

The Pegboard Game: A Recursive SAS[®] Macro Solution

Houliang Li, Frederick, MD

ABSTRACT

Rick Langston of SAS Institute Inc. provided an iterative SAS program to solve the famous Pegboard game during SUGI 30. Since this game is generally considered a recursion problem, a recursive solution is in order. This presentation describes recursion and the depth-first search algorithm, and how they are used to find all possible solutions to the Pegboard game regardless of the starting position. The highly sophisticated yet succinct macro program clearly demonstrates the elegance, utility, and power of recursion, which is fully backed by the SAS Macro Language.

INTRODUCTION

The Pegboard game is a triangular board with 15 holes on it. The holes are arranged in a triangular pattern, like this:

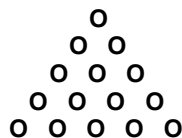


Figure 1

For practical purposes, we will number each hole with a unique, non-zero hexadecimal digit, i.e., 1 – 9 and a – f. We will use the capital letter O to represent an empty hole, and the capital letter X to represent a hole with a peg in it. Figure 2 shows an empty Pegboard.

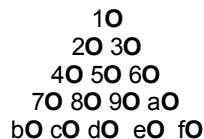


Figure 2

At the start of a game, pegs occupy 14 of the 15 holes. The single empty hole, called the starting hole, can be any of the 15 holes. We will refer to an empty hole as Hole-N, where N is the assigned hexadecimal digit, for example, Hole-1. If there is a peg in a hole, however, we will refer to the hole and peg together as Peg-N, such as Peg-2. Figure 3 illustrates a typical lineup of the pegs at the start of a game, represented by Hole-1 and Peg-2 to Peg-f.

(Please note that the Hole-N and Peg-N designations only refer to the starting status of the holes. They are used to explain the rules of the game. Once the pegs start moving, we will no longer use this system to identify a hole's status.)

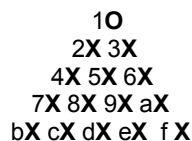


Figure 3

The rules of this one-player game are simple: any peg, Peg-J, may jump over an adjacent peg, Peg-K, only when there is a free hole, Hole-L, on the other side of, and immediately adjacent to, Peg K. In other words, Peg-J, Peg-K and Hole-L must form a straight line with no other pegs or empty holes in between for this move to be legal. The jumped peg, Peg-K, is removed. The goal of the game is to remove all but one of the pegs by repeatedly jumping pegs over one another. The last peg standing makes you a winner.

Given the lineup in Figure 3, on the first move, Peg-4 can jump over Peg-2 to Hole-1, resulting in the removal of Peg-2. Figure 4 shows the new lineup after the first move. We will use 421 to represent such a move, where the first

hexadecimal digit is the jumping peg, the second digit is the peg being jumped over, and the third digit is the targeted free hole. If we move Peg-6 instead of Peg-4, we will get a different lineup. This alternative move can be represented as 631.

```

      1X
     2O 3X
    4O 5X 6X
   7X 8X 9X aX
  bX cX dX eX f X

```

Figure 4

ALGORITHMS

Recursion is a programming technique whereby a function, such as a SAS macro, calls itself inside the function body with only a part of the original problem, and the self-calling process continues until the problem is solved.

In order to use recursion, a problem should be logically suitable for a “divide and conquer” strategy, and the new, smaller problem(s) resulting from the division should resemble the original problem in nature. Only the scale or scope is reduced with each division. Specifically, each time a recursive function calls itself, the new invocation tackles an increasingly smaller part of the original problem. The cycle of dividing and recursive calling continues until an answer is readily available to the newest, and usually smallest, problem. This is called the base case. At this point, no more division is necessary, or even possible, and the process starts to backtrack to the original function call. The final solution may be apparent when the base case is encountered, or it may not be available until the backtracking is completed.

