

## PO-04

### Solving Kenken Puzzles – By Not Playing

John R. Gerlach, SAS® / CDISC Analyst, Hamilton, NJ

#### ABSTRACT

Solving Kenken puzzles requires more than making sure that numbers are used only once in a row and column of a matrix. Unlike Sudoku puzzles that can use any symbol and have sub-matrices, Kenken puzzles require actual integers and have contiguous cells, called cages. And, unlike a sub-matrix that contains a unique collection of numbers or symbols, Kenken puzzles have cages that must contain natural numbers representing a total as a function of its assigned arithmetic operation. For example, consider a 4x4 Kenken puzzle having a cage containing 3 cells whose total is 11 as a function of simple addition. One possible set of 3 numbers would be:  $4+3+4=11$ . The objective is to complete the grid using numbers ranging from 1 to  $N$  that satisfies both *cage arithmetic* and *row / column uniqueness*.

Depending on the size of the  $N \times N$  grid, the number (and size) of the cages, as well as the arithmetic operations used, a Kenken puzzle offers a formidable challenge for logic puzzle fans. However, rather than play the game of considering numerous possible sets ranging from two integers, for subtraction and division, to  $N$ -digits, for addition and multiplication, this paper proposes a SAS solution that obtains the viable sets for each cage straight-away and solves the puzzle by identifying the only appropriate collection of cage-specific sets.

#### INTRODUCTION

Ken-Ken puzzles are the latest craze of logic puzzles that might even exceed Sudoku puzzles. Both puzzles are similar, that is, each symbol occurs exactly once in each row and exactly once in each column. However, Kenken puzzles require the use of actual numbers in order to complete a collection of adjacent cells, called Cages. In fact, a cage has an assigned total and associated arithmetic operation (the clues to the puzzle), which indicate the possible values allowed. Thus, Kenken puzzles pose an added challenge of populating a grid that has been partitioned by cages.

2÷		3×	
8+			4
3−		4	6×
	2÷		

2÷ 2	4	3× 3	1
8+ 3	2	1	4 4
3− 1	3	4 4	6× 2
4	2÷ 1	2	3

The arithmetic operations include addition, subtraction, multiplication, and division. Addition and multiplication use all sets. For example, the highlighted cage whose total is three when the numbers 1,3,1 are multiplied. However, subtraction and division are restricted to sets having two numbers; moreover, regardless of the order of these sets, the result is **always** a Natural number. For example, the sets {6,2} and {2,6} compute to the value 3 when divided and 4 when subtracted. In fact, their order determines their placement in the grid as defined by the cage. Finally, there is the singular set as illustrated by the highlighted cell (4,4) containing the number 3 and having no arithmetic operator.

## DEFINING A PUZZLE

We proceed to propose a data file that intuitively represents several Kenken puzzles. Certainly, there must be a field representing the  $i$ th puzzle. The remaining fields denote the Cage, the total, the arithmetic operation, and the cell (x,y) coordinates that define the cage. Notice that the operation for a singular cage is denoted by a dot, similar to the missing value in SAS, since there is no arithmetic operation. Finally, the order of the coordinates must not conflict with the natural order of valid sets, which is left as an exercise for the reader to ponder.

---

```
1 C1 2 / 1,1 1,2
1 C2 3 * 1,3 1,4 2,3
1 C3 8 + 2,1 2,2 3,2
1 C4 4 . 2,4
1 C5 3 - 3,1 4,1
1 C6 4 . 3,3
1 C7 6 * 3,4 4,4
1 C8 2 / 4,2 4,3
2 C1 3 . 1,1
2 C2 8 + 1,2 2,2 2,3
2 C3 3 - 1,3 1,4
2 C4 8 + 2,4 3,4 3,3
2 C5 1 . 4,4
2 C6 8 + 3,2 4,2 4,3
2 C7 7 + 2,1 3,1 4,1
```

---

Consider the following Data step that reads the two 4x4 Kenken puzzles, selecting the first puzzle. Notice that the variable NCELLS is computed, which is needed later when obtaining possible sets that satisfy a given cage.

```
%let puzzle = 1;

data puzzle;
  array cells{*}$3 c1-c4;
  infile p4x4 missover;
  input puzzle $ cage $ total oper $ (c1-c4)($&);
  if puzzle ne "&puzzle."
    then delete;
  do i = 1 to dim(cells);
    if cells{i} eq ''
      then leave;
    end;
    ncells = i - 1;
    drop puzzle i;
  run;
```

