Solving Sudokus in APL

Let's try to solve this sudoku in APL:

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

Note: You can read more about sudokus elsewhere on this homepage.

We represent the problem as a 9x9 matrix. In order to make it easier to get problems in and out of the APL system we start by defining two help functions:

```
    ∇ Z←A2N A
[1]   Z← 9 9 ⍴¯1+'0123456789'⍳A
    ∇

    ∇ Z←N2A N
[1]   Z←'0123456789'[1+,N]
    ∇
```

We can create the matrix O1 like this:

```
      O1←A2N '000060000400807005009103800080000090700658004020000080002301
      O1
  0  0  0  0  6  0  0  0  0
  4  0  0  8  0  7  0  0  5
  0  0  9  1  0  3  8  0  0
  0  8  0  0  0  0  0  9  0
  7  0  0  6  5  8  0  0  4
  0  2  0  0  0  0  0  8  0
  0  0  2  3  0  1  4  0  0
  1  0  0  5  0  9  0  0  8
  0  0  0  0  7  0  0  0  0
```

The unknown positions are represented by zeros.

We will now try to find the digits 1-9 that must replace the zeros so that the digits 1-9 occur once in each row, column, and 3x3 submatrix.

The brute force algorithm.

We use this simple algorithm:

1. Find the first zero.
2. If no zeros are found, we're done: Print the solution and exit.
3. Find the list of possible values for this cell, i.e. the list of numbers not occurring in this row, column, or submatrix.
4. If this list is empty, we have come to a dead end and we just exit.
5. For each number in this list, generate a copy of the original matrix, insert the number, and call the program recursively.

Let's see how this can be done:

We call the matrix with the puzzle *M*, and we find the row number *R* of the first zero:

```
      R←(⌊/M)⍳0
```

The expression ⌊/*M* gives us the minimum value in each row, and the following ⍳0 looks for a zero in this list. It will return 10 if no zero is found. This gives us the first line in the program:

```
[1]   →((R←(⌊/M)⍳0)≤9)/L1
```

which means: Assign the row number of the first occuring zero to *R* and jump to the label *L*1 if *R* is less than or equal to 9.

The compression operator / is often used together with conditional branches:

```
      0/17

      1/17
17
```

So the statement →(*R*≤9)/17 means go to line 17 if *R*≤9 else continue with the next line in the program. It is convenient to use symbolic labels for branches in APL as line numbers are renumbered automatically when lines are removed or inserted. Labels are local variables that automatically are given the value of the line number in which they occur when the program starts.

The next lines print *M* as the solution, a blank line, and exits the program:

```
[2]   M
[3]   ⍳0
[4]   →0
```

We now must find the column number, *C*, of the first zero occurring in row *R*:

```
[5]  L1:C←M[R;]⍳0
```

This line contains the label *L*1 we referred to in line no. 1.

Now the fun part begins: We join all the numbers from row *R* with column *C* together with the numbers from the 3x3 submatrix that contains the element *M*[*R*;*C*]:

```
      M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3]
```

Let's try this for row 1, column 1 (the first zero in the example):

```
      M←O1
      R←1
      C←1
      M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3]
0 0 0 0 6 0 0 0 0 0 4 0 0 7 0 0 1 0 0 0 0 4 0 0 0
```

We use the membership operator ∈ to see which of the numbers 1-9 that exist in this list:

```
      (⍳9)∈(M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3])
1 0 0 1 0 1 1 0 1
```

We can see that the list contains the numbers 1, 4, 6, 7, and 9. But we are interested in the numbers that does not exist, so we negate the answer:

```
      ~(⍳9)∈(M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3])
0 1 1 0 1 0 0 1 0
```

We are now ready to use the compress operator to get the list of possible candidates:

```
      (~(⍳9)∈(M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3]))/⍳9
2 3 5 8
```

So we have to try the numbers 2, 3, 5, and 8 as candidates in the first cell.

So we have the next line ready for the program:

```
[6]   I←⍴T1←(~(⍳9)∈(M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3])
```

We assign the result list to the variable *T*1 and the length of *T*1 to *I*.

If *I* is zero we have come to a dead end and we exit the program. This gives us the next line of the program:

```
[7]  L2:→(I=0)/0
```

This line has the label *L2*, we'll need this in a little while.

