

HILLSDALE COLLEGE

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

HONORS THESIS

**A Systematic Approach to
Filling m -by- n Numerical Arrays**

Author:
Gennady Stolyarov II

Supervisor:
Dr. David Murphy

December 3, 2008

Abstract

This paper develops a systematic way to fill any m -by- n numerical array where the row and column constraints are specified and the sum of the row constraints is equal to the sum of the column constraints. This problem has both relevance, due to the growing interest in numerical puzzles, and applications, especially to the field of algebraic geometry. In this paper, it is shown that any m -by- n numerical array with the given specifications can be filled, and a directional row-by-row filling algorithm is developed for doing so. However, many of even the simplest numerical arrays have multiple possible fillings, and we seek to arrive at a way of finding how many fillings any given array has. We develop a formula for the number of fillings for m -by- d arrays, where d is the sum of the row constraints and the sum of the column constraints, and each column constraint is equal to 1. Then we proceed to find formulas for the number of fillings for 2-by-2, 2-by-3, 2-by-4, 2-by- n , 3-by-3, and 3-by- n arrays. In the process, we develop the necessary techniques, insights, and notation to enable us to develop a formula for the number of fillings for a general m -by- n array.

Acknowledgments

My sincere thanks go to Dr. David C. Murphy for his extensive assistance with this project. His contributions include introducing me to the central problem discussed here, as well as reviewing my work every step of the way. Even the typesetting of this paper using \LaTeX would not have been possible without his urging that I use said approach and his generous efforts to familiarize me with it. Dr. Murphy's other contributions include his input into the structure of this paper, his suggestion that special notation be used to make the results appear more concise and easier for readers to grasp, and his work on the pseudo-code for a program that would compute the number of fillings for large arrays which, even with the formulas derived here, would be far too time-consuming to calculate by hand.

Furthermore, I am grateful to Dr. Nathan Sprague of Kalamazoo College for writing the Python source code for an array-filling program that was used to test the formulas derived in this paper on specific examples.

Thanks are also in order to Dr. Reinhardt Zeller for his consultation services to Dr. Murphy and myself regarding how a program for computing the number of array fillings might be created.

Contents

1	Introduction	4
1.1	Motivation: Numerical Puzzles	4
1.2	The Puzzles Under Consideration: m -by- n Arrays	5
1.3	Applications to Military Formations	6
1.4	Applications to the Assignment of Tasks	7
2	Row-by-Row Filling Algorithm	8
2.1	Examples of Row-Filling Algorithm's Application	8
2.2	Proof that the Row-by-Row Filling Algorithm Works	10
3	Fillings Are Not Unique	11
3.1	The Lack of Uniqueness for Fillings of 2-by- n , m -by-2, and Larger Arrays	11
3.2	The Number of Fillings for a m -by- d Array	12
4	Directional Filling Algorithms Do Not Suffice	13
5	The Legitimacy of Rearranging Array Row and Columns	14
6	The Number of Fillings for a 2-by-n Array	15
6.1	The Number of Fillings for a 2-by-2 Array	15
6.2	The Number of Fillings for a 2-by-3 Array	16
6.3	The Number of Fillings for a 2-by-4 Array	17
6.4	The Number of Fillings for a General 2-by- n Array	18
6.5	Induction Proof of the Formula for the Number of Fillings for a General 2-by- n Array	19
7	A More Concise and Convenient Notation	21
8	The Number of Fillings for a 3-by-n Array	23
8.1	The Flexibility of Cell Placements for Arrays With More Than 2 Rows and 2 Columns	23
8.2	The Number of Fillings for a 3-by-3 Array	24
8.3	The Number of Fillings for a General 3-by- n Array	26
8.4	Proof of the Formula for the Number of Fillings for a 3-by- n Array	28
9	The Number of Fillings for a m-by-n Array	29
9.1	A Formula for the Number of Fillings of a m -by- n Array	29
9.2	Proof of the Formula for the Number of Fillings of a General m -by- n Array	31
10	Questions for Further Exploration	33
10.1	The Symmetric Polynomial Approach to Filling Arrays	33
10.2	Sensitivity to Constraints and Complexity of the Problem	34
10.3	Extensions of the Array-Filling Problem	35
A	Permutations of Rows and Columns With Equal Constraints	35
A.1	Motivation for the Permutation Approach	35
A.2	Row Permutations Always Commute With Column Permutations	36
A.3	Two Row Permutations Do Not Necessarily Commute	37
A.4	Directional Filling Algorithms and Permutations Do Not Suffice to Find All Fillings	37