Next we need a mechanism to render the puzzle in a meaningful way so that you can discern one cage from another in the context of the grid. The following Data step processes a data set that emulates the raw data file representing a puzzle by performing two tasks: first it populates a 2-dimensional matrix that contains the Cage Number, Total, and Arithmetic Operation; and, it uses that matrix to generate the reporting data set such that it writes four observations containing the variables ROW, COL1-COL4.

```
data rep;
  array puzzle{4,4}$10 r1c1-r1c4 r2c1-r2c4 r3c1-r3c4 r4c1-r4c4;
  retain r1c1-r1c4 r2c1-r2c4 r3c1-r3c4 r4c1-r4c4;
  array cols{4} $10 col1-col4;
  array cells{4} $5 c1-c4;
  set puzzle end=eof;
  do i = 1 to 4;
    if cells{i} ne ''
      then do;
        row = input(scan(cells{i},1),best.);
        col = input(scan(cells{i},2),best.);
        puzzle{row,col} = trim(left(cage)) || '[';
```

```

        || put(total,3.) || ' ' || trim(left(oper)) || '>';
    end;
    else leave;
end;
if eof
then do;
    do row = 1 to 4;
        do col = 1 to 4;
            cols{col} = puzzle{row,col};
        end;
        output;
    end;
end;
keep row coll-col4;
run;

```

Then, the following Report procedure renders the puzzle such that the cages and their attributes are easily understood, as shown below.

```

proc report data=rep nowindows headline headskip split='!';
columns row ('- Column -' coll-col4);
define row / order 'Row';
define coll / display width=10 center '1';
define col2 / display width=10 center '2';
define col3 / display width=10 center '3';
define col4 / display width=10 center '4';
break after row / skip;
title1 "KenKen 4x4 Puzzle";
run;

```

---

	----- Column -----			
Row	1	2	3	4
1	C1[ 2 /]	C1[ 2 /]	C2[ 3 *]	C2[ 3 *]
2	C3[ 8 +]	C3[ 8 +]	C2[ 3 *]	C4[ 4 ]
3	C5[ 3 -]	C3[ 8 +]	C6[ 4 ]	C7[ 6 *]
4	C5[ 3 -]	C8[ 2 /]	C8[ 2 /]	C7[ 6 *]

---

## ASSUMPTIONS

Before proceeding to solve Kenken puzzles, it is assumed that a Kenken puzzle is well-constructed having only one unique solution. In order to emphasize the importance of having a well-constructed puzzle, imagine a simple 4x4 Kenken puzzle whose grid is partitioned into 8 cages such that each row contains two cages, as follows.

---

	----- Column -----			
Row	1	2	3	4
1	C1[ 3 +]	C1[ 3 +]	C2[ 7 +]	C2[ 7 +]
2	C3[ 5 +]	C3[ 5 +]	C4[ 5 +]	C4[ 5 +]
3	C5[ 7 +]	C5[ 7 +]	C6[ 3 +]	C6[ 3 +]
4	C7[ 5 +]	C7[ 5 +]	C8[ 5 +]	C8[ 5 +]

---

This simple puzzle has more than one solution, as shown below. In fact, the two solutions shown differ merely by swapping the second and fourth rows. Notice that the rows 1 and 3 could not be swapped in order to formulate another solution since the totals are not the same. However, if the arithmetic operation were subtraction and the total were 1 for cages: 1, 2, 5, and 6, rows 1 and 3 respectively; then these rows could have been swapped. More importantly, such puzzles are neither challenging nor interesting, hence not considered.

---

1	2	3	4		1	2	3	4
2	3	4	1		4	1	2	3
3	4	1	2	And	3	4	1	2
4	1	2	3		2	3	4	1

---

## VIABLE SETS

In the context of the grid, a cage is merely a collection of contiguous cells. However, in the context of its respective arithmetic operation, a cage represents a collection of natural numbers. Thus, depending on the number of cells and the arithmetic operation, there is a *pre-ordained* collection of numbers called Viable Sets. A *Viable Set* is an *ordered* collection of Natural numbers ranging from 1 to  $N$  whose values may be used more than once, however, cannot be consecutive. Consider the following valid and invalid sets

**Valid Sets**                      {1,2}, {1,2,3}, {1,2,3,4}, {1,2,3,2}, and {1,2,1,2}

**Invalid Sets**                    {1,1,2}, {1,2,3,3}, and {1,null,2}

The invalid sets are not viable since their placement in the  $N \times N$  grid would conflict with the requirement of having **unique** values per row and column, similar to Latin Squares. Also, Singular sets (e.g. {1}, {3}) are not considered viable sets since there is no associated arithmetic operation.