Take the familiar problem of factorial calculations. To get the factorial of a positive integer N , we can divide the problem into the number N and the factorial of $(N - 1)$, the product of which yields the desired factorial, that is, $N! = N * (N - 1)!$. For the new factorial $(N - 1)!$, we can further divide it into the number $(N - 1)$ and the factorial of $((N - 1) - 1)$, the product of which yields $(N - 1)!$, i.e., $(N - 1)! = (N - 1) * (N - 2)!$. And so on, until we reach the base case where the factorial is reduced to $1!$. Since $1! = 1$, we already have the answer, so no more division of the problem is necessary. We then backtrack to the original factorial of N , multiplying all the isolated numbers along the way: $1, 2, 3, \dots, N - 2, N - 1, N$. This gives the factorial we want: $N! = 1 * 2 * 3 * \dots * (N - 2) * (N - 1) * N$.

The Pegboard game is a good candidate for the “divide and conquer” strategy. Assuming we can solve it, there must be exactly 13 moves before we have only one peg left, since each move eliminates one peg and we start with 14. If we look at the game from the perspective of how many pegs still remain on the board, we can see that each move reduces the problem to a smaller one, until we have only two adjacent pegs left (the base case!). At that point, we can make our move and end the game – we found a solution! Unfortunately, the more probable outcome of our successive moves will be two or more pegs still on the board but no more legal moves! That is also a base case, but we have lost the game. Obviously, at each stage of the game, we can make one legal move, and then call the recursive function with the new game state (the new peg lineup). The new function call in turn makes one legal move, and then calls the function again with an even smaller problem. The whole process repeats itself until we either find a solution or run out of legal moves with more than one peg left.

The fact that we may end up with multiple pegs on the board yet no legal moves points to the complexity of the problem. Unlike a factorial calculation which always has a single answer, the Pegboard game actually has thousands of unique solutions for each starting lineup, and, of course, many, many more unsuccessful attempts! Take the starting lineup in Figure 3. We can do either 421 or 631. With either move, we will create different possibilities for the next move. You may consider these two starting moves as mirror images of each other, but it is a mere coincidence. If the starting hole is Hole-2 instead of Hole-1, then the two possible moves, 742 and 952, are no longer mirror images. In any case, as the game progresses, each possible move creates distinct consequences of its own, with some sequences leading to success and most pointing to failure!

To handle the complexity due to multiple possibilities on each move, we need to adopt the depth-first search algorithm in our recursive implementation. The partial tree diagram in Figure 5 demonstrates the strategy. Each numbered node on the tree represents a game state, or peg lineup. Each downward line linking two nodes represents a legal move and is called a branch. The topmost node, Node1, is called the “root” and represents the peg lineup at the start of the game. A continuous sequence of linked branches, starting from the root and proceeding through successive nodes without going up, is called a search path. The node at the end of a path represents an end state and is called a leaf. Obviously, a leaf can be either a solution or a failure.