1 Introduction

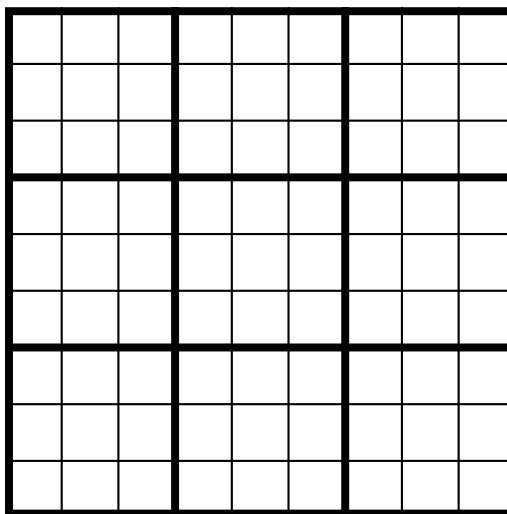
1.1 Motivation: Numerical Puzzles

The systematic study of puzzles continues to be of considerable interest to mathematicians. Numerical puzzles such as *Sudoku* and *Kakuro* have generated widespread interest among both mathematicians and the general public.

Sudoku, for instance, typically consists of a 9-by-9 grid which the solver fills with numbers - under the restrictions that

1. the numbers used must be integers 1 through 9;
2. the numbers used must not repeat within a particular row, column, or smaller 3-by-3 bold-outlined square within the larger grid;
3. each of the integers 1 through 9 must be used once within every particular row, column, or smaller 3-by-3 bold-outlined square;
4. every particular puzzle begins with some numbers already filled in, and the solver must complete the puzzle in accord with this given information. This enables a particular *Sudoku* puzzle to have a unique solution.

Here is an example of a *Sudoku* grid:



In 2006, Felgenhauer and Jarvis [3] showed that there are exactly 6,670,903,752,021,072,936,960 valid possibilities for 9-by-9 *Sudoku* grids. This leads us to ask similar questions about other numerical puzzles. How many solutions are possible for any given numerical puzzle, and is there a systematic way of counting them?

Kakuro is a numerical puzzle closer in nature to the puzzles which we will explore in this paper. An empty grid is given, with sums specified for each row, column, or part of a row or column. The solver fills with numbers - under the restrictions that

1. the numbers used must be integers 1 through 9;
2. no integer can appear twice within any series of cells for which a sum is specified.

Here is an example of a *Kakuro* puzzle, unsolved.

	23	30			27	12	16
16				24			
17			29				
35			15			12	
	7			8			
		16		7			7
	11	10					
21					5		
6					3		

Here is the solution to the *Kakuro* puzzle above.

	23	30			27	12	16
16	9	7		24	8	7	9
17	8	9	29	8	9	5	7
35	6	8	5	9	7		
	7	6	1	8	2	6	7
		16	4	6	1	3	2
	11	10					
21	8	9	3	1	5	1	4
6	3	1	2		3	2	1

Because duplication of numbers within a particular sum is prohibited, *Kakuro* puzzles often have unique solutions.

The puzzles we will explore are similar to *Kakuro* in that they also have specified sums for each row and column of a grid. The grids our puzzles of interest will have, however, will be somewhat simpler than *Kakuro* grids, and the restrictions regarding the numbers that can be placed in each cell of the grid will be considerably laxer.

1.2 The Puzzles Under Consideration: m -by- n Arrays

The numerical puzzles that this paper will explore differ somewhat from *Sudoku* in the constraints they entail; they more greatly resemble *Kakuro*, though with considerably looser restrictions. Each of these puzzles, to which we will henceforth refer as m -by- n arrays, is a particular instance of the following general form.

$a_{1,1}$	$a_{1,2}$	$a_{1,n}$	μ_1
$a_{2,1}$	$a_{2,2}$	$a_{2,n}$	μ_2
.....	
.....	
$a_{m,1}$	$a_{m,2}$	$a_{m,n}$	μ_m

ν_1	ν_2	ν_n	$d = \sum_{i=1}^m \mu_i = \sum_{j=1}^n \nu_j$
---------	---------	-------	---------	---

The cells of the array can be filled, subject to the following requirements:

1. Each cell $a_{i,j}$ must be filled with a nonnegative integer. The number 0 and repeats are allowed.
2. The sum of the entries in a particular row i must be equal to μ_i .
3. The sum of the entries in a particular column j must be equal to ν_j .
4. The sum of the row constraints is equal to the sum of the column constraints. We can call this sum d , where $d = \sum_{i=1}^m \mu_i = \sum_{j=1}^n \nu_j$.

It is possible for m -by- n arrays to have any number of rows, any number of columns, and any integers chosen as row and column constraints, provided that the above requirements are met.

