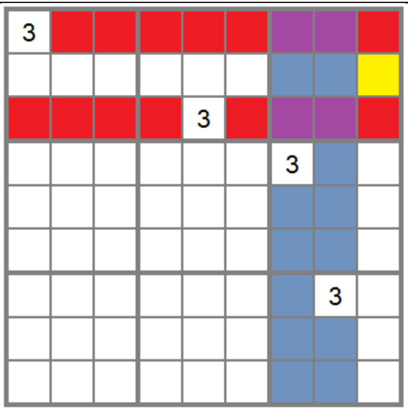
Step 2 Solving Functions: Cells which are only possibility in group for any value

Recall a group is any of a row, column, or box. The next level of logic finds values which are 'exclusive possibilities' in a group, meaning values that may be inserted in **only one** location in this group. When solving Sudoku puzzles by hand, often times this is performed visually with something similar to the below figures:

The rules of Sudoku state that each group (rows, columns, boxes) must contain exactly one of each value (1..9). It can be deduced that the highlighted top right box must contain a 3 (among others):



Knowing that the columns and groups with 3s must contain other 3s, it can be ruled out which other cells may have 3s:

		
Columns already containing a 3	Rows already containing a 3	Both rows and columns shown

The only remaining yellow cell corresponds to the only value that could contain a 3 in that box, therefore it must contain a 3 (recall each group must contain exactly one of each value).

Computationally, this is performed by checking the possibilities of each cell in each group. If a possibility occurs only once, it must occur there.

The below function checks for that by updating two variables for each empty cell of the group (where 'cellBits' is just the possibilities of the current cell):

- redundantBits: values occurring more than once
    - For each empty cell: `redundantBits |= cellBits & bits;`
    - This literally means: set the bits to true if they are found now and were found previously
- bits: values occurring at least once
    - For each empty cell: `bits |= cellBits;`

- This literally means: set bits to true if their value was found

They must be performed in this order to avoid including the ‘previously found bits’ before updating the ‘bits found more than once’. The function is defined below:

```c
       /* int checkExclPossib(char type, char num);
        *    Returns possibilities which occur
        *    exactly once per group.
        */
int checkExclPossib(char type, char num) {
    int bits, redundantBits, cellBits;
    point pt;

    redundantBits = 0;
    bits = 0;

    pt = first(type, num);
    do {
        if(Sudoku[pt.y-1][pt.x-1] != 0) { // cell not empty
            redundantBits |= numToBits(Sudoku[pt.y-1][pt.x-1]);
        } else { // cell empty
            cellBits = cellPossibBits(pt);
            // Values which were here before AND now
            // are occuring more than once
            redundantBits |= cellBits & bits;
            bits |= cellBits;
        }
    } while ( nextPt(&pt, type) );
    // Return values which occured at least once,
    // but not more: (takes values not in redundant bits)
    return bits & (0x1FF ^ redundantBits);
}
```

Note that this function only returns the values which occur, rather than where they occur. Because most often it does not return anything but zero, it would be inefficient to make it return anything related to *where* the value occurs.

To find **where** in each group the values found occur, another function may be declared. This function simply loops through a group and returns the first (and if called because of the result of the previous function, it is the only) occurrence of the inputted value:

```c
    /* point findBitInType(bit,type,pos)
     *  Returns the point of the first occurence of
     *  any number in input 'bit' in group 'type', at 'pos'
     *      e.g.: findBitInType(0b1000, COL, 3)
     *          finds first occurence of 4 in
     *          column 3
     *  NOTE: returns first NUMBER not a bit
     *  Intended to be used with 'checkExclPossib()'
     */
point findBitInType(int bits, char type, char pos) {
    char n;
    point pt = first(type, pos);

    if( bits == 0 ) { return Pt(0,0); }

    do {
        if( (cellPossibBits(pt) & bits) != 0 ) {
            return pt;
        }
    } while ( nextPt(&pt, type) );
    return Pt(0,0);
}
```