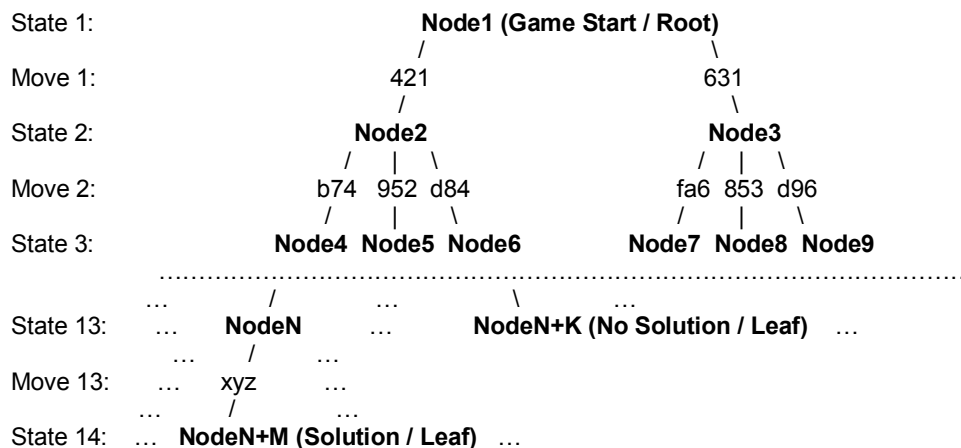


Figure 5

Simply put, the depth-first search algorithm starts with the root, follows the first branch at each node to go to the next node until a leaf is found, and examines the leaf to see if it is a solution. Then it backtracks one step to the nearest non-leaf node and follows the next branch. When all branches of this non-leaf node have been explored, the search backtracks one step further and repeats the process, until all paths from the root to all leaves have been traveled. Imagine the tree in Figure 5 with only three states, or levels, rather than 14. A depth-first search starts from Node1 (the root), goes to Node2 by following the first branch of Node1, then goes to Node4 by following the first branch of Node2, examines Node4 (a leaf), and then backtracks to Node2. The search then goes to Node5 by following the next possible branch of Node2, examines Node5 (another leaf), and backtracks to Node2. It then goes to Node6, examines it and backtracks to Node2 again. Now that every branch of Node2 has been explored, the search backtracks one more step to Node1, and continues with the next possible branch of Node1. A complete search of the 3-level tree, in order of nodes visited, will be:

```
Node1 → Node2 → Node4 → Node2 → Node5 → Node2 → Node6 → Node2 →
Node1 → Node3 → Node7 → Node3 → Node8 → Node3 → Node9 → Node3 →
Node1
```

Please note that Node9 is the last leaf visited on our 3-level tree, but the search did not stop at Node9. This is because in a depth-first search, we do not know whether we have examined every leaf of the tree until we go back to the root and find that no more branches remain to be explored. The goal is get to the first leaf, and every subsequent leaf, as soon as possible, hence the name “depth-first.” Counting branches before using them does not help with that goal. The algorithm works well when we want to find all solutions to a problem because the entire tree is searched. It truly shines when our objective is to find the first solution – we can stop the search as soon as a leaf satisfies our criteria.

PROGRAM DESIGN

The Pegboard game is two-dimensional. As a result, it is very difficult to keep track of all the legal moves and constantly changing peg lineups following these moves. By transforming it into a one-dimensional system, we can use character strings to represent the two key elements of the game: legal moves and peg lineups.

Earlier in **INTRODUCTION**, we used 421 to represent the move of Peg-4 over Peg-2 to Hole-1. By the same definition, the following 36 moves are the complete set of potential legal moves allowed by the Pegboard game rules:

```
124 247 47b b74 742 421 136 36a 6af fa6 a63 631 bcd cde def fed edc dcb
358 58c c85 853 259 59e e95 952 789 89a a98 987 69d d96 48d d84 456 654
```

For peg lineups, we will use a 15-character string to represent the status of the 15 holes. The first character always represents hole number 1, the second character always represents hole number 2, and the third character represents hole number 3, etc. At any stage of the game, if a certain hole is empty, e.g., the hole with the assigned hexadecimal digit N, then the Nth character in the string will be 0. However, if there is a peg in the same hole, then the Nth character will be a non-zero hexadecimal digit. Using this representation scheme, the peg lineup in Figure 3 becomes **023456789abcdef**, as only Hole-1 is empty.

If we number the pegs the same as the holes they occupy at the start of a game, we can track which hole's initial occupant becomes the winning peg, as well as each peg's whereabouts during the game. In the 15-character string representation, this can be done by replacing a target free hole's 0 with the jumping peg's own number. For example, when we make the first move of 421 to the lineup in Figure 3, we change the fourth character and the second character (the "4" and "2" parts of 421) into 0s, and replace the first character (the "1" part of 421) with the number 4. The resultant lineup in Figure 4 becomes **403056789abcdef**, as holes numbered 2 and 4 are now empty and the starting hole gets the peg from the initial Peg-4. Similarly, if the first move is 631, then the new lineup will be **620450789abcdef**.