Here, we will explore several questions. Is it possible to always fill such m -by- n arrays, no matter what their particular qualities? If a filling is always possible, is some particular filling of an array unique? If a unique filling does not exist for an array, how many distinct fillings exist? Can we arrive at a formula for counting the number of fillings for a general m -by- n array — a formula which can then be applied to any specific case?

We will demonstrate a row-by-row filling algorithm that can fill any m -by- n array. Then we will show that for arrays with 2 or more rows and 2 or more columns, there often exist cases where multiple distinct fillings are possible. Finally, we will arrive at a formula that can be used to find the total number of possible fillings for any m -by- n array.

1.3 Applications to Military Formations

The number of possibilities for some kinds of military formations can be discovered by solving for the number of fillings for some m -by- n arrays.

Consider a formation of spearmen who carry spears of varying lengths. In arranging the spearmen in a battle formation, their commander needs to ensure that every single spear has access to the enemy. Based on the number of spears of various lengths he has, the commander will put certain numbers of spearmen in each row so that the tips of their spears align exactly with the tips of the spears of all other available lengths. This formation also has a number of columns equal to the number of spearmen, since, in this scenario, no spearman can fight if there is a friendly spearman directly in front of him blocking his spear's access to the enemy.

In effect, the commander will be arranging the spearmen in some m -by- d array, where m is the number of distinct spear lengths available to the spearmen and d is the number of spearmen. The

army has μ_1 of the longest spears, μ_2 of the second-longest spears, and so on; the army has μ_m of the shortest spears. This is how the constraints for the arrangement of spearmen can be expressed in array form:

$a_{1,1}$	$a_{1,2}$	$a_{1,d}$	μ_1
$a_{2,1}$	$a_{2,2}$	$a_{2,d}$	μ_2
.....	
.....	
$a_{m,1}$	$a_{m,2}$	$a_{m,d}$	μ_m
1	1	1	$d = \sum_{i=1}^m \mu_i = \sum_{j=1}^d 1$

It is in the commander's interest to know how many possible ways of forming the spearmen are available to him so that he might choose the best possible formation. We will develop a solution to this problem in Section 3.2.

It is also possible to model military formation problems via arrays where the column constraints are greater than 1. Consider an army of medieval knights and squires, where each knight has a squire who carries messages and supplies to his knight. The squires are relatively ill-suited to fighting, so it is best to keep them immediately near or behind their respective knights at all times. Like the spearmen in the previous example, the knights have lances of varying lengths, and the length of a knight's lance will determine his row placement. Each column is occupied by one knight and one squire, with the knight always in front of the squire or in the same position as the squire. This is convenient for an array problem, because this means that we do not have to consider different interpretations the placement of two 1's in the same cells of a column. Every such placement can only be interpreted as a situation where the knight is the closest to the front of the formation.

The lord commanding the knights and squires wishes to know how many different ways of forming them exist so that every lance has access to the enemy. We can solve this problem by finding the number of fillings to a m -by- $d/2$ array, where $d/2$ is the number of knights, the number of squires, and the number of columns in the formation, while m is the number of rows in the formation.

Of course, the column constraints in this problem can be modified if some knights have more than one squire while other knights have no squires. If we allow knights to have any number of squires, the number of arrangements for this army becomes the number of fillings of some m -by- n array, where each column constraint is equal to the size of some knight's retinue, including the knight himself.

1.4 Applications to the Assignment of Tasks

We can also consider another application of the array-filling problem, which can be useful in military or civilian contexts. Consider a decision-maker — a military commander or a CEO — who has d people under his supervision, n tasks needing to be accomplished, and m different groups of workers that can be used in accomplishing them.

In a military setting, the worker groups can be contingents from different countries in a multinational coalition. A commander of a force might have soldiers from countries A, B, and C under his jurisdiction, and he might need to assign these soldiers to missions in cities X, Y, and Z. In a civilian corporation, the worker groups can be different construction crews capable of doing similar kinds of work, contingents of assembly-line workers, or any other types of laborers that are not required to perform a single highly specialized task and whose skills can be flexibly used in a range of occupations. Each of the n tasks requires some specific number of people to work on it. For the j th task, we can call the number of workers required ν_j . The number of workers in group i is μ_i .

In a military context, an army might have more soldiers from some countries than from others — which can account for differences among the row constraints. Furthermore, some tasks may require fewer people than others — which can account for differences among the column constraints.

The applications we have discussed above were developed during our work. However, the motivation for this problem arose from the research of Dr. Terrell L. Hodge and David C. Murphy investigating a notion of relative position for partial flags in \mathbb{C}^d .

2 Row-by-Row Filling Algorithm

We first show that it is possible to fill any m -by- n array where the sum of the row constraints is equal to the sum of the column constraints. We present an algorithm for doing so.