We must now prepare a copy of the matrix *M* and insert the numbers from the list *T*1 into row *R*, column *C*. We do it like this:

```
[8]   M1←M
[9]   M1[R;C]←T1[I]
[10]  SUDOKU M1
[11]  I←I−1
[12]  →L2
```

We start from the end of the list *T*1 by inserting *T*1[*I*] into *M*1[*R*;*C*], call the program again, and count down *I*.

That's it! The whole program looks like this:

```
     ∇ SUDOKU M;R;C;M1;T1;I
[1]    →((R←(⌊/M)⍳0)≤9)/L1
[2]    M
[3]    ⍳0
[4]    →0
[5]   L1:C←M[R;]⍳0
[6]    I←⍴T1←(~(⍳9)∈(M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3])
[7]   L2:→(I=0)/0
[8]    M1←M
[9]    M1[R;C]←T1[I]
[10]   SUDOKU M1
[11]   I←I-1
[12]   →L2
     ∇
```

Note that the declaration line contains `;R;C;M1;T1;I` after the parameter *M,* this is a list of the variables that are local to this program.

Does it work?

```
       SUDOKU O1
  8  1  7  2  6  5  9  4  3
  4  6  3  8  9  7  2  1  5
  2  5  9  1  4  3  8  7  6
  6  8  4  7  3  2  5  9  1
  7  9  1  6  5  8  3  2  4
  3  2  5  9  1  4  6  8  7
  5  7  2  3  8  1  4  6  9
  1  4  6  5  2  9  7  3  8
  9  3  8  4  7  6  1  5  2
```

Yes, it does! We can copy the function *TIME* from the workspace *ADVANCEDEX* in public library 1 to measure the CPU time it takes to run:

```
      )COPY 1 ADVANCEDEX TIME
SAVED  16.41.30 09/06/72
```

and reset the timing variable *TIMER*. Calling *TIME* will display the number of minutes, seconds, and 1/60's seconds since last call:

```
       TIMER←0
       TIME
0  38  53
       TIME
0  0  0
```

Lets time the sudoku program:

```
       TIME
0  0  0
       SUDOKU O1
  8  1  7  2  6  5  9  4  3
  4  6  3  8  9  7  2  1  5
  2  5  9  1  4  3  8  7  6
  6  8  4  7  3  2  5  9  1
```