(Using the depth-first search algorithm, the very first solution to the peg lineup in Figure 3 has the following moves: 421 → b74 → 952 → 247 → c85 → edc → 358 → a63 → 136 → 789 → 69d → cde → fed. The final peg lineup is **00000000000f00**, which means that Peg-f is the last peg standing in the hole numbered d.)

Now that we have resolved the game representation issues, it is time to formulate an overall strategy for the recursive macro implementation. In a combination of recursion's "divide and conquer" approach and the depth-first search algorithm, we will make each recursive macro call loop through all 36 potential legal moves. Whenever an actual legal move is found for the current peg lineup, the macro will make that move by updating the peg lineup, and then call the recursive macro again with the new lineup, essentially handing off the smaller problem to the new macro call. This cycle of "move and recursive call" continues until either a final winning move is found or no moves are possible with two or more pegs left on the board. In either base case, execution backtracks to the current macro's calling macro and continues with the next legal move, until all possible moves are exhausted. The whole process can have up to 13 levels of recursive calling and backtracking before it finds all solutions.

Looking back at Figures 3, 4, and 5, it becomes clear that we had the benefit of visually examining the peg lineup and easily finding the first possible moves: 421 and 631. It got a little more complicated on the second move. Had we continued with the manual process, we would soon be overwhelmed by the exponential growth of possibilities! A recursive implementation does not benefit from such visual cue and initial efficiency. It would never know to start with Peg-4 or Peg-6, because the starting hole is not necessarily Hole-1. However, its brute force more than makes up for this minor disadvantage. By systematically checking all 36 move possibilities for every (new) peg lineup and following every legal move to the next stage, a recursive function ensures that every leaf of the tree is reached and examined. This exhaustive check of move possibilities may seem a waste of time at first glance, but is actually extremely quick with very little overhead in terms of resources. Whenever the recursive function goes down a branch, the manual process by a human player will have to do the same in order to be thorough.

IMPLEMENTATION

The recursive macro is defined with four parameters:

```
%macro pegboard (start=, moves=, backtostart=NO, allsolutions=NO);
```

The parameter `start` provides the peg lineup for each function call and is always 15 characters long. `moves` keeps track of all legal moves made so far, so it defaults to blank at the initial macro call. The two flags, `backtostart` and `allsolutions`, default to NO. You can change `backtostart` to YES if you only want solutions where the final peg ends up in the starting hole. Similarly, change `allsolutions` to YES if all solutions to a particular starting lineup are desired. Beware that certain starting lineups can take quite a long time to find all solutions due to the sheer number of them, and not all starting lineups are able to put the winning peg in the starting hole.

The next block of code takes care of the initial setup:

```
%local index;

%if %length(%sysfunc(compress(&start, 0))) = 14 %then %do;
  %global originalstart starthole success legalmoves;
  %let originalstart = &start;
  %let starthole = %index(&start, 0);
  %let success = 0;
  %let legalmoves = 124|247|47b|b74|742|421|136|36a|6af|fa6|a63|631|bcd|cde|def|
                    fed|edc|dcb|358|58c|c85|853|259|59e|e95|952|789|89a|a98|987|
                    69d|d96|48d|d84|456|654;
%end;
```

The conditional macro statement makes sure that the four global macro variables are created and initialized only once – at the start of a game. Only the starting lineup has 14 non-zero hexadecimal digits in the string; subsequent

recursive macro calls will not try to change those macro variables when they start execution. The macro variable `success`, however, can be reset to 1 elsewhere when a solution is found. It is important to remember that every time a recursive macro is invoked, all the statements in the macro will be run through with the new parameter values and effective local and global macro variables, just like any other SAS macros. Only the intended statements and macro variables should be affected. By the same token, the macro variable `index` is declared local for every recursive call so that it can keep a separate record of the iterations over the possible legal moves during each macro's execution.