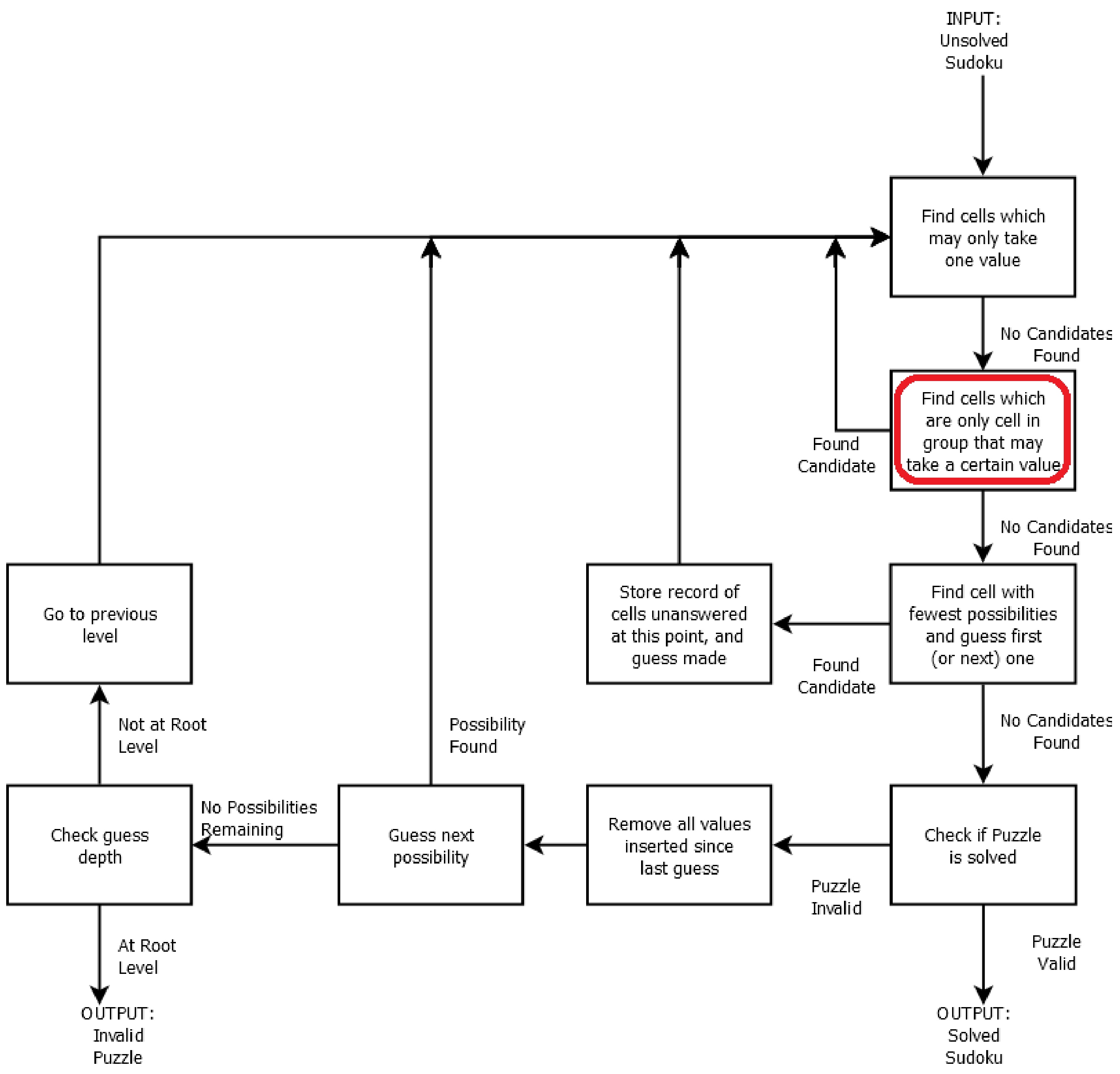
If no value is found, the 'null' point is returned.

Sufficient declarations have been made to now define the second step of the main function.

This step loops through each type of group (which are defined as numbers 0..2) and then each group number, and uses the checkExclPossib() function to check if the number 'num' group has any possibilities which can fit into only one cell of the group. If one is found, the point at which this occurs is located, and the value is inserted. The activity flag is set to go back to easier logic if a value has been inserted (as inserting a value may open up more possibilities).

```
        // ... continued from previous main declaration


        // Check each group for a values that can only fit in one cell

        // Check rows/cols/boxes for exclusive possibilities
        //  (meaning only one cell in this group that can
        //  be value X (where all are tested))
        for(type=0; type<3; type++) { // go through every type
            for(num=1; num<10; num++) {
                exclusiveBits = checkExclPossib(type, num);
                if( exclusiveBits != 0 ) {
                    pt = findBitInType(exclusiveBits, type, num);
                    valBits = bitsToNum(exclusiveBits & cellPossibBits(pt) );
                    insertVal(pt, valBits);
                    activity = TRUE;
                }
            }
        }

        if(activity) { continue; }
        if( sudokuComplete() ) { /*Sudoku complete*/ break; }
```

This corresponds to the step highlighted in the below figure:


INPUT: Unsolved Sudoku
Find cells which may only take one value
No Candidates Found
Find cells which are only cell in group that may take a certain value
Found Candidate
No Candidates Found
Find cell with fewest possibilities and guess first (or next) one
Found Candidate
Store record of cells unanswered at this point, and guess made
No Candidates Found
Check if Puzzle is solved
Puzzle Valid
OUTPUT: Solved Sudoku
Puzzle Invalid
Remove all values inserted since last guess
Guess next possibility
Possibility Found
No Possibilities Remaining
Check guess depth
Not at Root Level
Go to previous level
At Root Level
OUTPUT: Invalid Puzzle

Step 3 Solving Functions: Guessing

At this stage, if the puzzle is not yet solved, this algorithm resorts to guessing. For most Sudoku puzzles, few guesses are required to achieve the solution.

