The Mathematica® Journal

A Beginner's Guide to Solving Sudoku Puzzles by Computer

Robert Cowen

We simultaneously introduce effective techniques for solving Sudoku puzzles and explain how to implement them in Mathematica. The hardest puzzles require some guessing, and we include a simple backtracking technique that solves even the hardest puzzles. The programming skills required are kept at a minimum.

Introduction to Sudoku

Sudoku, for those unfamiliar with this puzzle, consists of a 9×9 square grid with nine 3×3 subgrids. The 81 entries are to be filled with the integers 1 to 9 in such a way that each row, column and subgrid contains all the nine integers. Some of the entries are already chosen, and the final puzzle solution must contain these initial choices. Here is a sample puzzle.

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

First Steps to Solving Sudoku

The input for this puzzle is a list of nine lists consisting of blanks (shown as □) or integers between 1 and 9. A list of lists of the same length is regarded as a matrix in Mathematica, so we input b for the puzzle and then show it in matrix form.

```
b = {{□, 3, □, □, □, □, □, 7, □}, {□, 1, □, 8, □, 6, □, 2, □},
    {□, □, 9, 1, 7, 5, 4, □, □}, {□, □, 7, □, □, □, 6, □, □},
    {2, □, □, 4, □, 9, □, □, 3}, {□, □, 3, □, □, □, 1, □, □},
    {□, □, 1, 5, 4, 8, 2, □, □}, {□, 6, □, 3, □, 2, □, 1, □},
    {□, 5, □, □, □, □, □, 4, □}};
```

```
MatrixForm[b]
```

$$\begin{pmatrix} □ & 3 & □ & □ & □ & □ & □ & 7 & □ \\ □ & 1 & □ & 8 & □ & 6 & □ & 2 & □ \\ □ & □ & 9 & 1 & 7 & 5 & 4 & □ & □ \\ □ & □ & 7 & □ & □ & □ & 6 & □ & □ \\ 2 & □ & □ & 4 & □ & 9 & □ & □ & 3 \\ □ & □ & 3 & □ & □ & □ & 1 & □ & □ \\ □ & □ & 1 & 5 & 4 & 8 & 2 & □ & □ \\ □ & 6 & □ & 3 & □ & 2 & □ & 1 & □ \\ □ & 5 & □ & □ & □ & □ & □ & 4 & □ \end{pmatrix}$$

We can also display this in Sudoku format by drawing column and row lines and a frame.

```
display[X_] := Grid[
  Map[
   If[ListQ@#, Row[#], #] &, X, {2}
  ],
  GridFrame → True,
  RowLines → {False, False, True, False, False, True,
    False, False},
  ColumnLines → {False, False, True, False, False, True,
    False, False}]
```

```
display[b]
```

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

In attempting a solution, a blank gets replaced with a list of candidate entries shown compactly without braces or commas.

```
Row[{4, 5, 6, 8}]
```

```
4568
```

Each element of a Sudoku matrix `X` is obtainable as `X[[i, j]]`, where `i` and `j` are the row and column of the element in the matrix.

```
b[[1, 2]]
```

```
3
```

```
b[[1, 1]]
```

```
□
```

To obtain the entire row in `X` that contains the entry `X[[i, j]]`, we evaluate `X[[i]]`; so, for example, this gives row 3, which contains element `b[[3, 5]]`.

```
b[[3]]
```

```
{□, □, 9, 1, 7, 5, 4, □, □}
```

To obtain the column in `X` that contains `X[[i, j]]` is a little trickier. One could first "transpose" the matrix `X`, that is, interchange rows and columns and then find row `j` of the transposed matrix; the command for transposing a matrix is `Transpose`. However, Mathematica has a faster way using the option `All`. We just enter `X[[All, j]]` to get column `j` of `X`. For example, the column that contains `b[[3, 5]]`, that is, column 5 of `b`, can be obtained by entering `b[[All, 5]]`.

```
b[[All, 5]]
```

```
{□, □, 7, □, □, □, 4, □, □}
```

The function `Column` displays that list vertically.

```
Column@b[[All, 5]]
```

```
□
□
7
□
□
□
4
□
□
```

It is more difficult to obtain the block to which `X[[i, j]]` belongs. To do this we define a function `block[X, i, j]` that gives a list of the entries that comprise the block of `X[[i, j]]` in `X`.

```
block[X_, i_, j_] := Which[
  1 ≤ i ≤ 3 && 1 ≤ j ≤ 3, Take[X, {1, 3}, {1, 3}],
  4 ≤ i ≤ 6 && 1 ≤ j ≤ 3, Take[X, {4, 6}, {1, 3}],
  7 ≤ i ≤ 9 && 1 ≤ j ≤ 3, Take[X, {7, 9}, {1, 3}],
  1 ≤ i ≤ 3 && 4 ≤ j ≤ 6, Take[X, {1, 3}, {4, 6}],
  4 ≤ i ≤ 6 && 4 ≤ j ≤ 6, Take[X, {4, 6}, {4, 6}],
  7 ≤ i ≤ 9 && 4 ≤ j ≤ 6, Take[X, {7, 9}, {4, 6}],
  1 ≤ i ≤ 3 && 7 ≤ j ≤ 9, Take[X, {1, 3}, {7, 9}],
  4 ≤ i ≤ 6 && 7 ≤ j ≤ 9, Take[X, {4, 6}, {7, 9}],
  7 ≤ i ≤ 9 && 7 ≤ j ≤ 9, Take[X, {7, 9}, {7, 9}]
 ]
```