In beginning to fill the array, we can let $a_{1,1}$ equal μ_1 or ν_1 , whichever is smallest. Suppose $\mu_1 \leq \nu_1$. Then $a_{1,1} = \mu_1$, and the remainder of row 1 consists of zeros. If, on the other hand, $\mu_1 \geq \nu_1$, then $a_{1,1} = \nu_1$, and we can fill $a_{1,2}$ with $\min(\mu_1 - \nu_1, \nu_2)$. Continuing in this way, we can fill the first row.

After eliminating the first row of the array, we have a new $(m - 1)$ -by- n array to fill, with the sum of the row constraints $(\mu_2 + \mu_3 + \cdots + \mu_m) = (\nu_1 + \nu_2 + \cdots + \nu_n) - \mu_1$. We can rename our column constraints and proceed to eliminating row 2.

Using the procedure described above, we move through each row from left to right, filling each cell with the minimum of the revised row constraint (the μ of the row minus the sum of the values in cells already filled in the row) and the revised column constraint (the ν of the column minus the sum of the values in cells already filled in the column to which the cell belongs).

This row-by-row algorithm fills every cell $a_{i,j}$ with

$$\begin{aligned} a_{i,j} &= \min(\mu_i - a_{i,1} - a_{i,2} - \cdots - a_{i,j-1}, \nu_j - a_{1,j} - a_{2,j} - \cdots - a_{i-1,j}) \\ &= \min\left([\mu_i - \sum_{t=1}^{j-1} a_{i,t}], [\nu_j - \sum_{t=1}^{i-1} a_{t,j}]\right). \end{aligned}$$

Clearly, $a_{i,j} \geq 0$, because μ_i is cannot be less than the sum of some of the cells in row i , so $\mu_i \geq \sum_{t=1}^{j-1} a_{i,t}$. Likewise, ν_j cannot be less than the sum of some of the cells in column j , so $\nu_j \geq \sum_{t=1}^{i-1} a_{t,j}$.

2.1 Examples of Row-Filling Algorithm's Application

We now use the row-by-row filling algorithm described above to fill the following array:

			9
			9
8	6	4	

We start in the first row, first column, and place $\min(9, 8) = 8$ in cell $a_{1,1}$. This leaves 1 to be filled in row 1, so for cell $a_{1,2}$, we take $\min(1, 6) = 1$, thus exhausting the row 1 constraint. We now

have the following partial filling:

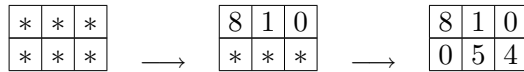
8	1	0
*	*	*

Now we fill the second row. Since the column 1 constraint has been exhausted, we place 0 in cell $a_{2,1}$. In column 2, there remains $6 - 1 = 5$ to be filled. To fill cell $a_{2,2}$, we take $\min(5, 9) = 5$, leaving 4 to be filled in both row 2 and column 3. Thus, our choice for cell $a_{2,3}$ is 4, which completely fills

the array thus:

8	1	0
0	5	4

We can summarize the filling procedure via the following diagram:



We now use the row-by-row filling algorithm described above to fill a larger array for which finding a filling by guesswork is more difficult:

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

We start in the first row, first column, and place $\min(8, 9) = 8$ in cell $a_{1,1}$. This exhausts the constraint of the first row, and we can put zeros in the other cells of that row. Now we proceed to the second row, noting that our choice for cell $a_{1,1}$ is $\min(9 - 8, 8) = 1$, so we place 1 in cell $a_{2,1}$, and then place $\min(8, 8 - 1) = 7$ in cell $a_{2,2}$. This exhausts the constraint of the second row, and we can put zeros in the other cells of that row. We continue this procedure for all nine rows and get the following filling.

8	0	0	0	0	0	0	0	0
1	7	0	0	0	0	0	0	0
0	1	7	0	0	0	0	0	0
0	0	0	4	0	0	0	0	0
0	0	0	2	2	0	0	0	0
0	0	0	0	3	1	0	0	0
0	0	0	0	0	3	0	0	0
0	0	0	0	0	0	3	0	0
0	0	0	0	0	0	0	2	1

2.2 Proof that the Row-by-Row Filling Algorithm Works

Here we prove that the row-by-row filling algorithm described above can be used to fill every m -by- n array. We will use induction on the number of rows.

Part 1:

For the case of $m = 1$, to fill square $a_{1,1}$, we use $a_{1,1} = \min(\mu_1, \nu_1) = \nu_1$, since we know that $\mu_1 = \nu_1 + \nu_2 + \dots + \nu_n$ and thus μ_1 cannot be less than ν_1 .

This leaves $\mu_1 - \nu_1$ to be filled in row 1, and also removes ν_1 from the remaining sum needing to be accounted for in the columns (a sum which was originally $\nu_1 + \nu_2 + \dots + \nu_n$).