Next comes the exhaustive looping over possible legal moves:

```
%do index = 1 %to 36;
  %if not &success or &success and %upcase(&allsolutions) = YES %then %do;
    <other statements>
  %end;
%end;
```

The `%do` loop applies each of the 36 potential legal moves to the current peg lineup and hopes to get lucky. If one is found, the enclosed statements, discussed below, will make the move and call the macro again. In case no legal move is possible and the current peg lineup is not a solution, the loop will naturally exit so that the currently executing macro returns to its calling macro, which will try its next legal move. Each recursive macro execution at each of the 13 levels will continue looping until the first solution is found or all solutions are found, depending on the flag value of `allsolutions`.

The following code segments are the key parts of the recursive implementation, enclosed within the conditional statement inside the `%do` loop above:

```
%let nextmove = %scan(&legalmoves, &index, |);
%let first = %substr(&nextmove, 1, 1);
%let second = %substr(&nextmove, 2, 1);
%let third = %substr(&nextmove, 3, 1);

%if &first > 9 %then
  %let first = %eval(%sysfunc(rank(&first)) - 87);
%if &second > 9 %then
  %let second = %eval(%sysfunc(rank(&second)) - 87);
%if &third > 9 %then
  %let third = %eval(%sysfunc(rank(&third)) - 87);
```

First, the next potential legal move is retrieved from the `legalmoves` macro variable. Then, its components are extracted, yielding the absolute positions of the jumping peg, the jumped peg, and the target free hole in the 15-character peg lineup string. Conversion from hexadecimal digits to decimal equivalents is performed for those components with values between `a` and `f`.

```
%if %substr(&start, &first, 1) > 0 and %substr(&start, &second, 1) > 0 and
  %substr(&start, &third, 1) = 0 %then %do;

  %let allmoves = &moves &first-&second-&third;

  %if &first = 1 %then
    %let newstart = 0%substr(&start, 2);
  %else %if &first = 15 %then
    %let newstart = %substr(&start, 1, 14)0;
  %else
    %let newstart = %substr(&start, 1, &first - 1)0%substr(&start, &first + 1);

  %let newstart = %substr(&newstart, 1, &second-1)0%substr(&newstart, &second+1);

  %if &third = 1 %then
    %let newstart = %substr(&start, &first, 1)%substr(&newstart, 2);
  %else %if &third = 15 %then
    %let newstart = %substr(&newstart, 1, 14)%substr(&start, &first, 1);
  %else
    %let newstart = %substr(&newstart, 1, &third - 1)%substr(&start, &first, 1)
```

```

        %substr(&newstart, &third + 1);

%if %length(%sysfunc(compress(&newstart, 0))) = 1 and
    (%upcase(&backtostart) = NO or %upcase(&backtostart) = YES and
    %substr(&newstart, &starthole, 1) > 0) %then %do;
    %let success = 1;
    %put Starting position is: &originalstart;
    %put Successful moves are: &allmoves;
%end;
%else
    %pegboard(start=&newstart, moves=&allmoves, backtostart=&backtostart,
        allsolutions=&allsolutions);

%end;

```

Next, the current peg lineup is analyzed to see if the proposed move is allowable. If it is not, the current iteration is finished and execution moves to the next iteration, i.e., checking the next potential legal move. On the other hand, if the proposed move is legal, the current peg lineup will be updated to reflect the move, which is also added to the sequence of moves performed so far.

The last `%if-%else` block in the above code checks to see if the updated peg lineup is a solution, taking into consideration whether the user wants just any game solution or a solution that places the last peg in the starting hole. If it is determined to be a valid solution, the `success` flag is reset to 1, and all the peg moves from game start to conclusion are output to the log in chronological order. If we want to store the solutions in a permanent dataset, here is the place to insert the code. If the updated peg lineup is not yet a solution, more work needs to be done. A recursive macro call is then placed with the updated peg lineup and other parameters.

