Sudoku Puzzle

Concepts used in this program:

- Algorithms
- Arithmetic operators (+, -, *, /)
- Arithmetic operators – advanced (mod, floor, ceiling, %, //, etc.)
- Arithmetic expressions
- Boolean operators (and, or, not)
- Conditions: YES/NO questions
- Conditionals (If-Else)
- Conditionals (IF)
- Conditionals (nested IF)
- Data structures – list
- Data structures – using list as 2-D array
- Looping - simple (for)
- Looping - nested
- Looping - conditional (while)
- Procedures with inputs
- Procedures with return value
- Recursion
- Relational operators (==, <, >, !=, <=, >=)
- Sequence
- String operations
- User text input
- Variables - numbers
- Variables - strings
- Variables - local/global scope

The Sudoku Puzzle:

Sudoku is a well-known number puzzle in which you are expected to fill a 9x9 matrix with digits 1 to 9 such that no digit is repeated in a row, column, or a 3x3 box.

Typically, the puzzle begins with a partially filled 9x9 matrix and you are expected to fill in the blanks.

Solution approach:

We will use what is called the "brute force" or "exhaustive search" approach. In this, we basically try all possible number combinations until the correct one is found. So, we start scanning the grid from top left and fill the first empty cell with a digit that fits (i.e. does not violate Sudoku rules) and then go to the next empty cell and fill it with a digit that fits, and so on. If a cell does not accept any digit, we go back to the previous cell that we filled and try a

different digit in it. This process of "trial and error" and "backtracking" can go on for a very long time if done manually, but since computers are fast this "dumb" approach turns out to be quite practical and solves most puzzles in a few seconds.

The following recursive algorithm shows how to implement this approach.

Solve-Sudoku Recursive algorithm:

```
Input: cell id (0 thru 80)

Look for the first blank cell
If no blank cell, puzzle either solved or unsolvable, stop
Repeat digit=1 to 9:
      If digit fits (not used already in cell's row, column or box)
            Place it in current cell
            Recursive call to Solve-Sudoku (input: current cell + 1)
            If Success returned:
                  Return Success
            End if
      End if
      Try the next digit in repeat loop
End repeat
Since no valid digit found, reset cell to 0 and return FAIL
```

Custom procedures with return values:

The recursive procedure above requires our custom blocks to have return values, which Scratch does not support. We will use ideas proposed in "Script variables" (see Scratch/blog/Script-variables.pdf for details) to overcome this problem.

Drawing the grid:

We will use the "Number-table" program (see Scratch/blog/number-tables.pdf for details) to draw the 9x9 grid with digits.

Utility procedures:

We need utility functions to check if a given digit already exists in a row, column, or 3x3 box, and to return sums of elements in a given row, column, or box. The following algorithms take care of these needs:

Algorithm Check Digit in Row

Check if the given digit is present in the given row. The 9 elements in a row appear one after the other.

```
Input: row, digit
Output: True if digit is present, False otherwise.

Procedure:
Count = 9*row
Repeat 9 times:
      If (List[count] == digit):
            Return True
```

```
      End if
      Count = Count + 1
End repeat
Return False
```

Algorithm Sum Elements Row

Return the sum of all elements in the given row. The 9 elements in a row appear one after the other.

```
Input: row (0 to 8)
Output: sum of all elements

Procedure:
Count = 9*row + 1 // Scratch list index begins at 1
Repeat 9 times:
      Sum += List[count]
      Count = Count + 1
End repeat
Return False
```

Algorithm Check Digit in Column

Check if the given digit is present in the given column. The 9 elements in a column appear one after the other but separated by 9 places. For example, column 0 elements would be at 0, 9, 18, and so on.

```
Input: column and digit
Output: True if digit is present, False otherwise.
Procedure:
Count = column
Repeat 9 times:
      If (List[count] == digit):
            Return True
      End if
      Count = Count + 9
End repeat
Return False
```

Algorithm Sum Elements Column

Return the sum of all elements in the given column. The 9 elements in a column appear one after the other but separated by 9 places. For example, column 0 elements would be at 0, 9, 18, and so on.

```
Input: column (0 to 8)
Output: sum of all elements
Procedure:
Sum = 0
Count = column+1   // In Scratch list index begins at 1
Repeat 9 times:
      Sum += List[count]
      Count = Count + 9
End repeat
Return False
```

Algorithm Check Digit in Box

Check if the given digit is present in the given 3x3 box. For this algorithm, we need to hard-code the starting location of each box in the 9x9 grid:
Boxes = [0, 3, 6, 27, 30, 33, 54, 57, 60]

The 9 elements in a box appear in a 3x3 grid starting at the location given in the list above. For example, box 0 elements would be at 0, 1, 2, 9, 10, 11, 18, 19, 20.

```
Input: box and digit
Output: True if digit is present, False otherwise.
Procedure:
Row = 0
Repeat 3 times:
      Start = starting location of box + Row*9
      Count = 0
      Repeat 3 times:
            If (List[Start + count] == digit):
                  Return True
            End if
            Count = Count + 1
      End repeat
      Row = Row + 1
End repeat
Return False
```

Algorithm Sum Elements Box

Return the sum of all elements in the given 3x3 box. For this algorithm, we need to hard-code the starting location of each box in the 9x9 grid:
Boxes = [0, 3, 6, 27, 30, 33, 54, 57, 60]

The 9 elements in a box appear in a 3x3 grid starting at the location given in the list above. For example, box 0 elements would be at 0, 1, 2, 9, 10, 11, 18, 19, 20.

```
Input: box (0 to 8)
Output: sum of all elements in box
Procedure:
Sum = 0
Row = 0
Repeat 3 times:
      Start = starting location of box + Row*9 + 1 // Scratch list
index begins at 1
      Column = 0
      Repeat 3 times:
            Sum += List[Start + Column]
            Column = Column + 1
      End repeat
      Row = Row + 1
End repeat
Return False
```

Algorithm Get Box

Find out the 3x3 box in which the given cell resides. Sudoku grid has 9 boxes. We do this computation by first looking at the row and then at the column of the cell.

```
Input: cell (0 to 80)
Output: box (0 to 8)
Procedure:
Column = cell % 9     // remainder
Row = int (cell / 9)  // integer division
If (row < 3)
      box = 0
Else if (row < 6)
      box = 3
Else
      box = 6
If (column > 5)
      box += 2
Else if (column > 2)
      box += 1
Return box
```

Algorithm Check if Puzzle Solved

If for every row, column, and box the sum of elements comes to 45 (sum of 1 to 9) that means the puzzle has been solved. You can simply call the Sum algorithms above for all rows, column, and boxes to verify this.

Algorithm Read Sudoku Puzzle

This routine reads the initial puzzle and creates the list S of 81 elements. The expected input format is 9 words separated by commas. For example:

```
000065130,000008900,100009002,025000709,000000000,607000240,800900004,001500000,032780000,
```

Solution:

sudoku-final.sb2

https://scratch.mit.edu/projects/323318591/

Author: Abhay B. Joshi (abjoshi@yahoo.com)

Last updated: 6 August 2019