Indeed, we are forced to fill every square $a_{1,j}$ with ν_j , because this is the only way to satisfy each of the column constraints. This filling is in accordance with the row-by-row filling algorithm, because ν_j is guaranteed to be no greater than $\mu_1 - \nu_1 - \dots - \nu_j - 1 = \nu_j + \nu_{j+1} + \dots + \nu_n$.

This is the result of filling a 1-by- n array using the row-by-row filling algorithm.

ν_1	ν_2	ν_n	μ_1
ν_1	ν_2	ν_n	$d = \sum_{i=1}^m \mu_i = \sum_{j=1}^n \nu_j$

Part 2: Now, we assume that the algorithm works for $m = k$. That is, it is possible to use this algorithm to fill a k -by- n array. We show that it works for $m = k + 1$ — that is, it is possible to use this algorithm to fill a $(k + 1)$ -by- n array.

We begin by filling the first row of the $(k + 1)$ -by- n array. To fill square $a_{1,1}$, we use $a_{1,1} = \min(\mu_1, \nu_1)$. This leaves $\mu_1 - a_{1,1}$ to be filled in row 1, and also removes $a_{1,1}$ from the remaining sum needing to be accounted for in the columns (a sum which was originally $\nu_1 + \nu_2 + \dots + \nu_n$). We fill $a_{1,2}$ with $a_{1,2} = \min(\mu_1 - a_{1,1}, \nu_2)$, which leaves us with $\mu_1 - a_{1,1} - a_{1,2}$ to be filled in row 1, and removes $a_{1,2}$ from the remaining sum needing to be accounted for in the columns.

Continuing in this manner, for some t we fill $a_{1,t}$ with $a_{1,t} = \min(\mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1}, \nu_t)$, which leaves us with $\mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1} - a_{1,t}$, to be filled in row 1, and removes $a_{1,t}$ from the remaining sum needing to be accounted for in the columns. This remaining sum is now

$$\nu_1 + \nu_2 + \dots + \nu_n - (a_{1,1} + a_{1,2} + \dots + a_{1,t-1} + a_{1,t}).$$

Finally, we fill $a_{1,n}$ with $a_{1,n} = \min(\mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1} - a_{1,t} - \dots - a_{1,n-1}, \nu_n)$, which leaves us with $\mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1} - a_{1,t} - \dots - a_{1,n-1} - a_{1,n}$ to be filled in row 1, and removes $a_{1,n}$ from the remaining sum needing to be accounted for in the columns. This remaining sum is now

$$\nu_1 + \nu_2 + \dots + \nu_n - (a_{1,1} + a_{1,2} + \dots + a_{1,t-1} + a_{1,t} + \dots + a_{1,n-1} + a_{1,n}).$$

If for some $t \leq n$, $a_{1,t} = \mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1}$, then we are guaranteed that the sum of the entries in row 1 is μ_1 , since $(a_{1,1} + a_{1,2} + \dots + a_{1,t-1}) + (\mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1}) = \mu_1$. Furthermore, all entries in row 1 following such an $a_{1,t}$ will be zero since the row constraint will have been exhausted by all the entries up to and including $a_{1,t}$.

The only potential problem arises if for $t = n$, it is the case that $\nu_n < \mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,n-1}$, in which case choosing ν_n as $a_{1,n}$ would not suffice to fulfill the row 1 constraint μ_1 . But the problematic scenario described above is impossible, because if $a_{1,n}$ is nonzero, this means that all prior entries $a_{1,j}$ were equal to ν_j . Hence, having $\nu_n < \mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,n-1}$ would lead to $(\nu_1 + \nu_2 + \dots + \nu_n) < \mu_1$, which violates the initial stipulation that the sum of the row constraints equals the sum of the column constraints. If sum of all the column constraints is less than μ_1 , then this initial stipulation does not hold, because then the sum of all the column constraints will certainly be less than the sum of *all* the row constraints. If the initial stipulation holds, then the problem discussed here cannot arise.

Thus, we are guaranteed to have $\nu_n \geq \mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1} - a_{1,t} - \dots - a_{1,n-1}$ for all t , in which case $a_{1,n} = \mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1} - a_{1,t} - \dots - a_{1,n-1}$ and the sum of the entries in row 1 is μ_1 , since $(a_{1,1} + a_{1,2} + \dots + a_{1,n-1}) + \mu_1 - a_{1,1} - a_{1,2} - \dots - a_{1,t-1} - a_{1,t} - \dots - a_{1,n-1} = \mu_1$. Furthermore, the remaining sum needing to be accounted for in the columns is exactly

$$(\nu_1 + \nu_2 + \dots + \nu_n) - (a_{1,1} + a_{1,2} + \dots + a_{1,n-1}) = (\nu_1 + \nu_2 + \dots + \nu_n) - \mu_1 = (\mu_1 + \mu_2 + \dots + \mu_m) - \mu_1 = (\mu_2 + \dots + \mu_m)$$

Thus, after filling the first row of the array, we are left with a k -by- n array for which the sum of the row constraints is equal to the sum of the column constraints.