A guess structure is defined to store the necessary data. The fields are:

- `int rowsOn[9];` 9 bits for each row, keeps track of which cells were filled before the guess is made. Note that the bits correspond to location rather than value, in this case.
- `point pt;` The point at which the guess is made.
- `char val;` The value which was guessed.

- `struct _guess * parentGuess;` A pointer pointing to the previous guess. If it is later determined that this guess may take on no value to achieve a solution, the error must be in a previous guess. For the root guess, this is set to zero.

A global variable `CurrentGuess` is defined to refer to the current guess. The `init()` function called in the main function initializes this to zero.

```
struct _guess {
    int rowsOn[9];
    point pt;
    char val;
    struct _guess * parentGuess;
}; typedef struct _guess guess;

guess* CurrentGuess;
```

To make use of these guess structures, the following function:

- calls `malloc()` to allocate space in the memory for a new guess
- sets the `parentGuess` of the new guess to the current `CurrentGuess`
- sets the `CurrentGuess` to this one
- records which cells were filled at the time of guessing, based on location
- sets the `CurrentGuess`'s `val` and `pt`
- inserts the value into the grid with `insertVal()`

```
       /* void makeGuess(point pt, char val);
        * Performs the necessary memory allocation,
        * initialization, and records the appropriate
        * values for a puzzle to be restored to the state
        * it was in before the guess was made
        */
void makeGuess(point pt, char val) {
    int i, mask, row;
    guess* newGuess;
    point ptIns;
    int* rowOn;

    newGuess = malloc( sizeof(guess) );
    newGuess->parentGuess = CurrentGuess;
    CurrentGuess = newGuess;

    // Record which cells were filled before guess
    for(ptIns.y=1; ptIns.y<=9; ptIns.y++) {
        mask = 0x1;
        row = 0;
        for( ptIns.x=1; ptIns.x<=9; ptIns.x++) {
            if( Sudoku[ptIns.y-1][ptIns.x-1] != 0 ) {
                row |= mask;
            }
            mask <<= 1;
        }
        CurrentGuess->rowsOn[ptIns.y-1] = row;
    }

    // Record guess (so next possibility can be tried later if
    // guess doesn't work out)
    CurrentGuess->pt = pt;
    CurrentGuess->val = val;

    // Guess is put into grid
    insertVal(pt, val);
}
```

To determine where to make the guess, a function is defined to find the point with the fewest possibilities. This is a normal minimum finder, modified to immediately return the first occurrence of 2 (as that is the minimum number this function will ever be called to find):

```
    /* point findFewestPossib(void);
     *  Returns the first value with only 2 possibilities,
     *      or the overall minimum.
     *  If no possibilities are found, Pt(0,0) is returned.
     */
point findFewestPossib(void) {
    point pt, minPt;
    char min, temp;
    min = 10;
    minPt = Pt(0,0);
    for(pt.y=1; pt.y<10; pt.y++) {
        for(pt.x=1; pt.x<10; pt.x++) {
            temp = countBits( cellPossibBits(pt) );
            if( temp == 2 ) { return pt; }
            if( temp != 0 && temp < min ) {
                min = temp;
                minPt = pt;
            }
        }
    }
    return minPt;
}
```

To catch the ‘NULL’ point this function returns, another simple function is defined:

```
char  isNull(point pt) {
    return pt.x == 0 && pt.y == 0; // though either one being zero is invalid
}
```

Now that guessing is defined, it is necessary to define a function to go to the next possibility if a guess is determined invalid. However, if the guess has no possibilities remaining, this function sets the current guess to the previous guess, and recursively calls itself until a value of previous guesses is found. If it is necessary to pop out of the root guess, there is an error with the Sudoku puzzle and this function returns false.

```
    /* char nextValidGuess()
     * Increments global variable CurrentGuess to next possibility,
     * popping to higher levels if necessary.
     * Returns TRUE if successful, FALSE if at root guess.
     */
char nextValidGuess() {
    guess* tmpGuess;
    int bits;

    if(CurrentGuess == 0) { return FALSE; }
    clearGuess();
    bits = cellPossibBits(CurrentGuess->pt);
    CurrentGuess->val = bitsToNumMin(bits, CurrentGuess->val);
    if( CurrentGuess->val != 0) {
        insertVal(CurrentGuess->pt, CurrentGuess->val);
        return TRUE;
    } else {
        // Need to pop:
        tmpGuess = CurrentGuess;
        CurrentGuess = CurrentGuess->parentGuess;
        free(tmpGuess);
        return nextValidGuess();
    }

}
```