For example, this is the block containing `b[[1, 6]]`, the sixth entry in row 1, □.

```
block[b, 1, 6]
```

```
{{□, □, □}, {8, □, 6}, {1, 7, 5}}
```

```
MatrixForm[block[b, 1, 6]]
```

$$\begin{pmatrix} □ & □ & □ \\ 8 & □ & 6 \\ 1 & 7 & 5 \end{pmatrix}$$

To get a single list of these entries by removing the inner parentheses, we use `Flatten`.

```
Flatten[block[b, 1, 6]]
```

```
{□, □, □, 8, □, 6, 1, 7, 5}
```

Our first step is to replace each □ in `b` (using `ReplaceAll`) by the list of the nine numbers, `{1, 2, ... , 9}`, which are possible candidates to occupy that position in `b`.

```
c = ReplaceAll[b, □ → Range[9]]
```

```
{{{1, 2, 3, 4, 5, 6, 7, 8, 9}, 3,
   {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
   {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
   {1, 2, 3, 4, 5, 6, 7, 8, 9}, 7, {1, 2, 3, 4, 5, 6, 7, 8, 9}},
  {{1, 2, 3, 4, 5, 6, 7, 8, 9}, 1, {1, 2, 3, 4, 5, 6, 7, 8, 9},
   8, {1, 2, 3, 4, 5, 6, 7, 8, 9}, 6,
   {1, 2, 3, 4, 5, 6, 7, 8, 9}, 2, {1, 2, 3, 4, 5, 6, 7, 8, 9}},
  {{1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
   9, 1, 7, 5, 4, {1, 2, 3, 4, 5, 6, 7, 8, 9},
```

```
 {1, 2, 3, 4, 5, 6, 7, 8, 9}}, {{1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, 7, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9}, 6,
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9}},
 {2, {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  4, {1, 2, 3, 4, 5, 6, 7, 8, 9}, 9,
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9}, 3},
 {{1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  3, {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, 1, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 2, 3, 4, 5, 6, 7, 8, 9}}, {{1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, 1, 5, 4, 8, 2,
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9}},
 {{1, 2, 3, 4, 5, 6, 7, 8, 9}, 6, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  3, {1, 2, 3, 4, 5, 6, 7, 8, 9}, 2, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  1, {1, 2, 3, 4, 5, 6, 7, 8, 9}}, {{1, 2, 3, 4, 5, 6, 7, 8, 9},
  5, {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 2, 3, 4, 5, 6, 7, 8, 9}, 4, {1, 2, 3, 4, 5, 6, 7, 8, 9}}}
```

`display[c]`

12345 6789	3	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	7	12345 6789
12345 6789	1	12345 6789	8	12345 6789	6	12345 6789	2	12345 6789
12345 6789	12345 6789	9	1	7	5	4	12345 6789	12345 6789
12345 6789	12345 6789	7	12345 6789	12345 6789	12345 6789	6	12345 6789	12345 6789
2	12345 6789	12345 6789	4	12345 6789	9	12345 6789	12345 6789	3
12345 6789	12345 6789	3	12345 6789	12345 6789	12345 6789	1	12345 6789	12345 6789
12345 6789	12345 6789	1	5	4	8	2	12345 6789	12345 6789
12345 6789	6	12345 6789	3	12345 6789	2	12345 6789	1	12345 6789
12345 6789	5	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	4	12345 6789

Our next task is to start eliminating candidate values in the entries that are lists of numbers in `X`, proceeding one entry `X[[i, j]]` at a time. We start with `A = X` in order to be able to redefine entries.

Since no entry can appear more than once in any row, column or block, we let `B` be the set of integers in the row, column and block containing `A[[i, j]]`.

If the entry `A[[i, j]]` is a list rather than an integer, we redefine `A[[i, j]]` by removing the entries that also belong to `B`.

