AARP Number Cruncher Puzzle

The **March 2020 AARP** magazine featured a simple sudoku type puzzle where a 3 by 3 grid is to filled in the digits 1,2,3,…,9. Digits are to be used only once and need satisfy 3 arithmetic row and column conditions.

A Disclaimer:
Puzzles like this are often prescribed for the aging (me) to help maintain mental acuity. Personally, when I am able, I would rather code a program to solve a puzzle as opposed to actually solving it unless the puzzle is a crossword or like. The endless repetitive solving of word finds, sudoku, and similar seem a bit pointless, though it could very well be they **are more effective at preventing dementia than programming a solution.**

If the grid is represented by a 9-element array, with non-standard indexing starting at one as follows:

x[1], x[2], x[3]
x[4], x[5], x[6]
x[7], x[8], x[9]

Then the six arithmetic conditions can be expressed in the following **MiniZinc** https://www.minizinc.org/ constraint program

```
include "alldifferent.mzn";
array[1..9] of var 1..9: x;

%Row conditions
constraint (x[1]+x[2])*x[3] == 24;
constraint x[4]+x[5]+x[6]== 17;
constraint (x[7]-x[8])*x[9]== 27;

%Column conditions
constraint (x[1]+x[4])*x[7] == 44;
constraint (x[2]*x[5])-x[8]== 20;
constraint (x[3]*x[6])+x[9]== 25;
constraint alldifferent(x);
solve satisfy;

When executed, this yields the unique solution:

x = array1d(1..9 ,[5, 7, 2, 6, 3, 8, 4, 1, 9]);
found in 56msec
```

Or, in grid form

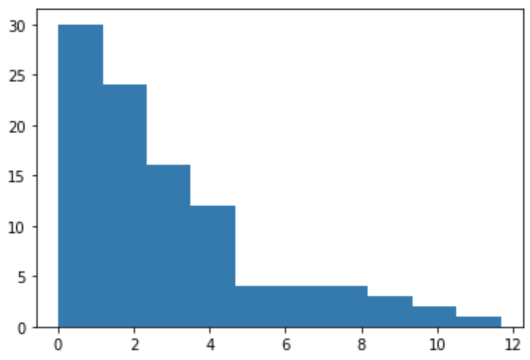
```
5, 7, 2
6, 3, 8
4, 1, 9
```

The following Python program also solves the puzzle by randomly assigning values until a solution is found. When run 100 times on my 1.8 GHz Intel Core i5 MacBook Air the program produced the expected skewed distribution of solving times shown below:

```
import random
import time
import matplotlib.pyplot as plt

def solve():
    start_time = time.clock()
    l = [1,2,3,4,5,6,7,8,9]
    while(True):
      random.shuffle(l)
      if((l[0]+l[1])*l[2] == 24 and \
         l[3]+l[4] + l[5] == 17 and \
         (l[6]-l[7])*l[8] == 27 and \
         (l[0]+l[3])*l[6] == 44 and \
          (l[1]*l[4])-l[7] == 20 and \
          (l[2]*l[5])+l[8] == 25 ):
            break
    return time.clock() - start_time

times = []
#Solve 100 times and display a histograms of times
for i in range(100):
    times.append(solve())
plt.hist(times, bins=10)
plt.show()
```

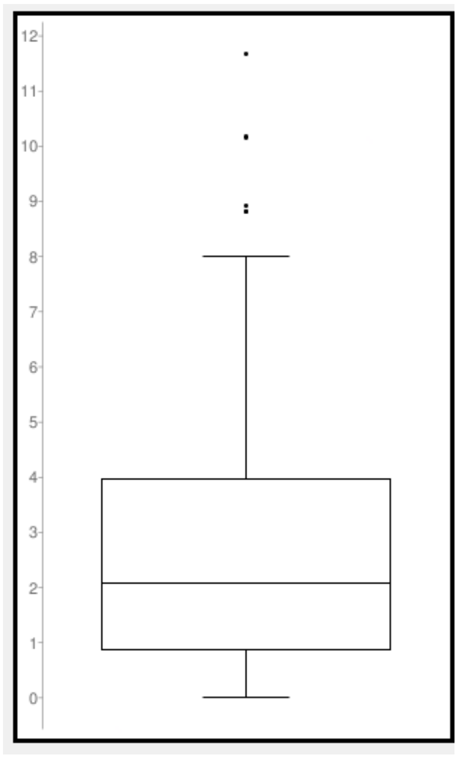


Summary statistics include (all times in seconds):

Sample size: 100
Median: 2.06889
Minimum: 0.015998000000081447
Maximum: 11.668273999999997
First quartile: 0.87124624999998
Third quartile: 3.96995525
Interquartile Range: 3.098709
Outliers: 11.668273999999997 10.183268999999996 10.148292999999967 8.928800999999964 8.82753200000002 8.815431999999987

A box-plot of the time data of 100 runs is given below:

[3.38, 4.06, 6.3, 3.98, 10.15, 7.67, 2.9, 8.01, 0.4, 1.95, 3.06, 1.16, 1.09, 1.2, 1.56, 3.65, 1.74, 4.44, 2.65, 0.66, 5.58, 0.11, 5.03, 4.46, 1.33, 2.1, 3.93, 0.86, 5.07, 1.28, 6.07, 6.08, 0.67, 0.05, 0.3, 1.18, 0.51, 0.51, 3.45, 1.24, 0.16, 0.17, 8.93, 4.48, 0.08, 0.08, 3.5, 3.34, 4.74, 1.02, 3.84, 1.43, 3.73, 4.32, 2.99, 3.1, 1.98, 3.07, 6.77, 0.1, 1.99, 1.49, 1.91, 3.33, 0.27, 0.42, 8.83, 1.82, 0.87, 2.04, 0.02, 0.41, 11.67, 0.75, 1.69, 3.63, 10.18, 0.1, 7.89, 0.38, 0.83, 3.34, 3.99, 0.57, 0.95, 0.89, 2.31, 1.52, 3.44, 2.57, 3.47, 1.68, 8.82, 2.26, 2.12, 7.41, 1.4, 2.48, 0.39, 1.27]