Notice this function calls the clearGuess function, which simply removes all values added since a guess was made (recall the cells filled in before the guess were stored in the structure):

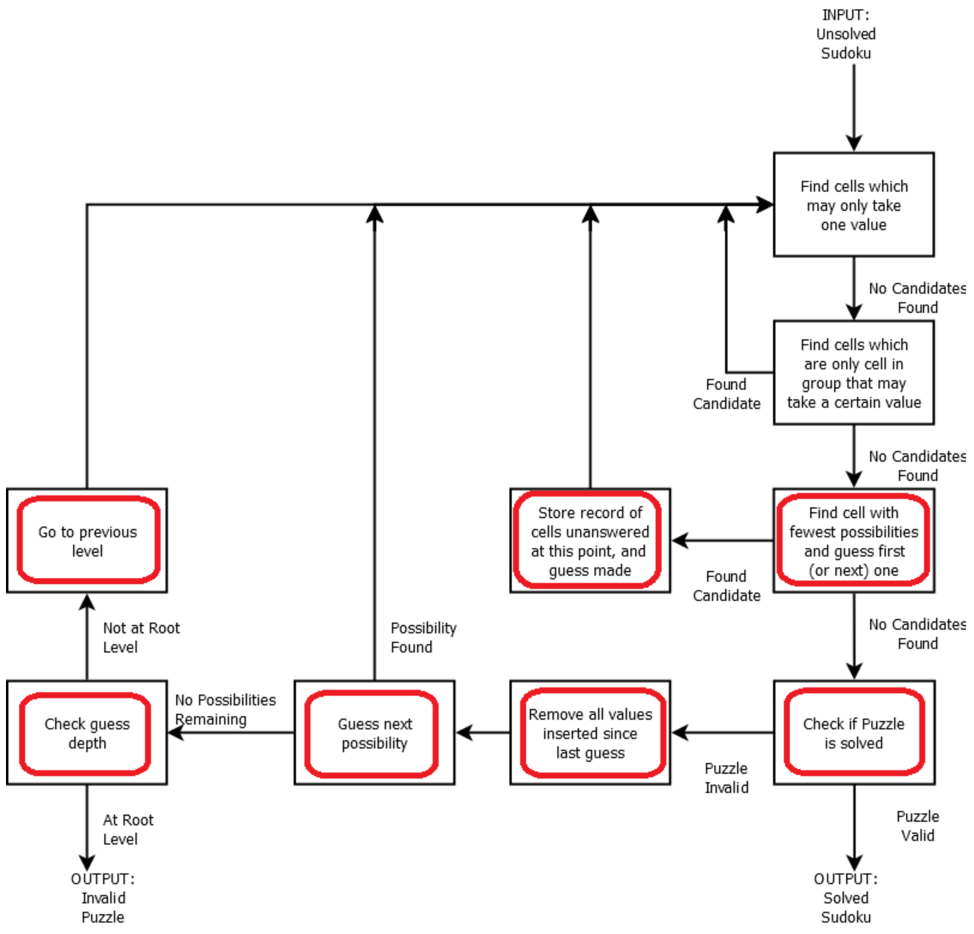
```
      /* void clearGuess(void);
       *    Removes all values inserted since
       *    CurrentGuess was made.
       */
void clearGuess(void) {
    point pt;
    int mask, row;
    for(pt.y=1; pt.y<10; pt.y++) {
        mask = 0x1;
        row = CurrentGuess->rowsOn[pt.y-1];
        for(pt.x=1; pt.x<10; pt.x++) {
            if( (row & mask) == 0 ) {
                removeVal(pt);
            }
            mask <<= 1;
        }
    }
}
```