Finally, if `A[[i, j]]` is a list of one element, we redefine it to be that element.

```
DeleteSingletonsFromLists[X_] := Module[{A = X, integers},
  Table[
   integers = Select[
     Join[A[[i]], A[[All, j]], Flatten[block[A, i, j], 1]],
     IntegerQ[#] &
    ];
   If[ListQ[A[[i, j]]],
    A[[i, j]] = Complement[A[[i, j]], integers]];
   If[Length[A[[i, j]]] == 1, A[[i, j]] = First[A[[i, j]]]],
   {i, 9}, {j, 9}];
  A]
```

```
DeleteSingletonsFromLists[c] // display
```

4568	3	24568	29	29	4	589	7	15689
457	1	45	8	39	6	359	2	59
68	28	9	1	7	5	4	368	68
14589	489	7	2	1358	13	6	589	4589
2	8	56	4	156	9	57	5	3
4569	49	3	67	568	7	1	89	2489
379	79	1	5	4	8	2	369	679
4789	6	48	3	9	2	578	1	578
3789	5	28	67	16	1	3789	4	6789

To apply `DeleteSingletonsFromLists` again and again to `c` until the result no longer changes, we use `FixedPoint`. The puzzle simplifies, but we see that we are still not done!

```
(d = FixedPoint[DeleteSingletonsFromLists, c]) // display
```

568	3	58	9	2	4	58	7	1568
457	1	45	8	3	6	59	2	59
68	2	9	1	7	5	4	368	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

However, the first block has three entries (colored red) that are all sublists of `{5, 6, 8}`.

568	3	58	9	2	4	58	7	1568
457	1	45	8	3	6	59	2	59
68	2	9	1	7	5	4	368	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

;

While we do not know the exact value of any of the red entries, we know that the three numbers 5, 6 and 8 will be used up filling them; thus we can remove 5, 6 and 8 from the *other* entries in this block (colored green).

568	3	58	9	2	4	58	7	1568
47	1	4	8	3	6	59	2	59
68	2	9	1	7	5	4	368	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

;

Similarly, in the first row, there are three entries that are sublists of `{5, 6, 8}`, so we remove 5, 6 and 8 from `{1, 5, 6, 8}` at the end of row 1; this defines `e`.

568	3	58	9	2	4	58	7	1568
47	1	4	8	3	6	59	2	59
68	2	9	1	7	5	4	368	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

;

```
(e = {{{5, 6, 8}, 3, {5, 8}, 9, 2, 4, {5, 8}, 7, 1},
    {{4, 7}, 1, 4, 8, 3, 6, {5, 9}, 2, {5, 9}},
    {{6, 8}, 2, 9, 1, 7, 5, 4, {3, 6, 8}, {6, 8}},
    {{1, 4, 5, 9}, {4, 9}, 7, 2, {5, 8}, 3, 6, {8, 9},
     {4, 8, 9}}, {2, 8, 6, 4, 1, 9, 7, 5, 3},
    {{4, 5, 9}, {4, 9}, 3, 6, {5, 8}, 7, 1, {8, 9},
     {2, 4, 8, 9}}, {{3, 7, 9}, {7, 9}, 1, 5, 4, 8, 2,
     {3, 6, 9}, {6, 7, 9}},
    {{4, 7, 8}, 6, {4, 8}, 3, 9, 2, {5, 8}, 1, {5, 7, 8}},
    {{3, 8, 9}, 5, {2, 8}, 7, 6, 1, {3, 8, 9}, 4, {8, 9}}}) //
 display
```

568	3	58	9	2	4	58	7	1
47	1	4	8	3	6	59	2	59
68	2	9	1	7	5	4	368	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

Then we use `FixedPoint` again and display the result. We are done! We explore this technique further in the next section.

```
FixedPoint[DeleteSingletonsFromLists, e] // display
```

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

Sudoku Twins and Multiples

If any row, column or block contains the pair {a, b} twice, both a and b must be used up in the two entries containing {a, b}, even though we do not know which pair contains a and which contains b. Hence, no other entry in that row, column or block can contain either a or b. This obvious fact is surprisingly useful in solving Sudoku puzzles.

To use it, we define the function Twin.

1. We select the set of pairs (the lists of length two).

2. The twins are the identical pairs.

3. The numbers in the twins are the numbers to prune.

4. The lists are pruned.

5. Any singleton list is changed to its element.

```
Twin[X_] :=
 Module[{pairs, twins, numberstoprune, prunedlists},
  pairs = Select[X, Length[#] == 2 &];
  twins = Select[Subsets[pairs, {2}], #[[1]] == #[[2]] &, 1];
  numberstoprune = Union[Flatten[twins]];
  prunedlists =
   If[Length[#] > 1 && # ≠ numberstoprune,
      Complement[#, numberstoprune], #] & /@ X;
  If[Length[#] == 1, First[#], #] & /@ prunedlists
 ]
```

Here are some examples using d, which was defined before. The twins in this row are {5, 8} and {5, 8}.

```
d[[1]]
```

```
{{5, 6, 8}, 3, {5, 8}, 9, 2, 4, {5, 8}, 7, {1, 5, 6, 8}}
```

Hence 5 and 8 are removed from the other lists in the row.

```
Twin[d[[1]]]
```

```
{6, 3, {5, 8}, 9, 2, 4, {5, 8}, 7, {1, 6}}
```

The twins in row 3 are {6, 8} and {6, 8}.

```
d[[3]]
```

```
{{6, 8}, 2, 9, 1, 7, 5, 4, {3, 6, 8}, {6, 8}}
```

```
Twin[d[[3]]]
```

```
{{6, 8}, 2, 9, 1, 7, 5, 4, 3, {6, 8}}
```

We can map Twin over all the rows of a matrix.

```
TwinRow[X_] := Twin /@ X
```

```
(f = Twin /@ d) // display
```

6	3	58	9	2	4	58	7	16
47	1	4	8	3	6	59	2	59
68	2	9	1	7	5	4	3	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

We now use DeleteSingletonsFromLists starting with f until the result does not change.

```
FixedPoint[DeleteSingletonsFromLists, f] // display
```

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

We are done. It was only necessary to use Twin on the rows.

