SESUG Paper 082-2021

May *i*? Lessons Learned from Using Nested DO Loops For a Family Card Game

Julia M. Skinner

ABSTRACT

SAS® is used to solve a wide variety of complex problems. However, it can also be used on a smaller scale, and these projects often provide ample opportunities for learning both the technical and non-technical skills needed to tackle these larger projects. This paper describes the author's use of SAS to solve a small problem born of family bickering: the wager required per person for a card game. Using nested DO loops, this paper demonstrates how to determine all possible combinations of wagers that satisfy the stakeholders and describes the process taken to arrive at this solution. Despite the small scope and light-hearted nature of the exercise, there are several lessons that can be learned about SAS coding, requirements gathering and other skills that can be applied to solving more complicated real-world problems.

INTRODUCTION

SAS is a powerful and versatile analytics tool. It is used to solve large and complex problems, such as pandemic response, fraud detection and electrical grid management[1]. However, it also can be used to solve smaller problems, and these are a good way to learn the skills necessary to tackle larger projects. Programming and data handling techniques learned in one situation can be widely applied to others. Even small projects necessitate requirements gathering and working with stakeholders. These non-technical skills are often overlooked but are critical to the success of any project.

When my family gets together at my paternal grandmother's house, we play our family version of a card game named "May I?". The beginning of the game is a familiar, beloved ritual - Grandma gets the special deck out, players chose their seats, and everyone pulls out their wallets. In our version of the game, we determine an amount to wager, and each player contributes that amount at the start of the game. This money is divided up and awarded at fourteen different game milestones. The number of players and the number of small bills and coins available varies each time, and the payouts get bigger as the game progresses. Bickering over how the money is divided up is a cornerstone of this pre-game ritual.

As I watched my grandmother divide up the money one night, I started wondering how many possible combinations of payouts were possible and thinking how I could write code to determine this. As so often happens, a problem that seemed straightforward at the start revealed itself to be more complicated, and I found myself using the same methods I have used throughout my career to determine the solution.

REQUIREMENTS

"May I?" is a form of contract rummy[2]. The game has many variations, but in my family, the game is played with three to eight players, using three decks of standard playing cards that includes six jokers. The game is played in seven rounds; in each round, the players have a different combination of cards (contracts) they must fulfill (Table 1). Players acquire the cards needed through drawing from the deck or picking up the top card from the discard pile. Players may also request to take the top card of the discard pile out of turn by saying "May I?", getting the permission of any other players that are entitled to claim it and drawing an additional card as a penalty. The object of each round is to fulfill the contract by laying the cards on the table and to be the first to eliminate all cards from your hand. Players eliminate cards by adding them to their own and/or other players' contracts, and they earn penalty points for each card left in their hand at the end of the round. The object of the game is to have the lowest number of points at the end of Round 7.

Table 1: Summary of Game Play

Round	Cards dealt	Contract
1	10	Two sets[a]
2	10	One set, one run[b]
3	10	Two runs
4	12	Three sets
5	12	Two sets, one runs
6	12	One set, two runs
7	12	Three runs

[a]Three or more cards of the same rank
[b]Four or more cards of the same suit in sequential order

In Rounds 1 through 6, there are two payouts per round – one for being the first to fulfill the contract and one for being the first to eliminate all cards from their hand. In Round 7, every card in the player's hand must be used to fulfill the contract, so the round ends when the first person fulfills the contract and only one payout is issued. The final payout is awarded to the player with the lowest number of points at the end of the game.

For my first attempt at the program, I set the parameters based on my observations of past games and refined by means of discussions with a group of stakeholders (my grandmother and immediate family):

- The payouts for Rounds 1-4 are less than the payouts for Rounds 5-6, which have more complicated contracts and require more cards to be dealt.
- The Round 7 payout is higher than the individual payouts in Rounds 5-6, and the final payout is the highest of all.
- If there are two payouts within a round, each should be the same amount.
- The smallest unit of currency used in the Round 1-6 payout is $0.25. The smallest unit of currency used in the Round 7 payout and the final payout is $1.00.
- The minimum wager per person is $1.00, and the maximum wager is $7.00. The smallest unit of currency for the wagers is $1.00.