Finally, the main function may be completed:

```
        // At this point, trivial logic can not help.
        // Resorting to guessing:
```

```
        pt = findFewestPossib(); // returns point with fewest possibilities

                                 // or first with only 2. If finds none,
                                 // returns null point (0,0).
        if( !isNull(pt) ) { // is not null, carry on with guess
            bits = cellPossibBits(pt);
            valBits = bitsToNum(bits);

            makeGuess(pt, valBits);
            activity = TRUE;
        } else {
            // if pt is null, no fillable empty cells remain.
            // This means either sudoku complete, or current guess is
            // invalid.
            if( sudokuComplete() ) { /* Sudoku complete */ break; }

            // At this point, the current guess must be bad.
            // nextValidGuess goes to next possibility, or to parent
            // guess until possibility found. Returns FALSE if no
            // possibilities remain at all (i.e. at root).
            activity = nextValidGuess();
            if(!activity) {
                /* Invalid Sudoku */
                return;
            }
        }

    }

    return;
}// end main
```

This completes the algorithm and the main function:


INPUT: Unsolved Sudoku
Find cells which may only take one value
No Candidates Found
Find cells which are only cell in group that may take a certain value
Found Candidate
No Candidates Found
Find cell with fewest possibilities and guess first (or next) one
Found Candidate
Store record of cells unanswered at this point, and guess made
No Candidates Found
Check if Puzzle is solved
Puzzle Valid
OUTPUT: Solved Sudoku
Puzzle Invalid
Remove all values inserted since last guess
Guess next possibility
Possibility Found
No Possibilities Remaining
Check guess depth
Not at Root Level
Go to previous level
At Root Level
OUTPUT: Invalid Puzzle

Sudoku Solver Test Cases

To test the algorithm, puzzles were inputted and the guesses were tracked. The time it took for the microcontroller to solve them was also recorded.

Indicated puzzles were taken from the Wikipedia article “Sudoku Algorithms”, under “Exceptionally difficult Sudokus”.

Input Puzzle	Guesses	Results

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

- (5,1): 2
  - (2,1): 1
    - (1,2): 2
      - (8,2): 1
      - (8,2): 8
    - (1,2): 7
      - (8,2): 1
      - (8,2): 8
  - (2,1): 7
- (5,1): 7

 * Solved

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

Time: negligible (<0.5 sec)

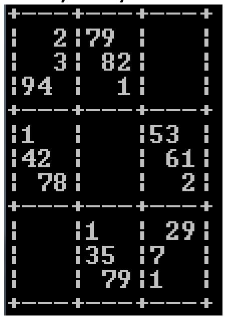
				6	2	9		
		3	7				2	
	2				4			7
	5				7	6		9
2				1				8
3		6	8				5	
4			6				9	
	6				1	7		
		9	4	5				

- (4,1): 1
  - (1,1): 5
  - *Solved

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

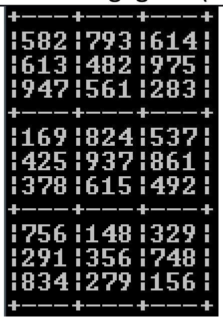
Time: negligible (<0.5 sec)

"Very Easy: #1 in Sudoku book"



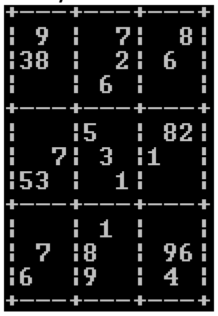
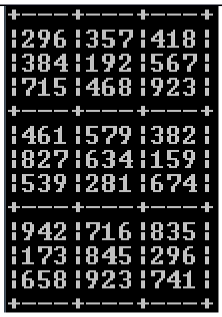
```
+---+---+---+
¦  2¦79 ¦   ¦
¦  3¦ 82¦   ¦
¦94 ¦  1¦   ¦
+---+---+---+
¦1  ¦   ¦53 ¦
¦42 ¦   ¦ 61¦
¦ 78¦   ¦  2¦
+---+---+---+
¦   ¦1  ¦ 29¦
¦   ¦35 ¦7  ¦
¦   ¦ 79¦1  ¦
+---+---+---+
```

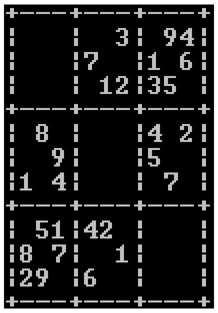
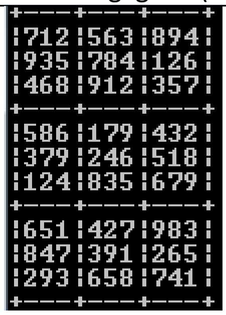
solved with no guessing

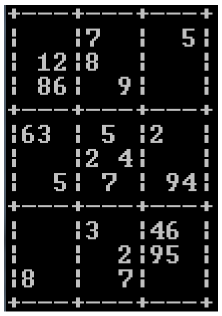
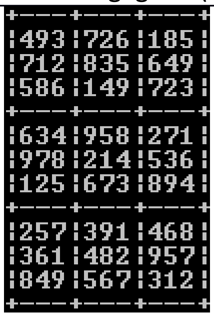