It is easy to apply `Twin` on the columns: transpose, apply `Twin` to the rows, and transpose back.

```
TwinColumn[X_] := Transpose[TwinRow[Transpose[X]]]
```

The blocks are more complicated. We make use of a general theorem: transform an $n^2 \times n^2$ matrix M by taking the n^2 elements of each block in order as the rows of a new $n^2 \times n^2$ matrix BT(M); then BT(BT(M)) = M (i.e. BT is an involution). Here BT stands for block transpose.

This verifies the theorem in the 9×9 case.

```
M = Table[m_{i,j}, {i, 9}, {j, 9}]; MatrixForm[M]
```

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & m_{1,5} & m_{1,6} & m_{1,7} & m_{1,8} & m_{1,9} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & m_{2,5} & m_{2,6} & m_{2,7} & m_{2,8} & m_{2,9} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} & m_{3,5} & m_{3,6} & m_{3,7} & m_{3,8} & m_{3,9} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & m_{4,5} & m_{4,6} & m_{4,7} & m_{4,8} & m_{4,9} \\ m_{5,1} & m_{5,2} & m_{5,3} & m_{5,4} & m_{5,5} & m_{5,6} & m_{5,7} & m_{5,8} & m_{5,9} \\ m_{6,1} & m_{6,2} & m_{6,3} & m_{6,4} & m_{6,5} & m_{6,6} & m_{6,7} & m_{6,8} & m_{6,9} \\ m_{7,1} & m_{7,2} & m_{7,3} & m_{7,4} & m_{7,5} & m_{7,6} & m_{7,7} & m_{7,8} & m_{7,9} \\ m_{8,1} & m_{8,2} & m_{8,3} & m_{8,4} & m_{8,5} & m_{8,6} & m_{8,7} & m_{8,8} & m_{8,9} \\ m_{9,1} & m_{9,2} & m_{9,3} & m_{9,4} & m_{9,5} & m_{9,6} & m_{9,7} & m_{9,8} & m_{9,9} \end{pmatrix}$$

To construct the new kind of transposed matrix, we define the function `BlockTranspose`.

```
BlockTranspose[x_] := {
  Flatten[block[x, 1, 1], 1],
  Flatten[block[x, 1, 4], 1],
  Flatten[block[x, 1, 7], 1],
  Flatten[block[x, 4, 1], 1],
  Flatten[block[x, 4, 4], 1],
  Flatten[block[x, 4, 7], 1],
  Flatten[block[x, 7, 1], 1],
  Flatten[block[x, 7, 4], 1],
  Flatten[block[x, 7, 7], 1]
 }
```

Here is the transformed matrix `BlockTranspose[M]`.

```
BlockTranspose[M] // MatrixForm
```

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{2,1} & m_{2,2} & m_{2,3} & m_{3,1} & m_{3,2} & m_{3,3} \\ m_{1,4} & m_{1,5} & m_{1,6} & m_{2,4} & m_{2,5} & m_{2,6} & m_{3,4} & m_{3,5} & m_{3,6} \\ m_{1,7} & m_{1,8} & m_{1,9} & m_{2,7} & m_{2,8} & m_{2,9} & m_{3,7} & m_{3,8} & m_{3,9} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{5,1} & m_{5,2} & m_{5,3} & m_{6,1} & m_{6,2} & m_{6,3} \\ m_{4,4} & m_{4,5} & m_{4,6} & m_{5,4} & m_{5,5} & m_{5,6} & m_{6,4} & m_{6,5} & m_{6,6} \\ m_{4,7} & m_{4,8} & m_{4,9} & m_{5,7} & m_{5,8} & m_{5,9} & m_{6,7} & m_{6,8} & m_{6,9} \\ m_{7,1} & m_{7,2} & m_{7,3} & m_{8,1} & m_{8,2} & m_{8,3} & m_{9,1} & m_{9,2} & m_{9,3} \\ m_{7,4} & m_{7,5} & m_{7,6} & m_{8,4} & m_{8,5} & m_{8,6} & m_{9,4} & m_{9,5} & m_{9,6} \\ m_{7,7} & m_{7,8} & m_{7,9} & m_{8,7} & m_{8,8} & m_{8,9} & m_{9,7} & m_{9,8} & m_{9,9} \end{pmatrix}$$