If you are strongly tempted to check the new lineup to see if other legal moves are possible before passing it on to the next macro call, you apparently do not appreciate the recursive implementation. Remember, when a new macro call starts execution, it does exactly one round of 36 checks and, whenever a legal move is found, instantly makes that move, and then immediately hands off the new lineup to the next macro call, which will do exactly the same things all over again. There may be other legal moves in addition to the current one, but they can wait until the new macro call, launched after the current move, returns. Obviously, this new macro call will similarly spawn its own multiple levels of new macro calls. In time, they will all return to the initial recursive call.

RUNTIME SNAPSHOTS

The first few runtime snapshots may help you understand the implementation better. We assume the starting lineup is **023456789abcdef** and the other parameters have default values. For practical purposes, we do not consider the initial game start macro to be a recursive macro call, even though technically it is, because the macro definition is recursive.

The table below shows all the user-defined macro variables when the macro execution just finished the initial setup, before the looping starts:

Global Macro Variables – Game Start	
Originalstart	023456789abcdef
Starthole	1
Success	0
Legalmoves	124 247 47b b74 742 421 136 36a 6af fa6 a63 631 bcd cde def fed edc dcb 358 58c c85 853 259 59e e95 952 789 89a a98 987 69d d96 48d d84 456 654
Local Macro Variables – Game Start	
Start	023456789abcdef
Moves	
Backtostart	NO
Allsolutions	NO
Index	

Figure 6

The next table displays all the macro variables after the first legal move has been made and the peg lineup updated, right before the first recursive call:

Global Macro Variables – Game Start	
originalstart	023456789abcdef
Starthole	1
Success	0
legalmoves	124 247 47b b74 742 421 136 36a 6af fa6 a63 631 bcd cde def fed edc dcb 358 58c c85 853 259 59e e95 952 789 89a a98 987 69d d96 48d d84 456 654
Local Macro Variables – Game Start	
Start	023456789abcdef
Moves	
backtostart	NO
allsolutions	NO
Index	6
Nextmove	421
First	4
Second	2
Third	1
Allmoves	4-2-1
Newstart	403056789abcdef

Figure 7

The table below lists all the macro variables when the first recursive macro call, launched by the game start macro, has started execution and is about to enter its %do loop:

Global Macro Variables – First Recursive Macro Call	
originalstart	023456789abcdef
Starthole	1
Success	0
legalmoves	124 247 47b b74 742 421 136 36a 6af fa6 a63 631 bcd cde def fed edc dcb 358 58c c85 853 259 59e e95 952 789 89a a98 987 69d d96 48d d84 456 654
Local Macro Variables – Game Start	
Start	023456789abcdef
Moves	
backtostart	NO
allsolutions	NO
Index	6
Nextmove	421
First	4
Second	2
Third	1
Allmoves	4-2-1
Newstart	403056789abcdef
Local Macro Variables – First Recursive Macro Call	
Start	403056789abcdef
Moves	4-2-1
backtostart	NO
allsolutions	NO
Index	

Figure 8

The next table displays all the macro variables after the first recursive macro call has made its first legal move and updated the peg lineup, right before it invokes the first recursive call of its own. To the currently executing macro, this legal move is a first, but cumulatively, it is the second legal move on the starting lineup.

Global Macro Variables – First Recursive Macro Call	
Originalstart	023456789abcdef
Starthole	1
Success	0

Legalmoves	124 247 47b b74 742 421 136 36a 6af fa6 a63 631 bcd cde def fed edc dcb 358 58c c85 853 259 59e e95 952 789 89a a98 987 69d d96 48d d84 456 654
Local Macro Variables – Game Start	
Start	023456789abcdef
Moves	
Backtostart	NO
Allsolutions	NO
Index	6
Nextmove	b74
First	11
Second	7
Third	4
Allmoves	4-2-1 11-7-4
Newstart	403b56089a0cdef
Local Macro Variables – First Recursive Macro Call	
Start	403056789abcdef
Moves	4-2-1
Backtostart	NO
Allsolutions	NO
Index	4