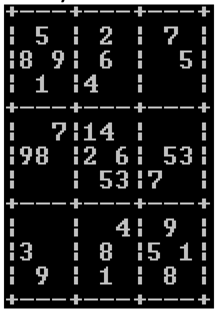
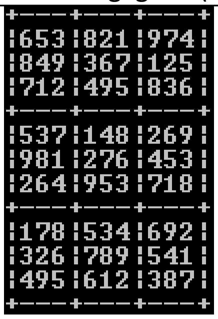
```
+---+---+---+
¦582¦793¦614¦
¦613¦482¦975¦
¦947¦561¦283¦
+---+---+---+
¦169¦824¦537¦
¦425¦937¦861¦
¦378¦615¦492¦
+---+---+---+
¦756¦148¦329¦
¦291¦356¦748¦
¦834¦279¦156¦
+---+---+---+
```

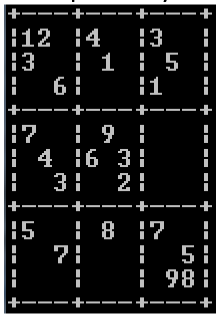
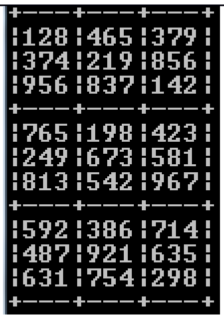
Time: negligible (<0.5 sec)

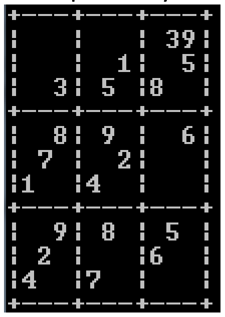
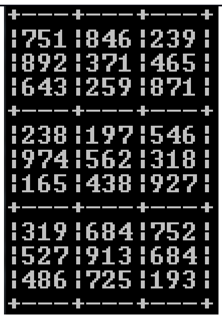
"Easy: #30 in Sudoku book"

`+---+---+---+`
`¦ 9 ¦  7¦  8¦`
`¦38 ¦  2¦ 6 ¦`
`¦   ¦ 6 ¦   ¦`
`+---+---+---+`
`¦   ¦5  ¦ 82¦`
`¦  7¦ 3 ¦1  ¦`
`¦53 ¦  1¦   ¦`
`+---+---+---+`
`¦   ¦ 1 ¦   ¦`
`¦ 7 ¦8  ¦ 96¦`
`¦6  ¦9  ¦ 4 ¦`
`+---+---+---+`
	solved with no guessing	
		`+---+---+---+`
		`¦296¦357¦418¦`
		`¦384¦192¦567¦`
		`¦715¦468¦923¦`
		`+---+---+---+`
		`¦461¦579¦382¦`
		`¦827¦634¦159¦`
		`¦539¦281¦674¦`
		`+---+---+---+`
		`¦942¦716¦835¦`
		`¦173¦845¦296¦`
		`¦658¦923¦741¦`
		`+---+---+---+`
		
		Time: negligible (<0.5 sec)

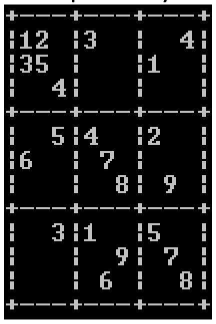
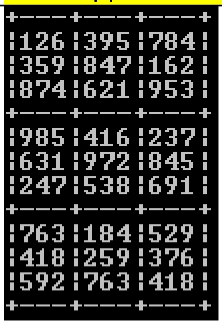
"Medium: #48 in Sudoku book"

`+---+---+---+`
`¦   ¦  3¦ 94¦`
`¦   ¦7  ¦1 6¦`
`¦   ¦ 12¦35 ¦`
`+---+---+---+`
`¦ 8 ¦   ¦4 2¦`
`¦  9¦   ¦5  ¦`
`¦1 4¦   ¦ 7 ¦`
`+---+---+---+`
`¦ 51¦42 ¦   ¦`
`¦8 7¦  1¦   ¦`
`¦29 ¦6  ¦   ¦`
`+---+---+---+`
	(1,1): 5
	(1,1): 7
	*Solved
		
		`+---+---+---+`
		`¦712¦563¦894¦`
		`¦935¦784¦126¦`
		`¦468¦912¦357¦`
		`+---+---+---+`
		`¦586¦179¦432¦`
		`¦379¦246¦518¦`
		`¦124¦835¦679¦`
		`+---+---+---+`
		`¦651¦427¦983¦`
		`¦847¦391¦265¦`
		`¦293¦658¦741¦`
		`+---+---+---+`
		