How do you generate viable sets? How many are there for a 4x4 puzzle? How many are there for a 6x6 or a 9x9 puzzle? Well, consider viable sets in the context of their arithmetic operation for 4x4 puzzles. Notice that addition and multiplication can utilize 2, 3, or 4 numbers; however, subtraction and division, naturally, are limited to two numbers. Moreover, a set *always* produces a natural number, regardless of their order. Yet, it is important to have both sets (e.g. {2,6} and {6,2}) in order to solve a puzzle. Consider the following examples.

**Addition & Multiplication**                    {1,2}; {1,2,3}; {1,2,3,4}; {1,2,3,2}

**Subtraction**                                        {2,6} and {6,2}  $\rightarrow$  4, not -4

**Division**    {2,6} and {6,2}  $\rightarrow$  3, not 0.33

    {1,2} and {2,1}  $\rightarrow$  2, not 0.50

Computing the number of viable sets is not a straight-forward combinatorial problem. However, the following formula seems to work for Kenken puzzle of **any** size.

$$\sum d(d-1)^{n-1}$$

where     $\Sigma$  denotes the aggregate sum such that  $d=2$  to  $d$ ; such that  
 $d$  denotes the dimension of the puzzle; and,  
 $n$  denotes the number of integers in a set, ranging from 2 to  $n$ .

Consider the cardinality of viable sets for  $N \times N$  Kenken puzzles:

- **4x4 Kenken Puzzle**

$$[4(4-1)^{2-1}] + [4(4-1)^{3-1}] + [4(4-1)^{4-1}] = 156 \text{ Viable Sets}$$

- **5x5 Kenken Puzzle**

$$[5(5-1)^{2-1}] + [5(5-1)^{3-1}] + [5(5-1)^{4-1}] + [5(5-1)^{5-1}] = 1,700 \text{ Viable Sets}$$

- **6x6 Kenken Puzzle**

$$[6(6-1)^{2-1}] + [6(6-1)^{3-1}] + [6(6-1)^{4-1}] + [6(6-1)^{5-1}] + [6(6-1)^{6-1}] = 23,430 \text{ Viable Sets}$$

- **8x8 Kenken Puzzle**

$$[8(8-1)^{2-1}] + [8(8-1)^{3-1}] + [8(8-1)^{4-1}] + [8(8-1)^{5-1}] + [8(8-1)^{6-1}] + [8(8-1)^{7-1}] + [8(8-1)^{8-1}] = 7,686,392 \text{ Viable Sets}$$

- **9x9 Kenken Puzzle**

$$[9(9-1)^{2-1}] + [9(9-1)^{3-1}] + [9(9-1)^{4-1}] + [9(9-1)^{5-1}] + [9(9-1)^{6-1}] + [9(9-1)^{7-1}] + [9(9-1)^{8-1}] + [9(9-1)^{9-1}] = 172,565,640 \text{ Viable Sets}$$

- **10x10 Kenken Puzzle** → Over 4.3 Billion Viable Sets!!

It is no wonder why Kenken puzzles pose a formidable challenge. Imagine trying to conceptualize possible viable sets for a cage consisting of five cells, for example, in attempting to solve a 6x6 puzzle. How many viable sets are there? Well, there are 23,430 viable sets for 6x6 puzzles. However, there are *only* 3,750 viable sets that may be considered for a 5-cell cage. That's a lot of possibilities. Even worse, try conceptualizing what those sets are, such as: {1,2,3,1,5} and {2,3,4,3,6}. But, wait! You must consider the arithmetic operation and the total amount, which means a subset of the 3,750 viable sets. But, what are they? The daunting task worsens knowing that you must select a viable set that computes to the assigned total based on the respective arithmetic operation, as well as attempting to satisfy the Latin Square requirement.

: : :

## GENERATING VIABLE SETS

Consider the following method for generating viable sets for a 4x4 puzzle. The method actually determines those sets that must be discarded. Also, keep in mind that you are always considering a four items.