Finally, we look at the matrix `BlockTranspose[BlockTranspose[M]]`.

```
BlockTranspose[BlockTranspose[M]] // MatrixForm
```

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & m_{1,5} & m_{1,6} & m_{1,7} & m_{1,8} & m_{1,9} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & m_{2,5} & m_{2,6} & m_{2,7} & m_{2,8} & m_{2,9} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} & m_{3,5} & m_{3,6} & m_{3,7} & m_{3,8} & m_{3,9} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & m_{4,5} & m_{4,6} & m_{4,7} & m_{4,8} & m_{4,9} \\ m_{5,1} & m_{5,2} & m_{5,3} & m_{5,4} & m_{5,5} & m_{5,6} & m_{5,7} & m_{5,8} & m_{5,9} \\ m_{6,1} & m_{6,2} & m_{6,3} & m_{6,4} & m_{6,5} & m_{6,6} & m_{6,7} & m_{6,8} & m_{6,9} \\ m_{7,1} & m_{7,2} & m_{7,3} & m_{7,4} & m_{7,5} & m_{7,6} & m_{7,7} & m_{7,8} & m_{7,9} \\ m_{8,1} & m_{8,2} & m_{8,3} & m_{8,4} & m_{8,5} & m_{8,6} & m_{8,7} & m_{8,8} & m_{8,9} \\ m_{9,1} & m_{9,2} & m_{9,3} & m_{9,4} & m_{9,5} & m_{9,6} & m_{9,7} & m_{9,8} & m_{9,9} \end{pmatrix}$$

It is the same as the original matrix `M`; here is a direct check that they are equal.

```
BlockTranspose[BlockTranspose[M]] == M
```

```
True
```

It is a common technique in problem solving to first transform the problem, solve the transformed problem and then transform back. As an example, we apply `BlockTranspose`, followed by `Twin`, followed by `BlockTranspose`, to the matrix `f` defined earlier; blocks (1, 3), (2, 1) and (2, 3) change.

```
display[f]
```

6	3	58	9	2	4	58	7	16
47	1	4	8	3	6	59	2	59
68	2	9	1	7	5	4	3	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

```
BlockTranspose[Twin /@ BlockTranspose[f]] // display
```

6	3	58	9	2	4	8	7	16
47	1	4	8	3	6	59	2	59
68	2	9	1	7	5	4	3	68
15	49	7	2	58	3	6	89	4
2	8	6	4	1	9	7	5	3
5	49	3	6	58	7	1	89	24
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

This makes a function out of that line of code.

```
TwinBlock[X_] := BlockTranspose[Twin /@ BlockTranspose[X]]
```

The function `TwinSolver` puts together the discussion so far.

```
TwinSolver[X_] := FixedPoint[
  TwinBlock@
    TwinColumn@
     TwinRow@FixedPoint[DeleteSingletonsFromLists, #] &, X]
```

We apply it to the matrix `f` and get a solution as before.

```
TwinSolver[f] // display
```

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

We generalize the function `Twin` to `Multiple` to deal with triples and quadruplets as well as twins.

```
Multiple[X_, n_] := Module[
  {ntuples, multiplets, numberstoprune, prunedlists},
  ntuples = Select[X, 1 < Length[#] ≤ n &];
  multiplets = Select[Subsets[ntuples, {n}],
    Length[Union[Flatten[#]]] == n &];
  numberstoprune =
   Union[Flatten[If[{} ≠ multiplets, First[multiplets],
       {}]]];
  prunedlists =
   (If[Length[#1] > 1 &&
         ! MemberQ[If[multiplets == {}, {}, First[multiplets]],
           #1], Complement[#1, numberstoprune], #1] &) /@ X;
  If[Length[#] == 1, First[#], #] & /@ prunedlists
 ]
```

Just as with `Twin`, we want to use `Multiple` on rows, columns and blocks of a matrix and then combine them in `MultipleSolver`.

```
MultipleRow[X_, n_] := Multiple[#, n] & /@ X
```

```
MultipleColumn[X_, n_] :=
 Transpose[Multiple[#, n] & /@ Transpose[X]]
```

```
MultipleBlock[X_, n_] :=
 BlockTranspose[Multiple[#, n] & /@ BlockTranspose[X]]

MultipleSolver[X_, n_] :=
 FixedPoint[
  MultipleBlock[#, n] & @MultipleColumn[#, n] & @
      MultipleRow[#, n] & @
    FixedPoint[DeleteSingletonsFromLists, #] &, X]
```

We had already solved `f` with `TwinSolver`, but let us apply `MultipleSolver` for triples as a check.

```
display[f]
```

6	3	58	9	2	4	58	7	16
47	1	4	8	3	6	59	2	59
68	2	9	1	7	5	4	3	68
1459	49	7	2	58	3	6	89	489
2	8	6	4	1	9	7	5	3
459	49	3	6	58	7	1	89	2489
379	79	1	5	4	8	2	369	679
478	6	48	3	9	2	58	1	578
389	5	28	7	6	1	389	4	89

```
MultipleSolver[f, 3] // display
```