In this revised k -by- n array, the row constraints are $\mu_2, \mu_3, \dots, \mu_k, \mu_k + 1$ and the column constraints are $\nu_1 - a_{1,1}, \nu_2 - a_{1,2}, \dots, \nu_n - a_{1,n}$. By our assumption, any k -by- n array can be filled, so this array can also be filled. Hence, by filling the first row of a $k + 1$ -by- n array and then filling the resulting k -by- n array, we have filled the entire $k + 1$ -by- n array. Thus, we have shown that if the row-by-row filling algorithm works for $m = k$, then it also works $m = k + 1$. **Q. E. D.**

3 Fillings Are Not Unique

3.1 The Lack of Uniqueness for Fillings of 2-by- n , m -by-2, and Larger Arrays

Naturally, a 1-by- n or m -by-1 array will have a unique filling, because a single row or column constraint must be exhausted by each entry in each cell, and there is only one way of doing so. However, there can exist multiple ways of filling an array where there are at least two rows and at least two columns.

We consider the array

		1
		1
1	1	

Here are two possible fillings (and these are the only possible fillings) of the array above without

row and column sum constraints listed:

1	0
0	1

 is one filling of the array above while

0	1
1	0

 is the other.

In general, for an n -by- n array with $\mu = (1, 1, \dots, 1)$ and $\nu = (1, 1, \dots, 1)$ for their row and column-sum constraints, there are $n!$ possible fillings. (The proof is straightforward: There are n choices for the 1 in the first row, leaving $n - 1$ choices for the placement of the 1 in the second row, $n - 2$ choices in the third row, and so on, so that the number of possible fillings is $n(n - 1)(n - 2) \cdots 1 = n!$.)

Because there is no unique filling to even such simple arrays, we address the question of *how many* fillings there are for a given m -by- n array, where both m and n are greater than 1.

3.2 The Number of Fillings for a m -by- d Array

To explore a simple yet non-trivial version of the problem we wish to solve, we consider the number of fillings for a m -by- d array, which has m rows, with row constraints $\mu_1, \mu_2, \dots, \mu_m$, and d columns, with 1 as the constraint for each column. Every m -by- d array will be of the following general form:

$$\begin{array}{cccc|c}
 a_{1,1} & a_{1,2} & \dots & a_{1,d} & \mu_1 \\
 a_{2,1} & a_{2,2} & \dots & a_{2,d} & \mu_2 \\
 \vdots & \vdots & \dots & \vdots & \vdots \\
 a_{m,1} & a_{m,2} & \dots & a_{m,d} & \mu_m \\
 \hline
 1 & 1 & \dots & 1 & d = \sum_{i=1}^m \mu_i = \sum_{j=1}^d 1
 \end{array}$$

We note that every single cell in such a m -by- d array will have an entry of either a 0 or 1, since no column constraint permits a larger entry in any cell. Furthermore, there must be μ_1 1's in the first row, μ_2 1's in the second row, and so forth, with μ_m 1's in the m th row.

In how many ways can we place μ_1 1's in the first row? Out of d possible cells, we have to fill some μ_1 of those cells with 1's, and assign 0's to the rest of the cells. This is a combination: $C(d, \mu_1) = \frac{d!}{(\mu_1)!(d-\mu_1)!}$.

Now the μ_1 columns in which we put 1's in the first row can no longer have 1's in them. Thus, we have $d - \mu_1$ columns left, and μ_2 of those columns must have 1's in the second row. The number of ways to pick μ_2 1's in the second row is thus $C(d - \mu_1, \mu_2)$.

Likewise, for the third row, the number of ways to pick μ_3 1's after μ_1 columns have been filled in the first row and μ_2 columns have been filled in the second row is $C(d - \mu_1 - \mu_2, \mu_3)$.

The same pattern holds for filling the remaining μ_m columns with 1's in row m . The number of such selections that can be made is $C(d - \mu_1 - \mu_2 - \dots - \mu_{m-1}, \mu_m)$.

To find the total number of ways to fill this m -by- d array, we multiply together the number of ways to fill each row. We get the following result.

Proposition 1. *There are*

$$C(d, \mu_1)C(d - \mu_1, \mu_2)C(d - \mu_1 - \mu_2, \mu_3) \dots C(d - \mu_1 - \mu_2 - \dots - \mu_{m-1}, \mu_m) = \frac{d!}{(\mu_1)!(\mu_2)! \dots (\mu_m)!}$$

ways to fill a m -by- d array, where d is the sum of the row constraints and the sum of the column constraints.