- First two numbers are equal {1,1,null, null}
- Second and third numbers are equal and not null {1,2,2,null}
- Third and fourth numbers are equal and not null {1,3,1,1}
- Third number is null and fourth number is not null {1,3,null,2}

The following Data step implements this method using DO loops such that the aforementioned rules are placed in the inner DO loop. Of the four DO loops, notice that the third and fourth loops include the Null-value as part of its list of enumerated values.

```
data p4sets;
  do n1=1,2,3,4;
    do n2=1,2,3,4;
      do n3=.,1,2,3,4;
        do n4=.,1,2,3,4;

          < Determine Viability Of Set >
          < If Viable, Then Compute Totals >
```

```

end;
end;
end;
end;
run;

```

For a 6x6 Kenken puzzle the above SAS code would require two more DO loops such that all the DO loops would have enumerated values up to its dimensional limit (i.e. 6), and so on for larger puzzles. Certainly, the SAS Macro Language could be used to generate the appropriate Data step that would create the collection of viable sets for any puzzle. Then, if a set is viable, the innermost DO loop would compute the four totals representing the arithmetic operations, as follows:

- **Addition** – Simply use the SUM function, as follows.

```
add = sum(of n1-n4);
```

- **Multiplication** – Requires more effort because an assignment statement in SAS returns a missing value whenever one or more of its operands contains a missing value.

```

if n2 eq .
then mult = n1;
else if n3 eq .
then mult = n1 * n2;
else if n4 eq .
then mult = n1 * n2 * n3;
else mult = n1 * n2 * n3 * n4;

```

- **Division & Subtraction** – Can be performed when *only* the first two numbers are not null. Even then, the assignment statement is performed depending on the order of the values of the numbers such that the result is always a positive number; otherwise, the result is for both operations.

```

if n3 eq . and n4 eq .
then do;
  if n1 gt n2
  then subtr = n1 - n2;
  else subtr = n2 - n1;
  if n1 gt 2 and int(n1/n2) eq n1/n2
  then div = n1 / n2;
  else if n2 gt n1 and int(n2/n1) eq n2/n1
  then div = n2 / n1;
end;
else do;
  subtr = .;
  div = .;
end;

```

Similar to the DO loops, for larger puzzles, the assignment statements would need to be altered accordingly in order to include the extra operands. For multiplication, the Nested IF statement would require considerable adjustment. For subtraction and division, however, only the IF clause would be expanded to make sure that all numbers except for N1 and N2 would be null; otherwise, the code remains in tact.

Recall that there are 156 viable sets for a Kenken puzzle. However, how many of those sets apply to subtraction and division? Or, how many sets have three operands that are used for addition? In other words, what is the distribution with respect to the content of those sets having missing / non-missing values? Consider the following table that shows more detailed information about the viable sets.

## Viable Sets for 4x4 KenKen Puzzles

Cells				Add		Subtract		Multiply		Divide	
1	2	3	4								
Not Null	Not Null	Null	Null	12	7.69	12	100.0	12	7.69	8	100.0
Not Null	Not Null	Not Null	Null	36	23.08	.	.	36	23.08	.	.
Not Null	Not Null	Not Null	Not Null	108	69.23	.	.	108	69.23	.	.
				=====	=====	=====	=====	=====	=====	=====	=====
				156	100.0	12	100.0	156	100.0	8	100.0

Notice that there are three types of sets for a 4x4 KenKen puzzle, which makes sense since a cage can have no more than 4 cells of which the first two cells are not null. There are only 12 viable sets that are applicable to subtraction and only 8 viable sets for division. Reasonably, there are 156 viable sets for addition and multiplication; moreover, they have the same distribution with respect to actual and null values. Finally, it is reasonable that the number of sets increases with the number of cells.

### IMPLEMENTING A STRATEGY

When playing by the rules, solving KenKen puzzles requires logic, arithmetic, even some algebra, and lots of trial and error. Hence, the proposed strategy for solving KenKen puzzles abandons the rules outright. Instead, let's consider an obvious fact: *A solution to a puzzle consists of a unique collection of viable sets representing the several cages.* Also, we can determine the viable sets for any cage consisting of  $N$ -cells and an associated arithmetic operator. Therefore, if we perform a Cartesian product on those collections of viable sets, then one of those joins must be the solution to the puzzle. Consequently, programmatically, we can process each conglomerate of viable sets, representing all cages, and test whether it conforms to a Latin Square – single instance of row and column values. Ironically, the hard part of determining a viable set that might contribute to the solution becomes a non-issue.