CODE DEVELOPMENT

I used SAS© Studio in SAS OnDemand for Academics to write the code, registering as an independent learner. This project was for non-commercial use, and the cloud-based service would allow me to access and edit the code from any location, including my grandmother's home computer.

I used a DATA step with nested DO loops to output every possible combination of payouts. For each possible number of players (*n*), the program cycles through every possible wager (*w*) and determines all combinations of eight Round 1-4 payouts (*a*), four Round 5-6 payouts (*b*), one Round 7 payout (*c*) and one final payout (*d*) that satisfy the requirements (Table 2).

Table 2: DO Loop Parameters

Type	Variable	Number of payouts	Minimum payout	Minimum sum of payouts
Round 1-4	*a*	8	$0.25	$2.00
Round 5-6	*b*	4	$0.50	$2.00
Round 7	*c*	1	$1.00	$1.00
Final	*d*	1	$2.00	$2.00

Based on the rules for incrementation and the relative size of the payouts, I set the start value of *a* at $0.25, *b* at $0.50, *c* at $1.00 and *d* at $2.00. I initially set the stop values as the total amount of money wagered (*total*, calculated as *n*w*). However, this was unnecessarily time intensive; five runs of the program took a median of 11.22 seconds in real time. Instead, I coded the DO loop to iterate only when the sum of the payouts was less than or equal to the total amount of money available, minus the minimum number to fulfill all the other payouts. This gave identical results and decreased the median run time to 0.19 seconds:

```
data all_combinations;
   format n 3. w a b c d total dollar6.2;
   do n=3 to 8;
      do w=1 to 7;
         total=n*w;
         do a=0.25 to total by 0.25 while (a*8<=total-5);
            do b=0.5 to total by 0.25 while (b*4<=total-5);
               do c=1 to (total-6);
                  do d=2 to (total-5);
                     if ((8*a)+(4*b)+c+d)=total and a<b<c<d then output;
                  end;
               end;
             end;
          end;
       end;
    end;
run;
```

The resulting program produced many possible combinations (Table 3). As I examined the results, I noticed there were many combinations that fulfilled the initial set of requirements but would never be chosen in a real-life game. Table 4 shows four selected combinations for a five-player game where each player wagers $4.00. There are multiple combinations where there is a large gap between the tiers (such as $0.25 for Rounds 1-4 and $1.75 for Rounds 5-6) that is not commensurate with the increase in difficulty. There were also multiple combinations where almost all the money is used for the final two payouts (such as $5.00 for Round 7 and $11.00 for the final payout with $20.00 total). The person who wins Round 7 often wins the whole game because there is no opportunity to shed cards to minimize penalty points, so it is preferable to allocate the money more evenly across all payouts.

Table 3: Number of Possible Combinations by Number of Players and Wager Per Person

Number of players	Wager per person							Total
	$1.00	$2.00	$3.00	$4.00	$5.00	$6.00	$7.00	
3	-	-	3	10	25	50	88	176
4	-	2	10	32	74	141	239	468
5	-	5	25	74	162	302	506	1,074
6	-	10	50	141	302	556	921	1,980
7	1	19	88	239	506	921	1,517	3,291
8	2	32	141	376	786	1,420	2,327	5,084

Table 4: Selected Results for Five Players Wagering $4.00

Round 1-4	Round 5-6	Round 7	Final
$0.25	$0.50	$1.00	$15.00
$0.25	$0.50	$5.00	$11.00
$0.25	$0.50	$4.00	$6.00
$0.25	$1.75	$4.00	$5.00

To recreate these unwritten rules about which combinations to use, I tested several additional parameters to select results that would be satisfactory to most players. I chose the following:

- The payouts in Round 5-6 should be less than or equal to three times the amount of the payouts in Round 1-4.
- The payout for Round 7 should be more than the sum of two Round 5-6 payouts, but less than the sum of all four Round 5-6 payouts.
- The final payout should be less than or equal to two times the amount of the payouts in Round 7.
- The sum of the Round 7 and final payouts should be less than or equal to the sum of all Round 1-6 payouts.