When the column constraints are allowed to be greater than 1, the problem becomes substantially more complicated. In the remainder of this paper, we will develop the conceptual and procedural framework necessary to resolving such a problem.

4 Directional Filling Algorithms Do Not Suffice

At first, it might be conjectured that the problem of discovering the number of fillings for a given array is a matter of using multiple directional filling algorithms on the same array. The row-by-row filling algorithm presented previously went from the left of each row to the right of that row. We can term it the (left)^m filling algorithm, because we go from left to right in m rows. But other filling algorithms are possible, such as left-right-left for a 3-by-n array, where the first and third rows are filled from left to right, and the second row is filled from right to left.

Different directional filling algorithms may produce different fillings of an array, but they are not sufficient to produce all the fillings of some arrays. As an example of a case where the directional filling algorithms are insufficient, we consider the following array.

				1
				1
				1
				1
	1	1	1	1

0	0	0	1
0	1	0	0
0	0	1	0
1	0	0	0

The filling

0	0	0	1
0	1	0	0
0	0	1	0
1	0	0	0

 cannot be produced by any directional filling algorithm. It is possible to fill the first row

0	0	0	1
---	---	---	---

 by going from right to left, but thereafter, the second row filling cannot be produced no matter whether one chooses to go from right to left or from left to right. If

one goes from right to left, one will get

0	0	0	1
0	0	1	0

 for the first two rows of the array. If one goes

from left to right, one will get

0	0	0	1
1	0	0	0

. Neither of these possibilities corresponds to the first two rows of the filling under consideration. Thus, directional filling algorithms alone cannot produce all the possible fillings of some arrays, and a different approach toward enumerating such fillings is necessary.

The finding that directional filling algorithms alone do not suffice to find every filling of some numerical arrays raises the question of whether directional filling algorithms combined with permutations of identical row and column constraints would suffice to fill every numerical array. We investigated this question and found that even directional filling algorithms combined with such permutations do not suffice to find all the fillings of some arrays. Our approach and findings are explained in Appendix A.

However, while permutations cannot be used to find all the fillings of some arrays, they are nonetheless useful for rearranging every numerical array into a standard form that is easy to work

with. In this form, the row and column constraints are arranged in descending order. We will discuss the legitimacy of doing so next.

5 The Legitimacy of Rearranging Array Row and Columns

Suppose that we have an m -by- n array with non-negative integer entries $a_{i,j}$ in row i and column j (see figure below). We let

$$\mu_i = \sum_{j=1}^n a_{i,j} \text{ and } \nu_j = \sum_{i=1}^m a_{i,j}$$

for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$. We observe that $\sum_i \mu_i = \sum_j \nu_j$.

$a_{1,1}$	$a_{1,2}$...	$a_{1,n}$	μ_1
$a_{2,1}$	$a_{2,2}$...	$a_{2,n}$	μ_2
...	
$a_{m,1}$	$a_{m,2}$...	$a_{m,n}$	μ_m
ν_1	ν_2	...	ν_n	$d = \sum_{i=1}^m \mu_i = \sum_{j=1}^n \nu_j$

Now suppose we exchange columns 1 and 2 of this array. Will the new array be a valid filling for the row-sum vector $\mu = (\mu_1, \mu_2, \dots, \mu_m)$ and the modified column-sum vector $\nu' = (\nu_2, \nu_1, \nu_3, \dots, \nu_n)$? We demonstrate that it will indeed be a valid filling and thus that it is legitimate to rearrange any given array's rows and columns such that the row and column constraints are in descending order.

If we exchange columns 1 and 2 of the array, then the column sums ν_1 and ν_2 will still remain the same but in different positions in the array. We consider the effect on each of the row sums $(\mu_1, \mu_2, \dots, \mu_m)$: Previously, adding the values in the first row from left to right, we had

$$\mu_1 = a_{1,1} + a_{1,2} + \dots + a_{1,n}$$

Now, adding the values in the first row from left to right, we have

$$\mu_1 = a_{1,2} + a_{1,1} + \dots + a_{1,n}$$

We see that the value of μ_1 has not changed; only the order in which we add the terms of this sum has changed. The same applies to any μ_c since

$$a_{c,1} + a_{c,2} + \dots + a_{c,n} = a_{c,2} + a_{c,1} + \dots + a_{c,n}.$$

Thus, the row-sum vector is not affected by shifting the order of the columns, and the modified column-sum vector is produced by the column shift. Hence, the new array is a valid filling for the row-sum vector and the modified column sum vector.