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

`MasterSolver` combines the three solvers into one.

```
MasterSolver[X_] :=
 DeleteSingletonsFromLists@MultipleSolver[#, 2] &@
      MultipleSolver[#, 3] &@MultipleSolver[#, 4] &@
  ReplaceAll[X, □ → Range[9]]
```

Consider the puzzle `g`.

```
(g = {{5, □, □, □, 6, 3, □, □, 4}, {□, 7, □, 1, 5, □, 6, 8, □},
     {□, □, □, □, □, □, □, □, □}, {□, □, □, □, □, □, □, 6, 1},
     {□, □, 2, □, 3, □, 4, □, □}, {6, 9, □, □, □, □, □, □, □},
     {□, □, □, □, □, □, □, □, □}, {□, 5, 3, □, 7, 6, □, 2, □},
     {1, □, □, 5, 4, □, □, □, 8}}) // display
```

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

Unfortunately, `MasterSolver` does not solve the puzzle.

```
(g1 = MasterSolver[g]) // display
```

5	18	18	27	6	3	279	79	4
2	7	4	1	5	9	6	8	3
9	3	6	247	28	478	27	157	257
3	4	578	279	28	578	2789	6	1
78	18	2	6	3	578	4	579	57
6	9	578	247	1	4578	2378	357	257
78	2	78	3	9	1	5	4	6
4	5	3	8	7	6	1	2	9
1	6	9	5	4	2	37	37	8

However, there are entries that are pairs.

```
Select[Flatten[g1, 1], Length[#] == 2 &]
```

```
{{1, 8}, {1, 8}, {2, 7}, {7, 9}, {2, 8}, {2, 7}, {2, 8},
 {7, 8}, {1, 8}, {5, 7}, {7, 8}, {7, 8}, {3, 7}, {3, 7}}
```

We propose to replace the pair `{1, 8}` by 1 and to try to solve the modified puzzle; if that leads to a contradiction, then `g1[[1, 2]] = 8`.

We introduce the functions `GoLeft` and `GoRight` via the helper function `GoAux`. If there are any pairs in `X`, `GoLeft[X]` replaces the first such pair `{u, v}` with its left entry `u` and applies `MasterSolver`; the function `GoRight` replaces `{u, v}` with its right entry `v`.

```
GoAux[X_, leftorright_] := Module[
  {firstpair = SelectFirst[Flatten[X, 1], Length@# == 2 &]},
  MasterSolver@If[
    2 ≠ Length@firstpair, (* there was no pair *)
    X,
    ReplacePart[X, First@Position[X, firstpair] →
      If[leftorright === Left, First, Last]@firstpair]
   ]
 ]

GoLeft[X_] := GoAux[X, Left]

GoRight[X_] := GoAux[X, Right]
```

The two blank entries indicate a contradiction.

```
GoLeft[g1] // display
```

5	1	8	2	6	3	9	7	4
2	7	4	1	5	9	6	8	3
9	3	6	7	8	4	2	1	5
3	4	5	9	2	7	8	6	1
7	8	2	6	3	5	4	9	
6	9		4	1	8	3	5	27
8	2	7	3	9	1	5	4	6
4	5	3	8	7	6	1	2	9
1	6	9	5	4	2	7	3	8

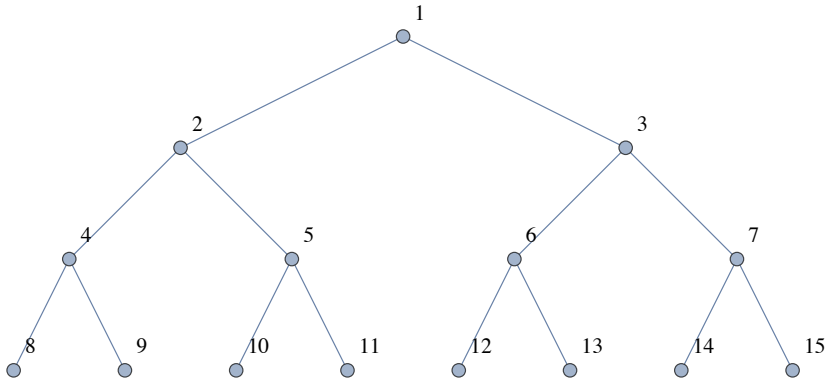
Therefore, the alternative `g1[[1, 2]] = 8` must solve the puzzle, and it does.

```
GoRight[g1] // display
```