Once defined, the parameters were written as a WHERE statement to obtain the combinations that complied with these rules:

```
data all_satisfactory;
   set all_combinations;
   where b<=(3*a) and (2*b)<c<(4*b) and d<=(c*2) and (c+d)<=(8*a)+(4*b);
run;
```

RESULTS

The number of satisfactory payout combinations ranged from nine for a three-person game and 244 for an eight-person game (Table 5). Using this method, the minimum wager per person is between $2.00 and $4.00, depending on the number of players.

Table 5: Number of Satisfactory Combinations by Number of Players and Wager Per Person

Number of players	Wager per person							Total
	$1.00	$2.00	$3.00	$4.00	$5.00	$6.00	$7.00	
3	-	-	-	1	1	4	3	9
4	-	-	1	2	4	7	14	28
5	-	1	1	4	8	16	23	53
6	-	1	4	7	16	29	44	101
7	-	-	3	14	23	24	69	153
8	-	2	7	17	38	70	110	244

I used the SQL procedure with an in-line view to create reference tables that I could distribute to my family. For each number of players (*n*), the code identifies the minimum wager (*w*) that fulfills all the conditions and displays all possible permutations of payouts for that dollar value. Our extended family is large, and each group of players has its own preferences, so I wrote it as a macro with the option to add additional conditions:

```
%macro pick_min(param=);
   proc sql;
      select orig.*
      from (select n, min(w) as w
            from all_satisfactory
            %if &param^= %then %do;
               where &param
            %end;
            group by n) as mins inner join all_satisfactory as orig
            on orig.n=mins.n and orig.w=mins.w
            %if &param^= %then %do;
               where &param
            %end;
            order by orig.n, orig.w, orig.a, orig.b, orig.c, orig.d;
   quit;
%mend pick_min;
%pick_min(param=)
%pick_min(param=%str(d>=5))
```

The first macro call uses no additional conditions beyond the standard parameters. The results are shown in Table 6. As expected, the minimum wager is between $2.00 and $4.00. A seven-person game offers three possible combinations of payouts for the minimum $3.00 wager.

Table 6: Minimum Wager by Number of Players

Number of players	Wager per person	Round 1-4	Round 5-6	Round 7	Final	Total wagered
3	$4.00	$0.50	$0.75	$2.00	$3.00	$12.00
4	$3.00	$0.50	$0.75	$2.00	$3.00	$12.00
5	$2.00	$0.25	$0.75	$2.00	$3.00	$21.00
6	$2.00	$0.50	$0.75	$2.00	$3.00	$12.00
7	$3.00	$0.75	$1.25	$4.00	$6.00	$21.00
			$1.50		$5.00	
		$1.00	$1.25	$3.00	$5.00	
8	$2.00	$0.50	$1.00	$3.00	$5.00	$16.00
			$1.25		$4.00	

The second macro call reflects my immediate family's preference for the final prize to contain a five-dollar bill that the winner can flaunt in their victory. Table 7 shows the minimum wager per player where the final payout is $5.00 or greater. The minimum wager is now between $2.00 and $6.00, and three-, five-, six- and seven-player games have multiple possible combinations of payouts for the minimum values.