### THE SAS SOLUTION

The SAS code that implements the proposed strategy is a bit involved. Keep in mind that the following code is part of a SAS macro. Using the data file discussed earlier, initially, it is necessary to determine the number of cages, which is easily done using the SQL procedure with the INTO operator.

```
proc sql noprint;
    select count(*) into :ncages from puzzle;
quit;
```

Next we must obtain the viable sets for each cage. Knowing that each record in the data set PUZZLE represents a cage, we use the %DO loop to process each cage performing two functions: creating macro variables and obtaining the viable sets. The Data \_null\_ step creates macro variables representing the number of cells, the arithmetic operation, and the total, as shown below.

```
%do i = 1 %to &ncages.;
    data _null_;
        set puzzle(where=(cage eq "C&i."));
        call symput('total', trim(left(put(total,best.))));
        call symput('ncells', trim(left(put(ncells,best.))));
        select(oper);
            when('+') call symput('oper','ADD');
            when('-') call symput('oper','SUBT');
            when('*') call symput('oper','MULT');
            when('/') call symput('oper','DIV');
            otherwise call symput('oper','NO');
        end;
    run;
```

Fortunately, the design of the PUZZLE data set facilitates the code needed to obtain the desired sets. Given one of four arithmetic operations, the %THEN %DO block generates a SQL step such that the WHERE clause subsets the total set of Viable sets, accordingly. Notice the macro variable &PSIZE, which denotes the size of the puzzle, which resolves to the value 4 when processing 4x4 puzzles. For instance, the FROM clause accesses a permanent data set called, for example, P4SETS, which contains all 156 Viable sets for 4x4 puzzles. This same macro variable is used in several %DO loops, as well. For our discussion, let's assume that we're processing a 4x4 puzzle.

```
%if "&oper." ne "NO"
  %then %do;
    proc sql;
      create table c&i._vsets as
      select %do j = 1 %to %eval(&psize.-1);
        n&j. as c&i.n&j., %end; n&j. as c&i.n&j.
      from kk.p&psize.sets
      where &oper. eq &total.
        and n(%do j = 1 %to %eval(&psize.-1); n&j., %end; n&j.) eq &ncells.;
      quit;
    %end;
```

In the event that there is a Singular cage, the %ELSE %DO block generates a simple Data step that creates a Viable set emulating those Viable sets representing non-Singular cages.

```
%else %do;
  data c&i._vsets;
    retain c&i.n1 &total. %do j = 2 %to &psize.; c&i.n&j. %end; .;
  run;
%end;
%end;
```

Since the cages are defined vertically in the PUZZLE data set, it is necessary to create a single observation data set, called CAGES, that will be employed in the next step as a 2-dimensional table, as follows: c1c1-c1c4, c2c1-c2-c4, c3c1-c3c4, etc, for as many as there are cages. As mentioned earlier, the solution is a bit involved.

```
data cages;
  array cages{&ncages.,&psize.} $3
    %do i = 1 %to &ncages.; c&i.c1-c&i.c&psize. %end; ;
  retain %do i = 1 %to &ncages.; c&i.c1-c&i.c&psize. %end; ;
  array cells{*}$ c1-c&psize.;
  set puzzle end=eof;
  do i = 1 to dim(cells);
    cages{_n_,i} = cells{i};
  end;
  if eof then output;
  keep %do i = 1 %to &ncages.; c&i.c1-c&i.c&psize. %end; ;
run;
```

Finally, we're able to evaluate the aggregate collection of Viable sets obtained from the Cartesian product. The information needed is available either as macro variables or two data sets: CAGES and VSETS, the latter containing the sets to be evaluated. Notice that the Data step reads CAGES only once; thus preserving the 2-dimensional array called CELLS that is used during the evaluation process. In fact, there are two other 2-dimensional arrays: VSETS that contains the aggregate collection of viable sets to be evaluated; and, PUZZLE that represents the Kenken grid.

```
data solution;
  array cells{&ncages.,&psize.} $3
    %do i = 1 %to &ncages.; c&i.c1-c&i.c&psize. %end; ;
  array vsets{&ncages.,&psize.}
    %do i = 1 %to &ncages.; c&i.n1-c&i.n&psize. %end; ;
  array puzzle{&psize.,&psize.}
    %do i = 1 %to &psize.; r&i.c1-r&i.c&psize. %end; ;
  if _n_ eq 1
    then set cages;
  set vsets;
  do cage = 1 to &ncages.;
    do item = 1 to &psize.;
      if cells{cage,item} ne ''
        then do;
```



```

        row = input(scan(cells{cage,item},1,','),best.);
        col = input(scan(cells{cage,item},2,','),best.);
        puzzle{row,col} = vsets{cage,item};
    end;
end;
end;