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

Backtracking

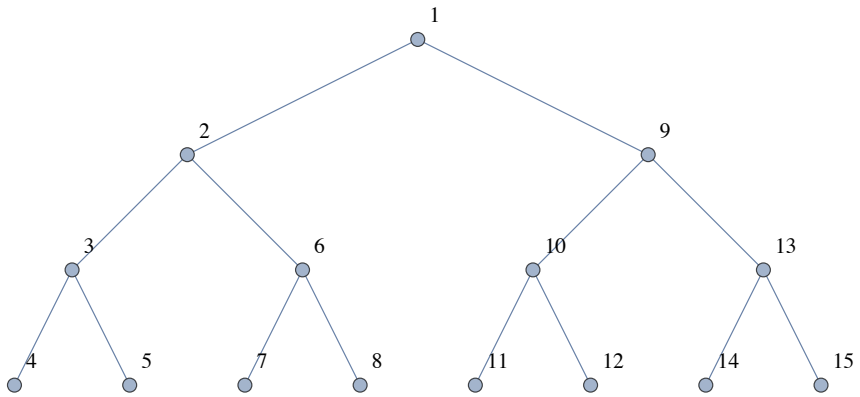
We have just seen that guessing between two alternatives quickly led us to a solution. However, if a solution was not obtained with the first alternative, it might be necessary to again guess between two alternatives, and so on. If there are always just two alternatives, this leads to a binary tree with the root at the top.



It is not clear how many levels or guesses are needed before reaching a solution. Also, it may not be necessary to generate the entire tree before a solution is reached. There is a systematic and efficient way to search such a tree, usually referred to as backtracking.

Here is the method: start at the root and go left as long as there is no contradiction. If there is a contradiction, go back one level and go right. Then resume going left as far as possible. If there is a contradiction after going right, go back through all the right branches traversed so far; then go back through an additional left branch and go right.

For example, assume that contradictions exist at all nodes on level 4 except for the last one, node 15. The labels in the following tree indicate how to backtrack to the solution at node 15.



The binary choice in the Sudoku situation is to go either left or right. A path through the tree corresponds to a sequence of such choices; for example, the path (1, 2, 6, 8) generates the sequence: {`Left, Right, Right`}, from which a composition of functions can be built.

Here is how the built-in function `RightComposition` works with undefined functions.

```
(RightComposition @@ {F, G, H})[x]
```

```
H[G[F[x]]]
```

This example shows a clear contradiction, since there are two blank entries.

```
(g2 = (RightComposition @@ {GoLeft, GoRight, GoLeft})[g1]) //
 display
```

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

The function `BacktrackSequence`, when given a nonempty sequence of `GoLeft` and `GoRight` functions, drops the last `GoRight` in the sequence (if any) until there are none, drops the last `GoLeft`, and finally appends a `GoRight`.

```
BacktrackSequence[x_] := Module[
  {go = x},
  While[
   go ≠ {} && Last[go] == GoRight,
   go = Drop[go, -1]
  ];
  If[! go == {}, go = Drop[go, -1]];
  Append[go, GoRight]
 ]
```

Here is an example.

```
BacktrackSequence[{GoLeft, GoRight, GoLeft, GoRight,
  GoRight}]
```

```
{GoLeft, GoRight, GoRight}
```

In the next two functions, this kind of code tests for a list of lists of integers, a necessary condition for a Sudoku solution.

```
VectorQ[Flatten[g2, 1]]
```

```
False
```

The function `Backtrack` goes left if applying the sequence `go` (a global variable with entries `GoRight` and `GoRight`) to the matrix with `RightComposition` does not contain an empty list `{}`; otherwise it backtracks using `BacktrackSequence`. If the new sequence applied to the matrix contains only numbers, `Backtrack` throws the matrix to the nearest containing `Catch`.

```
Backtrack[X_] := Module[
  {t},
  go = If[
     ! MemberQ[Flatten[(RightComposition @@ go)[X], 1], {}],
    Append[go, GoLeft],
    BacktrackSequence[go]
   ];
  t = (RightComposition @@ go)[X];
  If[VectorQ[Flatten[t, 1]], Throw[t]];
  go
 ]
```

We now use `Backtrack` inside the function `BacktrackSolver`, which initializes the global variable `go`.

```
BacktrackSolver[X_] := Catch[
  go = {};
  While[
   ! VectorQ[Flatten[(RightComposition @@ go)[X], 1]],
   Backtrack[X]
  ]
 ]
```

Consider the following Sudoku puzzle.

```
(h = {{1, □, □, □, 9, 8, □, 4, □}, {□, 2, □, □, 5, □, □, 9, □},
     {3, □, 8, □, □, □, 6, □, □}, {□, □, □, 5, □, □, □, □, □},
     {□, □, 9, 6, □, 4, 2, □, □}, {□, □, □, □, □, 9, □, □, □},
     {□, □, 2, □, □, □, 1, □, 9}, {□, 8, □, □, □, □, □, 6, □},
     {□, 1, □, 8, 7, □, □, □, 3}}) // display
```

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

```
(h1 = MasterSolver[h]) // display
```