		Time: negligible (<0.5 sec)

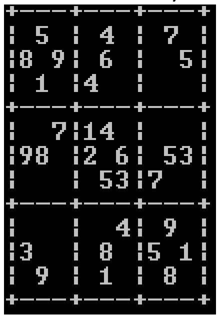
"Hard: #82 in Sudoku book"

`+---+---+---+`
`¦   ¦7  ¦  5¦`
`¦ 12¦8  ¦   ¦`
`¦ 86¦  9¦   ¦`
`+---+---+---+`
`¦63 ¦ 5 ¦2  ¦`
`¦   ¦2 4¦   ¦`
`¦  5¦ 7 ¦ 94¦`
`+---+---+---+`
`¦   ¦3  ¦46 ¦`
`¦   ¦  2¦95 ¦`
`¦8  ¦  7¦   ¦`
`+---+---+---+`
	(2,1): 4
	(1,1): 3
	(1,1): 9
	(2,1): 9
	*Solved
		
		`+---+---+---+`
		`¦493¦726¦185¦`
		`¦712¦835¦649¦`
		`¦586¦149¦723¦`
		`+---+---+---+`
		`¦634¦958¦271¦`
		`¦978¦214¦536¦`
		`¦125¦673¦894¦`
		`+---+---+---+`
		`¦257¦391¦468¦`
		`¦361¦482¦957¦`
		`¦849¦567¦312¦`
		`+---+---+---+`
		
		Time: negligible (<0.5 sec)

"Very Hard: #104 in Sudoku book"

`+---+---+---+`
`¦ 5 ¦ 2 ¦ 7 ¦`
`¦8 9¦ 6 ¦  5¦`
`¦ 1 ¦4  ¦   ¦`
`+---+---+---+`
`¦  7¦14 ¦   ¦`
`¦98 ¦2 6¦ 53¦`
`¦   ¦ 53¦7  ¦`
`+---+---+---+`
`¦   ¦  4¦ 9 ¦`
`¦3  ¦ 8 ¦5 1¦`
`¦ 9 ¦ 1 ¦ 8 ¦`
`+---+---+---+`
	(1,1): 4
	(1,1): 6
	(3,1): 3
	*Solved
		
		`+---+---+---+`
		`¦653¦821¦974¦`
		`¦849¦367¦125¦`
		`¦712¦495¦836¦`
		`+---+---+---+`
		`¦537¦148¦269¦`
		`¦981¦276¦453¦`
		`¦264¦953¦718¦`
		`+---+---+---+`
		`¦178¦534¦692¦`
		`¦326¦789¦541¦`
		`¦495¦612¦387¦`
		`+---+---+---+`
		
		Time: negligible (<0.5 sec)

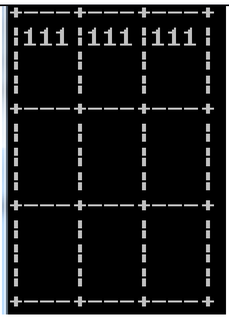
“Exceptionally Difficult Sudokus”

`+---+---+---+`
`¦12 ¦4  ¦3  ¦`
`¦3  ¦ 1 ¦ 5 ¦`
`¦  6¦   ¦1  ¦`
`+---+---+---+`
`¦7  ¦ 9 ¦   ¦`
`¦ 4 ¦6 3¦   ¦`
`¦  3¦  2¦   ¦`
`+---+---+---+`
`¦5  ¦ 8 ¦7  ¦`
`¦  7¦   ¦  5¦`
`¦   ¦   ¦ 98¦`
`+---+---+---+`
	--Incredibly large number of guesses--	
		`+---+---+---+`
		`¦128¦465¦379¦`
		`¦374¦219¦856¦`
		`¦956¦837¦142¦`
		`+---+---+---+`
		`¦765¦198¦423¦`
		`¦249¦673¦581¦`
		`¦813¦542¦967¦`
		`+---+---+---+`
		`¦592¦386¦714¦`
		`¦487¦921¦635¦`
		`¦631¦754¦298¦`
		`+---+---+---+`
		
		Time: approx. 1 minute

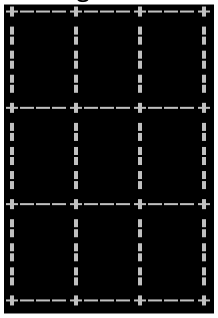
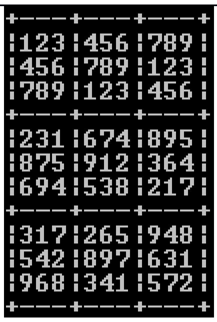
“Exceptionally Difficult Sudokus”