Table 7: Minimum Wager by Number of Players Where the Final Payout is At Least $5.00

Number of players	Wager per person	Round 1-4	Round 5-6	Round 7	Final	Total wagered
3	$6.00	$0.50	$1.25	$3.00	$6.00	$18.00
				$4.00	$5.00	
		$0.75	$1.00	$3.00	$6.00	
4	$4.00	$0.50	$1.00	$3.00	$5.00	$16.00
5	$4.00	$0.50	$1.50	$4.00	$6.00	$20.00
		$0.75	$1.25	$3.00		
				$4.00	$5.00	
6	$3.00	$0.50	$1.25	$3.00	$6.00	$18.00
				$4.00	$5.00	
		$0.75	$1.00	$3.00		
7	$3.00	$0.75	$1.25	$4.00	$6.00	$21.00
			$1.50		$5.00	
		$1.00	$1.25	$3.00		
8	$2.00	$0.50	$1.00	$3.00	$5.00	$16.00

DISCUSSION

Wagering in a family card game is a low-stakes problem, but to get a satisfactory outcome, I needed to use many of the same methods that I've used when solving consequential real-world problems. In addition to the coding techniques of nested DO loops and PROC SQL in-line views, I had to conduct requirements gathering, performance benchmarking and code validation. These are all important skills that SAS users need to cultivate along with their coding skills.

No matter how small the project, it is a good idea to consider the lessons you learned from it and use them to improve your work going forward. There are several lessons that can be learned from this exercise:

- **The unwritten needs to become written:** There are often unwritten rules and institutional practices that govern how a given process works, be it a family card game or a multinational supply chain. If you are writing code to incorporate into one of these processes, those rules need to be defined and communicated to the programmer to ensure a satisfactory result. My initial requirements draft missed several of these unwritten rules about how payouts are structured and had to be revised. Furthermore, I needed the institutional knowledge I had about how our family games work, such as knowing which aspects of game play differ from the standard rules, or that payouts could be increments of $0.25 because Grandma has a jar of quarters to make change for dollar bills. Clearly defining and communicating these unwritten rules increases accuracy and saves development time.
- **Ensure the accuracy of your requirements**: I have been playing May I? with my family for over two decades, but my initial requirements write-up contained several errors that my stakeholders corrected. If your requirements or supporting documentation is incorrect, no amount of coding expertise will get you to the correct solution. When possible, confirm that the requirements have been reviewed for accuracy by a knowledgeable stakeholder, and review them yourself for any obvious errors before you start work.
- **Build in time for unexpected problems:** I did not expect to have so many possible combinations that I would have to further refine my parameters. I also had not considered that the initial draft of the program would be so resource-intensive. It took extra time for me to address these issues, and in the latter case, verify that the more efficient code was returning the same results as the original code. Every project will have at least one unanticipated problem. When determining how much time is necessary to complete a task, always allocate extra time to cover such issues.
- **Take the opportunity to try new techniques**: It is common to re-use code that has worked in the past, and this is a sensible practice. However, if time allows and the project may benefit from a technique that you have learned but never tried, consider using the new technique. The best way to

learn something is to practice it, and an investment of a small amount of time now could save large amounts in the future. I knew about in-line views in PROC SQL, but before this project, I would usually default to the PROC SQL syntax which I was already proficient. This opportunity to practice using in-line views has made it so that I will be able to use this more efficient method with ease in the future.

SAS users must continually learn to retain and sharpen their skills, and chances to do so can be found in many places. While the big projects are the things we highlight on our resumes or talk about at parties, the small projects can have enormous value as well. If SAS users continually seek out new opportunities to learn, they will be able to play any hand they are dealt.

REFERENCES

[1]SAS Institute. "SAS Industry Solutions". Accessed June 10, 2021. https://www.sas.com/en_us/industry.html

[2]Gibson, Walter B. 1974. *Hoyle's Encyclopedia of Card Games*. New York, NY: Broadway Books.

ACKNOWLEDGMENTS

The author would like to thank her grandmother Marion Gerace for inspiring this paper, and Jonathan Barcklow, Garrett Lodewyck and Wen-Wen Sun for their review and support.

CONTACT INFORMATION

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

Julia M. Skinner
julia.m.skinner@gmail.com
https://www.linkedin.com/in/juliamskinner/