Figure 9

The final table lists all the macro variables after the second recursive macro call has started execution and is ready to enter its %do loop:

Global Macro Variables – Second Recursive Macro Call	
Originalstart	023456789abcdef
Starthole	1
Success	0
Legalmoves	124 247 47b b74 742 421 136 36a 6af fa6 a63 631 bcd cde def fed edc dcb 358 58c c85 853 259 59e e95 952 789 89a a98 987 69d d96 48d d84 456 654
Local Macro Variables – Game Start	
Start	023456789abcdef
Moves	
Backtostart	NO
Allsolutions	NO
Index	6
Nextmove	b74
First	11
Second	7
Third	4
Allmoves	4-2-1 11-7-4
Newstart	403b56089a0cdef
Local Macro Variables – First Recursive Macro Call	
Start	403056789abcdef
Moves	4-2-1
Backtostart	NO
Allsolutions	NO
Index	4
Local Macro Variables – Second Recursive Macro Call	
Start	403b56089a0cdef
Moves	4-2-1 11-7-4
backtostart	NO
allsolutions	NO
Index	

Figure 10

As you can see, neither the initial game start macro nor the first recursive macro call has finished its execution before a new macro call is launched, and this recursive process goes deeper and deeper. No macro is actually finished until all its spawned macro calls are completed. To find a single Pegboard game solution, 12 levels of recursive macro calls are needed after the game starts, for a total of 13 levels of recursive invocations, with each level removing one peg from the board.

You probably have also noticed the addition of five new local macro variables for each level of recursive macro call: `start`, `moves`, `backtostart`, `allsolutions`, and `index`. If we continue with the next level of macro call, there will be another set of five local macro variables added, and so on. The bottom five local macro variables always belong to the currently executing macro, and the five variables above them belong to the macro that invoked the current macro. You can trace further up to see the local macro variables belonging to higher levels of macro calls, until you reach the game start macro. The SAS Macro Language fully supports recursion by enforcing local macro variable scopes of nested macro invocations, in this case 13 levels of nesting. The four global macro variables, `originalstart`, `starthole`, `success`, and `legalmoves`, as well as the six local macro variables belonging to the game start macro, `nextmove`, `first`, `second`, `third`, `allmoves`, and `newstart`, also retain or change their values accordingly.

SOLUTIONS

Running this recursive macro takes various amounts of time depending on the starting lineup, whether all solutions are desired, and whether you want the last peg to be in the starting hole. As Rick Langston correctly pointed out in his paper, the Pegboard game really has only 4 unique starting holes: Hole-1, Hole-2, Hole-4, and Hole-5. All other starting holes can be considered mirror images of the four. The table below lists the total number of unique solutions for each starting hole.

Starting Hole	Number of Solutions	Last-Peg-in-Starting-Hole Solutions
Hole-1, Hole-b, Hole-f	29760	6816
Hole-2, Hole-3, Hole-7, Hole-a, Hole-c, Hole-e	14880	720
Hole-4, Hole-6, Hole-d	85258	51452
Hole-5, Hole-8, Hole-9	1550	0

Figure 11

It is amazing to see the majority of solutions to games with starting holes in Hole-4, Hole-6, and Hole-d end up putting the last peg in the starting hole! Equally surprising is the lack of any such solutions to games with starting holes in Hole-5, Hole-8, and Hole-9.

Now that we have found all the solutions to the traditional Pegboard game, we can add a little twist to it. Can you modify the macro to find the fastest way to lose a game in each unique starting lineup? In other words, try to move the pegs in such a way that a maximum number of pegs remain on the board yet no legal moves are possible. How about doing everything all over again on a bigger Pegboard? You can add 6 more holes and pegs and follow the same rules. They should be a lot of fun!