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

       TIME
0   39   25
```

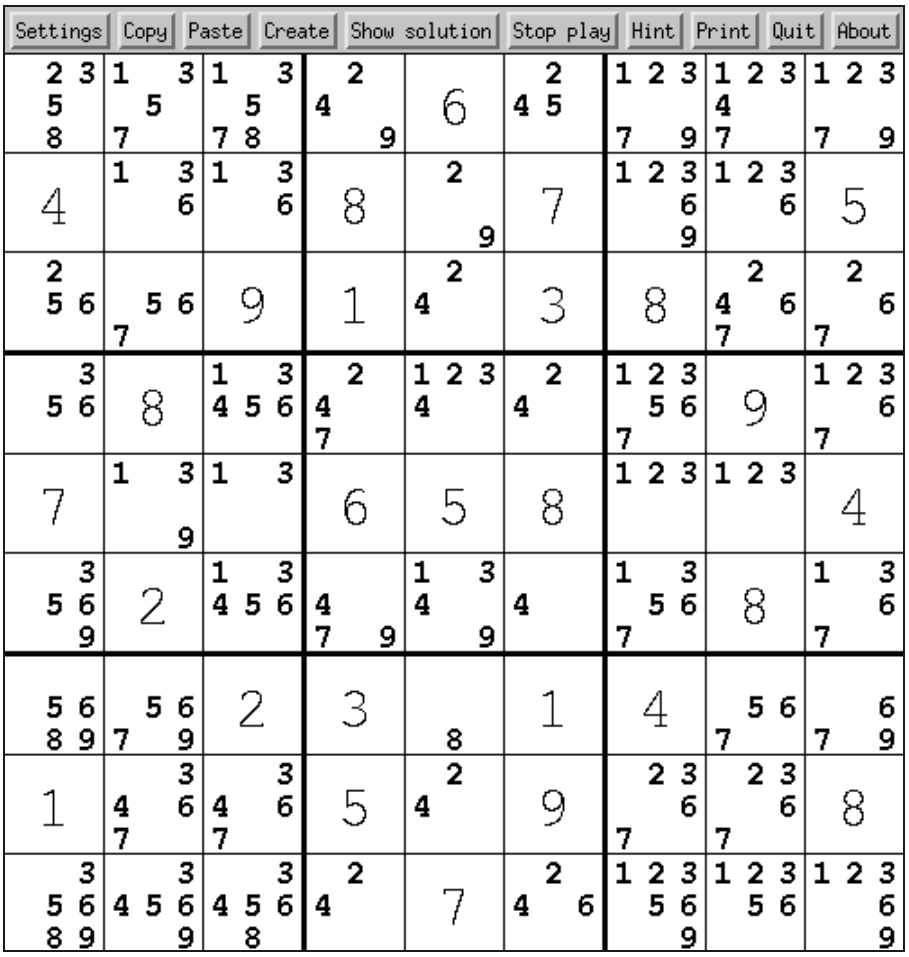
It takes nearly 39.5 seconds on my laptop.

This algorithm does not work very well if there are many zeros in the puzzle. It can take hours to finish.

We shall make improvements to this in the next chapter.

Improved algorithm

When I solve sudoku's with my own program it looks like this:



The small numbers in the empty cells are the possible candidates for this cell. You can recognize the numbers 2, 3, 5, and 8 we found for the first cell.

But you see also something else: Two of the cells have only one candidate, cells `O1[6;6]` and `O1[7;5]`. It would be a good idea to start with these cells instead of just taking the first zero in the puzzle.

Also note that in row 5, number 9 occurs only once.

In column 5, number 8 occur once. In column 6, the numbers 5 and 6 occur only once.

In the second submatrix, `O1[⍳3;3+⍳3]`, the number 5 occurs only once. And in the eights submatrix, `O1[6+⍳3;3+⍳3]`, the numbers 6 and 8 occur only once.

How can we get this information in APL?

We now call the matrix to be investigated *M*.

We start by making a help table with 81 rows and 27 columns.

One row for each cell in the puzzle.

The first 9 columns in the table row `((R-1)×9)+C` contains the 9 numbers from row *R*.

The next 9 numbers in this row contains the 9 numbers from column *C*.

The last 9 numbers contains the numbers from the submatrix `M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3]`.

```
      T1←M[,⍉ 9 9 ⍴⍳9;]
      T1←T1,⍉M[;81⍴⍳9]
      T1←T1,(9 9 ⍴ 1 3 2 4 ⍉ 3 3 3 3 ⍴M)[, 2 4 1 3 ⍉ 3 3 3 3 ⍴⍳9;]
```

Let's see how the indices are calculated:

```
      ,⍉ 9 9 ⍴⍳9
 1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3  3
       5  5  5  5  5  5  5  5  5  6  6  6  6  6  6  6  6  6  7  7  7  7  7
       8  8  9  9  9  9  9  9  9  9  9
      81⍴⍳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
       1  2  3  4  5  6  7  8  9  1  2  3  4  5  6  7  8  9  1  2  3  4  5
       8  9  1  2  3  4  5  6  7  8  9
      , 2 4 1 3 ⍉ 3 3 3 3 ⍴⍳9
 1  1  1  2  2  2  3  3  3  1  1  1  2  2  2  3  3  3  1  1  1  2  2  2  3
       4  4  4  5  5  5  6  6  6  4  4  4  5  5  5  6  6  6  7  7  7  8  8
       9  9  7  7  7  8  8  8  9  9  9
```

The first list is the row numbers for the rows to be inserted, the second list is the column numbers, and the last list is the submatrix numbers.

We use this smart transformation to convert from submatrix to row format:

```
      M
 0  0  0  0  6  0  0  0  0
 4  0  0  8  0  7  0  0  5
 0  0  9  1  0  3  8  0  0
 0  8  0  0  0  0  0  9  0
 7  0  0  6  5  8  0  0  4
 0  2  0  0  0  0  0  8  0
 0  0  2  3  0  1  4  0  0
 1  0  0  5  0  9  0  0  8
 0  0  0  0  7  0  0  0  0
      9 9 ⍴ 1 3 2 4 ⍉ 3 3 3 3 ⍴M
 0  0  0  4  0  0  0  0  9
 0  6  0  8  0  7  1  0  3
 0  0  0  0  0  5  8  0  0
 0  8  0  7  0  0  0  2  0
 0  0  0  6  5  8  0  0  0
 0  9  0  0  0  4  0  8  0
 0  0  2  1  0  0  0  0  0
```