A faster random algorithm (.06 secs) exploring all possible permutations of the digits via Python’s itertools is given below:

```
from itertools import permutations
import time
start_time = time.clock()
p = permutations([1,2,3,4,5,6,7,8,9])
for l in p:
    if((l[0]+l[1])*l[2] == 24 and \
         l[3]+l[4] + l[5] == 17 and \
         (l[6]-l[7])*l[8] == 27 and \
         (l[0]+l[3])*l[6] == 44 and \
          (l[1]*l[4])-l[7] == 20 and \
          (l[2]*l[5])+l[8] == 25 ):
        print(l)
        print(round(time.clock() - start_time, 2))
        break

(5, 7, 2, 6, 3, 8, 4, 1, 9)
0.06 sec
```

Lastly, the seemingly fastest Back Tracking algorithm is given below:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: joscelynec

The March 2020 AARP magazine featured a simple sudoku type
puzzle where a 3 by 3 grid is to filled
in the digits 1,2,3,…,9. Digits are to be used only once and
need satisfy
3 arithmetic row conditions and 3 arithmetic column conditions.

If the grid is represented by a 9-element array as follows:

x[0][0], x[0][1], x[0][2]
x[1][0], x[1][1], x[1][2]
x[2][0], x[2][1], x[2][2]

then the 6 conditions are
```

```
Rows
(x[0][0]+x[0][1])*x[0][2] == 24
x[1][0]+ x[1][1]+x[1][2] == 17
(x[2][0]-x[2][1])*x[2][2] == 27

Columns
(x[0][0]]+x[1][1][0])*x[2][0] == 44
(x[0][1]*x[1][1])-x[2][1] == 20
(x[2]*x[5])+x[8] == 25

this puzzle has the unique solution

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

The code below adapts a Backtrack Sudoku solver to solve the
puzzle

******* A Disclaimer ********
Puzzles like this are often prescribed for the aging (me) to
help maintain mental acuity.
Personally, when I am able, I would rather code a program to
solve a puzzle as opposed
to actually solving it unless the puzzle is a crossword or like.
The endless repetitive solving
of word finds, sudoku, and similar seem a bit pointless, though
it could very well be they
are more effective at preventing dementia than programming a
solution.
"""

#Driver function to kick off the recursion
import time
start_time = time.clock()

def solveAARP(bd):
    return solveAARPCell(0, 0, bd)

"""
This function chooses a placement for the cell at (row, col)
and continues solving based on the rules we define.

Our strategy:
We will start at row 0.
We will solve every column in that row.
```

```
When we reach the last column we move to the next row.
If this is past the last row( row == bd.length) we are done.
The whole bd has been solved.
"""
def solveAARPCell(row, col, bd):


#Have we finished placements in all columns for 3the row we are
working on?
    if (col == len(bd)):
#Yes. Reset to col 0 and advance the row by 1. We will work on
the next row

        col = 0
        row += 1

#Have we completed placements in all rows? If so then we are
done.
#If not, drop through to the logic below and keep solving
things.
        if (row == len(bd)):
            return True; # Entire bd has been filled without
conflict.

#Skip non-empty entries. They already have a value in them.
    if (bd[row][col] != 0):
        return solveAARPCell(row, col + 1, bd)

#Try all values 1 through 9 in the cell at (row, col).
#Recurse on the placement if it doesn't break the constraints.
    for val in range(1, 10):

#Apply constraints. We will only add the value to the cell if
#adding it won't cause us to break sudoku rules.

        if (canPlaceValue(bd, row, col, val)):
            bd[row][col] = val
            if (solveAARPCell(row, col + 1, bd)):#recurse with
our VALID placement
                return True;

#Undo assignment to this cell. No values worked in it meaning
that
#previous states put us in a position we cannot solve from.
Hence,
#we backtrack by returning "false" to our caller.
```

```
    bd[row][col] = 0
    return False #No valid placement was found, this path is 
faulty, return false


#Will the placement at (row, col) break the puzzle

def canPlaceValue(bd, row, col, valToPlace):
    #Check column constraint. For each row, we do a check on 
column "col"
    for element in bd:
        if (valToPlace == element[col]):
            return False;

#Check row constraint. For each column in row "row", we do a 
check.
    for i in range(len(bd)):
        if (valToPlace == bd[row][i]):
            return False;

#Check 3 row and 3 col constraints
    if(row == 0 and col == 2 and (bd[0][0]+bd[0]
[1])*valToPlace != 24):
        return False
    if(row == 1 and col == 2 and bd[1][0] + bd[1][1] + 
valToPlace != 17):
        return False
    if(row == 2 and col == 0 and (bd[0][0] + bd[1]
[0])*valToPlace != 44):
        return False
    if(row == 2 and col == 1 and (bd[0][1]*bd[1][1]) - 
valToPlace != 20):
        return False
    if(row == 2 and col == 2 and  (bd[2][0]-bd[2][1])*valToPlace 
!= 27):
        return False
    if(row == 2 and col == 2 and  (bd[0][2]*bd[1][2])+valToPlace 
!= 25):
        return False
    return True

#Initialize board
bd = [[0,0,0],
      [0,0,0],
      [0,0,0]]
```

```
solveAARP(bd)
print(bd)
print(round(time.clock() - start_time,3),'secs')
'''
[[5, 7, 2], [6, 3, 8], [4, 1, 9]]
0.012 secs
'''
```