CONCLUSION

Although SAS Institute Inc. does not provide technical support to recursion questions, the SAS Macro Language does fully support this programming technique in its current implementation. Some users may hesitate to rely on such a “non-official” feature, but there is really no reason to believe that SAS Macro Language will change the scoping rules in nested macro calls, of which the recursive macro call is a special case. The Pegboard game is just a fun example to demonstrate what is possible with recursion in SAS macro. You should take advantage of this technique whenever a problem is a good candidate for the “divide and conquer” strategy and does not involve a lot of complicated calculations. Where hardware resources are not a big concern, you should probably always use recursion if possible. You will learn to appreciate its elegance, utility, and power that programmers in other computer languages have long enjoyed.

REFERENCES

Langston, Rick. “The Pegboard Game: An Iterative SAS® Program to Solve It” *Proceedings of the Thirtieth Annual SAS® Users Group International Conference*. April 2005. <<http://www2.sas.com/proceedings/sugi30/066-30.pdf>>

CONTACT INFORMATION

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

Houliang Li
(301) 682-6832
houliang_li@yahoo.com

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 registered trademarks or trademarks of their respective companies.

APPENDIX

```
/*
Here is the complete Pegboard macro program. You can launch it using various
combinations of the parameter values. To obtain the sample solution provided in
section PROGRAM DESIGN, simply submit everything that follows.
*/
```

```
%macro pegboard (start=, moves=, backtostart=NO, allsolutions=NO);

%local index;

%if %length(%sysfunc(compress(&start, 0))) = 14 %then %do;

  %global originalstart starthole success legalmoves;
  %let originalstart = &start;
  %let starthole = %index(&start, 0);
  %let success = 0;
  %let legalmoves = 124|247|47b|b74|742|421|136|36a|6af|fa6|a63|631|bcd|cde|def|
                    fed|edc|dcb|358|58c|c85|853|259|59e|e95|952|789|89a|a98|987|
                    69d|d96|48d|d84|456|654;

%end;

%do index = 1 %to 36;

  %if not &success or &success and %upcase(&allsolutions) = YES %then %do;

    %let nextmove = %scan(&legalmoves, &index, |);
    %let first = %substr(&nextmove, 1, 1);
    %let second = %substr(&nextmove, 2, 1);
    %let third = %substr(&nextmove, 3, 1);

    %if &first > 9 %then
      %let first = %eval(%sysfunc(rank(&first)) - 87);
    %if &second > 9 %then
      %let second = %eval(%sysfunc(rank(&second)) - 87);
    %if &third > 9 %then
      %let third = %eval(%sysfunc(rank(&third)) - 87);

    %if %substr(&start, &first, 1) > 0 and %substr(&start, &second, 1) > 0 and
      %substr(&start, &third, 1) = 0 %then %do;

      %let allmoves = &moves &first-&second-&third;

      %if &first = 1 %then
        %let newstart = 0%substr(&start, 2);
      %else %if &first = 15 %then
        %let newstart = %substr(&start, 1, 14)0;
      %else
        %let newstart = %substr(&start, 1, &first -1)0%substr(&start, &first + 1);

      %let newstart = %substr(&newstart,1,&second-1)0%substr(&newstart, &second+1);

      %if &third = 1 %then
        %let newstart = %substr(&start, &first, 1)%substr(&newstart, 2);
      %else %if &third = 15 %then
        %let newstart = %substr(&newstart, 1, 14)%substr(&start, &first, 1);
      %else
        %let newstart = %substr(&newstart, 1, &third -1)%substr(&start, &first, 1)
          %substr(&newstart, &third + 1);
```

```
%if %length(%sysfunc(compress(&newstart, 0))) = 1 and
    (%upcase(&backtostart) = NO or %upcase(&backtostart) = YES and
    %substr(&newstart, &starthole, 1) > 0) %then %do;

    %let success = 1;
    %put Starting position is: &originalstart;
    %put Successful moves are: &allmoves;

%end;
%else
    %pegboard(start=&newstart, moves=&allmoves, backtostart=&backtostart,
        allsolutions=&allsolutions);

%end;

%end;

%end;

%mend;

%pegboard (start=023456789abcdef);
```