```
    3   0   1   5   0   9   0   7   0
    4   0   0   0   0   8   0   0   0
```

The first row in the matrix `9 9 ⍴ 1 3 2 4 ⍉ 3 3 3 3 ⍴M` contains submatrix no. 1 from *M*, the second row contains submatrix no. 2, etc.

Let's show this in colors:

M:

0	0	0	0	6	0	0	0	0
4	0	0	8	0	7	0	0	5
0	0	9	1	0	3	8	0	0
0	8	0	0	0	0	0	9	0
7	0	0	6	5	8	0	0	4
0	2	0	0	0	0	0	8	0
0	0	2	3	0	1	4	0	0
1	0	0	5	0	9	0	0	8
0	0	0	0	7	0	0	0	0

`9 9 ⍴ 1 3 2 4 ⍉ 3 3 3 3 ⍴M`:

0	0	0	4	0	0	0	0	9
0	6	0	8	0	7	1	0	3
0	0	0	0	0	5	8	0	0
0	8	0	7	0	0	0	2	0
0	0	0	6	5	8	0	0	0
0	9	0	0	0	4	0	8	0
0	0	2	1	0	0	0	0	0
3	0	1	5	0	9	0	7	0
4	0	0	0	0	8	0	0	0

The matrix `T1` contains 81 rows and 27 columns.

We must now see which numbers that occur in each row.

We can't use the membership operator this time, as it operates on all rows of `T1`:

```
      ⍴T1
81  27
      (⍳9)∈T1
1  1  1  1  1  1  1  1  1
```

Instead we create an outer product using the = operator: `(⍳9)∘.=T1`:

```
      ⍴(⍳9)∘.=T1
9  81  27
```

This is a big table, but APL stores a table of logical values using one bit per element.

We reduce it along the rows and negates:

```
      ~∨/(⍳9)∘.=T1
0 1 1 0 0 0 1 1 1 0 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 0
      0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 1 1 1 1
      0 0 0 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 1
0 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1 1 1 1 0 0 0 0
      0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0
1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0 0 0 1 1 1 0 0 0 1 1 0 0
      1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 0
```

```
  0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 0
        1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1
  0 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 0 0 1 0 0 1 1 1 0
        1 1 1 0 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0
  1 0 1 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 1 0 1 0 1 0 0 0 0
  0 0 0 1 1 0 1 0 1 0 0 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
        1 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1
```

This is a matrix with dimensions 9 81. If element `[I;J]` is true it means that the number *I* is a possible candidate for cell no. *J*. You can see from the first number in each row that the candidates for the first cell is: 2, 3, 5, and 8.

We call this matrix *T2*:

```
      T2←~∨/(⍳9)∘.=T1
```

If we sum the rows and reshape to 9x9, and add 10 if the correponding cell in the original matrix is non-zero, we get this:

```
      (9 9 ⍴+/[1] T2)+10×M≠0
 4   4   5   3  13   3   5   5   5
13   3   3  12   2  11   5   4  15
 3   3  13  12   2  13  13   4   3
 3  15   5   3   4   2   6  16   5
12   3   2  12  14  11   3   3  13
 4  16   5   3   4   1   5  15   4
 4   4  14  10   1  11  14   3   3
12   4   4  12   2  13   4   4  14
 5   5   5   2  13   3   6   5   5
```

You can see that this table contains two elements that are 1: Row 6, column 6, and row 7 column 5. It means that these two cells contain only a single candidate.

We can do other operations on the *T2* matrix, but first we get rid of the cells that are already occupied:

```
      T2← 9 9 9 ⍴⍉T2∧ 9 81 ⍴M=0
```

Now `T2[R;C;I]` is true if the number *I* is a candidate in row *R*, column *C*.

If we reduce along dimension 2 we get:

```
      +/[2] T2
5  6  6  3  4  0  5  2  3
4  3  4  0  0  4  0  0  2
0  4  0  2  2  4  3  0  0
4  5  5  4  3  4  3  0  0
4  2  4  0  0  0  0  0  1
4  0  5  4  3  4  3  0  3
0  0  0  0  3  4  3  2  3
0  3  4  3  0  4  4  0  0
3  5  6  4  5  7  0  2  4
```

This table contains a number 1 in row 5, column 9. It means that the number 9 is a single candidate in row 5.

We add along dimension 1:

```
      +/[1] T2
  0  2  4  0  6  5  0  3  3
  3  0  5  2  4  5  4  0  3
  5  0  7  4  4  5  2  2  0
  0  3  0  4  0  0  2  0  2
  2  4  2  4  0  0  0  1  2
  0  3  0  4  1  1  0  0  0
  6  6  7  0  3  5  4  0  3
  4  6  5  2  2  5  4  0  0
  4  4  4  0  0  5  5  0  3
```

This table contains a number 1 in row 5, column 8. It means that the number 8 is a single candidate in column 5. And we have two 1's in row 6: The numbers 5 and 6 occur only once in column 6.

In order to find the single occurances in the submatrices, we rearrange the matrix before adding:

```
      +/[2] 9 9 9 ⍴ 1 3 2 4 5 ⍉ 3 3 3 3 9 ⍴T2
  4  2  5  0  5  4  3  2  0
  0  4  0  3  1  0  0  0  2
  5  7  5  2  0  4  5  0  3
  4  0  6  2  4  4  0  0  2
  2  3  2  6  0  0  2  0  2
  6  4  6  0  2  4  4  0  0
  0  0  5  4  5  7  3  3  4
  0  3  0  3  0  1  0  1  0
  3  5  5  0  3  7  4  0  3
```

Now we have a number 1 in row 2, column 5. It means that the number 5 occurs once in submatrix 2, and we have two 1's in row 8, the numbers 6 and 8 occur once in submatrix 8.

We now have all the tools ready to assemble the program that uses the new algorithms. We still keep the brute force algorithm as a fallback, but with one improvement: Instead of just take the first zero in the matrix and try all possible candidates, we select the unoccupied cell that has fewest possible candidates.

Let's take a look at the final program:

```
     ∇ SUDOKU M;R;C;M1;T1;T2;T3;I
[1]    T1←M[,⍉ 9 9 ⍴⍳9;]
[2]    T1←T1,⍉M[;81⍴⍳9]
[3]    T1←T1,(9 9 ⍴ 1 3 2 4 ⍉ 3 3 3 3 ⍴M)[, 2 4 1 3 ⍉ 3 3 3 3 ⍴⍳9;]
[4]    T2←~∨/(⍳9)∘.=T1
[5]    T1←,(9 9 ⍴+/[1] T2)+10×M≠0
[6]    T2← 9 9 9 ⍴⍉T2∧ 9 81 ⍴M=0
[7]    →L1×SUDOKU1
[8]    →L2×SUDOKU2
[9]    →L2×SUDOKU3
[10]   →L2×SUDOKU4
[11]   →L2×SUDOKU5
[12]   →L2×SUDOKU6
[13] L1:M
[14]   ⍳0
[15]   →0
[16] L2:I←⍴T1
[17] L3:→(I=0)/0
[18]   M1←M
[19]   M1[R;C]←T1[I]
[20]   SUDOKU M1
```

```
[21]   I←I-1
[22]   →L3
     ∇
```

It has been split up in several programs to make it easier to read. The programs *SUDOKU*1 to *SUDOKU*6 each test for a specific condition and returns either ⍳0 or 1. So if *SUDOKU*1 returns 1, we jump to the label *L*1.

The program *SUDOKU*1 looks like this:

```
     ∇ Z←SUDOKU1
[1]    R←(⍋T1)[1]
[2]    Z←⍳0
[3]    →(T1[R]≤9)/0
[4]    Z←1
     ∇
```

We sort the number of candidates and assign the number of the first to *R*. If this number is larger than 9 it means that there are no zeros left in the puzzle. In the main program we jump to label *L*1 and print the solution.

In *SUDOKU*2 we look for cells with a single candidate:

```
     ∇ Z←SUDOKU2
[1]    Z←⍳0
[2]    →(T1[R]>1)/0
[3]    Z←1
[4]    C←T1[R]
[5]    T1←⍳0
[6]    →(C=0)/0
[7]    C←1+9|(R-1)
[8]    R←1+⌊(R-1)÷9
[9]    T1←,+/T2[R;C;]×⍳9
     ∇
```

In line two we check that *T*1[*R*] is one, else we quit and go on to the next algorithm.

In line six we check if cell with fewest candidates has zero candidates. If this is the case we return an empty vector in *T*1 and the main program exits: We have reached a dead end.

Else we calculate the row and column number and assign the candidate number to *T*1. The comma in front makes sure that *T*1 is a vector of lenght one, and not a scalar.

*SUDOKU*3 looks for candidates that occur once in a row. It looks like this:

```
     ∇ Z←SUDOKU3
[1]    Z←⍳0
[2]    T3←+/[2] T2
[3]    I←(,T3)⍳1
[4]    →(I=82)/0
[5]    Z←1
[6]    T1←,1+9|(I-1)
[7]    R←1+⌊(I-1)÷9
[8]    C←+/(,T2[R;;T1])×⍳9
     ∇
```

In line 3 we look for the first occurrence of the number 1. If none exist, we exit in line 4. Else we assign the candidate number to *T*1 and calculates the row and column numbers.

*SUDOKU*4

```
     ∇ Z←SUDOKU4
[1]    Z←⍳0
[2]    T3←+/[1] T2
[3]    I←(,T3)⍳1
[4]    →(I=82)/0
[5]    Z←1
[6]    T1←,1+9|(I-1)
[7]    C←1+⌊(I-1)÷9
[8]    R←+/(,T2[;C;T1])×⍳9
     ∇
```

And *SUDOKU5* checks the submatrices:

```
     ∇ Z←SUDOKU5;R1;C1
[1]    Z←⍳0
[2]    T3←+/[2] 9 9 9 ⍴ 1 3 2 4 5 ⌽ 3 3 3 3 9 ⍴T2
[3]    I←(,T3)⍳1
[4]    →(I=82)/0
[5]    Z←1
[6]    T1←,1+9|(I-1)
[7]    R1← 3 3 ⊤⌊(I-1)÷9
[8]    C1← 3 3 ⊤¯1++/(,(9 9 9 ⍴ 1 3 2 4 5 ⌽ 3 3 3 3 9 ⍴T2)[1+ 3 3 ⊥R1;;T1])>
[9]    R←1+C1[1]+3×R1[1]
[10]   C←1+C1[2]+3×R1[2]
     ∇
```

We use the ⊤ operator to convert the index into base 3 numbers, and the ⊥ operator to convert back again:

```
       3 3 ⊤0
0  0
       3 3 ⊤1
0  1
       3 3 ⊤2
0  2
       3 3 ⊤3
1  0
       3 3 ⊤4
1  1
       3 3 ⊤5
1  2
       3 3⊥1 2
5
```

The last program *SUDOKU6* contains the brute force algorithm:

```
     ∇ Z←SUDOKU6
[1]    Z←1
[2]    C←1+9|(R-1)
[3]    R←1+⌊(R-1)÷9
[4]    T1←(~(⍳9)∈(M[R;],M[;C],,M[(3×⌊((¯1+R)÷3))+⍳3;(3×⌊((¯1+C)÷3))+⍳3]))/⍳9
     ∇
```

When *SUDOKU6* is called, *R* points to the cell with the fewest number of candidates. We return the list in the vector *T1*.

Now the program is finished - is it faster?

```
       TIME
```

```
0  0  1
      SUDOKU O1
  8  1  7  2  6  5  9  4  3
  4  6  3  8  9  7  2  1  5
  2  5  9  1  4  3  8  7  6
  6  8  4  7  3  2  5  9  1
  7  9  1  6  5  8  3  2  4
  3  2  5  9  1  4  6  8  7
  5  7  2  3  8  1  4  6  9
  1  4  6  5  2  9  7  3  8
  9  3  8  4  7  6  1  5  2

      TIME
0  3  54
```

Yes, ten times faster than before.

We can try a more difficult problem:

```
      O2←A2N '8000000000036000000700902000500070000000457000001000300010000
      TIME
0  0  0
      SUDOKU O2
  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

      TIME
1  49  4
```

It took 109 seconds. It would take hours using only the brute force algorithm.

I'll close my little APL corner here. You can continue and try to implement the other sudoku algorithms in APL yourself!

```
      )OFF
010  14.12.07 03/10/14 MK
CONNECTED     6.25.35  TO DATE   52.03.29
CPU TIME      0.01.57  TO DATE    1.16.35
```