1	567	567	237	9	8	357	4	257
467	2	467	1347	5	1367	378	9	178
3	9	8	1247	124	127	6	1257	1257
24678	3467	13467	5	1238	1237	34789	137	14678
578	357	9	6	138	4	2	1357	1578
245678	34567	134567	1237	1238	9	34578	1357	145678
4567	34567	2	34	346	356	1	8	9
457	8	3457	12349	1234	1235	457	6	2457
9	1	456	8	7	256	45	25	3

We have failed so far to solve the puzzle using `MasterSolver`; so we try the backtracking technique.

```
BacktrackSolver[h1] // display
```

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

To see what sequence solved this puzzle, we only have to enter `go`.

```
go
```

```
{GoRight, GoRight, GoLeft, GoLeft}
```

We next try `BacktrackSolver` on `g1` defined in the previous section.

```
BacktrackSolver[g1] // display
```

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

If we now enter `go`, we can see what sequence solved this puzzle.

```
go
```

```
{GoRight}
```

Next is Evil Puzzle 8,076,199,743 from Web Sudoku [1].

```
(i = {{□, 7, □, □, □, □, □, 3, □}, {8, 1, □, 4, □, □, □, □, 6},
     {□, □, □, □, □, 5, 4, □, □}, {□, □, □, □, 2, □, □, □, 9},
     {□, □, 9, 1, 5, 4, 2, □, □}, {□, □, □, □, 8, □, □, □, □},
     {□, □, 7, 9, □, □, □, □, □}, {2, □, □, □, □, 6, □, 7, 5},
     {□, 9, □, □, □, □, □, 6, □}}) // display
```

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

```
(i1 = MasterSolver[i]) // display
```

4569	7	2456	268	169	128	158	3	128
8	1	235	4	379	237	57	259	6
369	236	236	23678	13679	5	4	129	1278
134567	34568	134568	367	2	37	13567	145	9
367	36	9	1	5	4	2	8	37
134567	23456	123456	367	8	9	13567	145	1347
13456	34568	7	9	134	1238	138	124	12348
2	348	1348	38	134	6	9	7	5
1345	9	13458	23578	1347	12378	138	6	12348

```
BacktrackSolver[i1] // display
```

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

Again, the solving sequence is given by `go`.

```
go
```

```
{GoRight, GoRight, GoLeft}
```

This final puzzle was created by Arto Inkala, a mathematician based in Finland; it is claimed to be the world's hardest Sudoku puzzle [2].

```
(j = {{8, □, □, □, □, □, □, □, □}, {□, □, 3, 6, □, □, □, □, □},
     {□, 7, □, □, 9, □, 2, □, □}, {□, 5, □, □, □, 7, □, □, □},
     {□, □, □, □, 4, 5, 7, □, □}, {□, □, □, 1, □, □, □, 3, □},
     {□, □, 1, □, □, □, □, 6, 8}, {□, □, 8, 5, □, □, □, 1, □},
     {□, 9, □, □, □, □, 4, □, □}}) // display
```

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

```
(j1 = MasterSolver[j]) // display
```

8	1246	24569	2347	12357	1234	13569	4579	1345
								679
12459	124	3	6	12578	1248	1589	45789	14579
1456	7	456	348	9	1348	2	458	13456
123469	5	2469	2389	2368	7	1689	2489	12469
12369	12368	269	2389	4	5	7	289	1269
24679	2468	24679	1	268	2689	5689	3	24569
23457	234	1	23479	237	2349	359	6	8
23467	2346	8	5	2367	23469	39	1	2379
23567	9	2567	2378	123678	12368	4	257	2357

```
BacktrackSolver[j1] // display
```

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

Here is how this puzzle was solved.

```
go
```

```
{GoRight, GoLeft, GoLeft, GoLeft, GoRight,
 GoLeft, GoRight, GoLeft, GoLeft, GoRight}
```

There are many other techniques known to experienced Sudoku solvers that could be added to our programs; also backtracking could obviously be extended to triples, and so on.

Conclusion

Sudoku provides a superb opportunity to introduce useful programming techniques to students of Mathematica. Backtracking is one such technique that is largely absent from standard discussions of Mathematica programming but, as we have shown, is easily implemented in Mathematica when needed.

References

[1] Web Sudoku. (Jan 18, 2018) www.websudoku.com.

[2] Efamol. "Introducing the World's Hardest Sudoku." (Jan 18, 2018) www.efamol.com/efamol-news/news-item.php?id=43.

R. Cowen, "A Beginner's Guide to Solving Sudoku Puzzles by Computer," *The Mathematica Journal*, 2018. dx.doi.org/doi:10.3888/tmj.20-1.

About the Author

Robert Cowen is Professor Emeritus in Mathematics, Queens College, CUNY. He does research in logic, combinatorics and set theory. He taught a course in Mathematica programming for many years, emphasizing discovery of mathematics, and is currently working on a text on learning Mathematica through discovery with John Kennedy. His website is sites.google.com/site/robertcowen.

Robert Cowen
16422 75th Avenue
Fresh Meadows, NY 11366
robert.cowen@gmail.com