`+---+---+---+`
`¦   ¦   ¦ 39¦`
`¦   ¦  1¦  5¦`
`¦  3¦ 5 ¦8  ¦`
`+---+---+---+`
`¦  8¦ 9 ¦  6¦`
`¦ 7 ¦  2¦   ¦`
`¦1  ¦4  ¦   ¦`
`+---+---+---+`
`¦  9¦ 8 ¦ 5 ¦`
`¦ 2 ¦   ¦6  ¦`
`¦4  ¦7  ¦   ¦`
`+---+---+---+`
	--Incredibly large number of guesses--	
		`+---+---+---+`
		`¦751¦846¦239¦`
		`¦892¦371¦465¦`
		`¦643¦259¦871¦`
		`+---+---+---+`
		`¦238¦197¦546¦`
		`¦974¦562¦318¦`
		`¦165¦438¦927¦`
		`+---+---+---+`
		`¦319¦684¦752¦`
		`¦527¦913¦684¦`
		`¦486¦725¦193¦`
		`+---+---+---+`
		
		Time: approx. 55 seconds

“Exceptionally Difficult Sudokus”

`+---+---+---+`
`¦12 ¦3  ¦  4¦`
`¦35 ¦   ¦1  ¦`
`¦  4¦   ¦   ¦`
`+---+---+---+`
`¦  5¦4  ¦2  ¦`
`¦6  ¦ 7 ¦   ¦`
`¦   ¦  8¦ 9 ¦`
`+---+---+---+`
`¦  3¦1  ¦5  ¦`
`¦   ¦  9¦ 7 ¦`
`¦   ¦ 6 ¦  8¦`
`+---+---+---+`
	--Incredibly large number of guesses--	
		`+---+---+---+`
		`¦126¦395¦784¦`
		`¦359¦847¦162¦`
		`¦874¦621¦953¦`
		`+---+---+---+`
		`¦985¦416¦237¦`
		`¦631¦972¦845¦`
		`¦247¦538¦691¦`
		`+---+---+---+`
		`¦763¦184¦529¦`
		`¦418¦259¦376¦`
		`¦592¦763¦418¦`
		`+---+---+---+`
		
		Time: approx. 35 seconds

Invalid Sudoku (puzzle #104 from before but with point (5,1) changed to 4 to be invalid)

`+---+---+---+`
`¦ 5 ¦ 4 ¦ 7 ¦`
`¦8 9¦ 6 ¦  5¦`
`¦ 1 ¦4  ¦   ¦`
`+---+---+---+`
`¦  7¦14 ¦   ¦`
`¦98 ¦2 6¦ 53¦`
`¦   ¦ 53¦7  ¦`
`+---+---+---+`
`¦   ¦  4¦ 9 ¦`
`¦3  ¦ 8 ¦5 1¦`
`¦ 9 ¦ 1 ¦ 8 ¦`
`+---+---+---+`
	(7,1): 6
	(7,1): 9
	* INVALID *
		INVALID

Blatantly invalid Sudoku	--Incredibly large number of guesses--	The large number of invalid guesses cause this to not finish in a reasonable amount of time. Because it is so


`+---+---+---+`
`¦111¦111¦111¦`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`+---+---+---+`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`+---+---+---+`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`+---+---+---+`
		obvious to the user that this is invalid, this problem is acceptable.

Blank grid

`+---+---+---+`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`+---+---+---+`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`+---+---+---+`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`¦   ¦   ¦   ¦`
`+---+---+---+`
	(1,1): 1
	(2,1): 2
	(3,1): 3
	(4,1): 4
	(5,1): 5
	(6,1): 6
	(7,1): 7
	(8,1): 8
	(1,2): 4
	(2,2): 5
	(3,2): 6
	… 47 guesses without a single pop
		
		`+---+---+---+`
		`¦123¦456¦789¦`
		`¦456¦789¦123¦`
		`¦789¦123¦456¦`
		`+---+---+---+`
		`¦231¦674¦895¦`
		`¦875¦912¦364¦`
		`¦694¦538¦217¦`
		`+---+---+---+`
		`¦317¦265¦948¦`
		`¦542¦897¦631¦`
		`¦968¦341¦572¦`
		`+---+---+---+`
		
		Time: negligible (<0.5 sec)

References

Wikipedia contributors. Sudoku algorithms [Internet]. Wikipedia, The Free Encyclopedia; 2012 Mar 26, 21:33 UTC [cited 2012 Apr 2]. Available from: http://en.wikipedia.org/w/index.php?title=Sudoku_algorithms&oldid=484078096.