```

After the PUZZLE matrix is populated, the Data step proceeds to evaluate the grid. Every observation in the VSETS data set is assumed to be the solution, that is, until proven otherwise. Fundamentally, the following code is checking whether the Latin Square criterion is maintained. If there's any instance of having the same number occur more than once in a row or column, then the variable SOLUTION is re-assigned the value zero, which denotes a false condition in SAS. If there is a solution found, then that observation is written out and the Data step terminates immediately.

```

solution=1;
do row = 1 to &psize.;
    do col = 1 to &psize.;
        value = puzzle{row,col};
        if value eq .
            then solution=0;
        else do;
            do i = 1 to &psize.;
                if (i ne col and puzzle{row,i} eq value)
                    or (i ne row and puzzle{i,col} eq value)
                        then solution=0;
            end;
        end;
    end;
end;
if solution
    then do;
        output;
        stop;
    end;
keep %do i = 1 %to &psize.; r&i.c1-r&i.c&psize. %end; ;
run;

```

## THE SAS SOLUTION – WHY IT FAILS

In theory, the solution works. In fact, the SAS solution works very well for 4x4 puzzles. However, unless you have access to a supercomputer, the solution is not practical. Ironically, the solution fails for the same reason that it works, namely, the Cartesian product. For example, consider a 9x9 puzzle that employs all four operations and consists of around twenty-five cages. Now let's assume that several of those cages consist of 3, 4, or 5 cells, which automatically implies that their totals are obtained by either addition or multiplication. Just imagine the number of possible viable sets for those cages, not to mention the more singular and two-cell cages. Recall that 9x9 puzzles have **172,565,640** Viable sets of which many of these would be considered for the larger cages. And, here is where it fails -- The Cartesian product of all these Cage-specific sets becomes HUGE, thus impractical. Even if the Cartesian product is attainable, the process of evaluating each aggregate collection of viable sets as a possible solution only enhances its limitations.

## CONCLUSION

There are several lessons to be learned. First of all, the concept and cardinality of viable sets affords a better understanding of Kenken puzzles, as well as distinguishing it from its enigmatic cousin, Sudoku. Secondly, the notion of pre-ordained cage-specific viable sets was crucial in realizing that there must be a unique aggregate collection of these sets that represents the solution to the puzzle. Also, the Cartesian product of cage-specific sets, in theory, generates the solution, albeit amidst many other aggregate collections; whereupon, each collection is evaluated to determine whether it meets the Latin Square criterion, of which there is only one. Finally, however, the solution fails for practical reasons for even SAS indexes would not be used from such outrageous SQL-joins. And the big lesson – Back to drawing board.

## REFERENCES

Davis, Tom; "Kenken For Teachers" <http://www.geometer.org/mathcircles> March, 2010.

Fisher, Dave; "Solving Kenken Logic Puzzles:

[http://puzzles.about.com/od/logicpuzzles/ss/KenKen\\_tutorial\\_easy\\_8.htm](http://puzzles.about.com/od/logicpuzzles/ss/KenKen_tutorial_easy_8.htm).

Stuart, Andrew; "6x6 Kenken Solver (Beta)" <http://www.sudokuwiki.org/kenken6x6.asp>.

Mathdoku Web Site <http://www.mathdoku.com/?level=easy>.

Wikipedia, The Free Encyclopedia: "Latin Square" [http://en.wikipedia.org/wiki/Latin\\_square](http://en.wikipedia.org/wiki/Latin_square).

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name:	John R. Gerlach
Enterprise:	SAS / CDISC Analyst
Address:	
City, State ZIP:	Hamilton, NJ 08619
Work Phone:	609-672-5034
Fax:	
E-mail:	<a href="mailto:jrgerlach@optonline.net">jrgerlach@optonline.net</a>
Web:	

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.