We can demonstrate in a manner essentially identical to the discussion above that switching two rows does not alter the sum of the column constraints of an array. Instead of considering how switching some two columns affects any μ_c , we need only consider how switching some two rows affects any ν_d . Thus, we arrive at the following proposition.

Proposition 2. *Permuting the rows and columns of an array in any fashion will not fundamentally change the nature of the array-filling problem. The number of fillings in the array thus permuted will equal the number of fillings in the original array.*

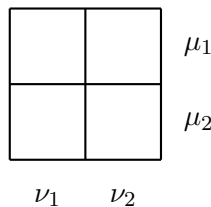
The usefulness of this result is as follows. We will be able to freely rearrange the rows and columns of any array so that the row and column constraints are in descending order. This will later be used when we count the number of fillings of any particular array. Hereafter, we will assume that our row and column constraints are arranged in descending order.

6 The Number of Fillings for a 2-by- n Array

6.1 The Number of Fillings for a 2-by-2 Array

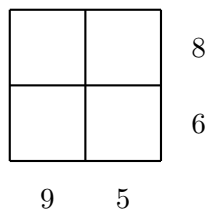
We already noted that any array with only one row or one column will have a unique filling. For a 2-by-2 array, however, this is not necessarily the case.

We consider the array



By our assumption, the row and column constraints are arranged in descending order, so that $\mu_1 \geq \mu_2$ and $\nu_1 \geq \nu_2$. This is convenient as it enables us to place any of a range of values in cell $a_{2,2}$ without encountering the possibility that this placement would not be feasible. If the placement in cell $a_{2,2}$ were not limited only by the smallest row and column constraints, then subsequent difficulties in filling the array might arise. A subsequent smaller constraint might show that our initial choice would lead to an impossibility in determining the values in other cells.

Consider, for instance what would happen if we tried to fill the following array by placing a 2 in cell $a_{1,1}$.



Then the value in cell $a_{1,2}$ would need to be a 6 to satisfy the first row constraint, but this would violate the second column constraint, which restricts the sum of entries in the second column to 5. Thus, there is no way to fill the array above by placing a 2 in cell $a_{1,1}$, even though the first row constraint and the first column constraint are not violated by such a placement.

By having the smallest constraint always in the bottom-right corner of the array, we are assured that no other constraint in the array binds our choice of the value in that cell.

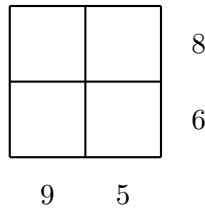
We notice that the placement of a value in $a_{2,2}$ will determine the entirety of the array. So placing some value k in cell $a_{2,2}$ will mandate us to put $\nu_2 - k$ in $a_{1,2}$, $\mu_2 - k$ in $a_{2,1}$, and $\mu_1 - \nu_2 + k$ in $a_{1,1}$.

Thus, we can choose any number to go into $a_{2,2}$, provided that it satisfies both the row constraint μ_2 and the column constraint ν_2 . That is, we can choose any integer from 0 to $\min(\mu_2, \nu_2)$ as the value in $a_{2,2}$. Each distinct choice of the value in $a_{2,2}$ will result in a distinct filling of the array. Moreover, no other fillings of the array are possible, because choosing any value other than an integer from 0 to $\min(\mu_2, \nu_2)$ as the value in $a_{2,2}$ would violate either the constraint μ_2 or the constraint ν_2 . Between 0 and $\min(\mu_2, \nu_2)$ inclusive, there are $\min(\mu_2, \nu_2) + 1$ possible integer values. Thus, we have shown the following.

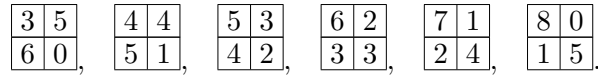
Proposition 3. *There are $\min(\mu_2, \nu_2) + 1$ fillings for a 2-by-2 array.*

Example Illustrating the Number of Fillings for a 2-by-2 Array

We consider the array

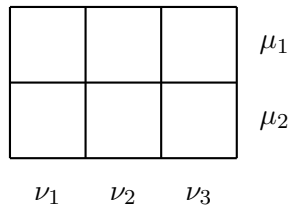


By our formula above, this array has $\min(\mu_2, \nu_2) + 1 = \min(6, 5) + 1 = 6$ fillings. By placing values ranging from 0 to 5 in cell $a_{2,2}$, we can generate all six of these fillings:



6.2 The Number of Fillings for a 2-by-3 Array

We consider a 2-by-3 array, with row and column constraints in descending order:



We notice that, after we fill the third column of this array, what remains will be a 2-by-2 array with modified row constraints — an array that can then be filled by a choice of one value in one of the cells.

We can fill cell $a_{2,3}$ of the array with any value $k_{2,3}$ from 0 to $\min(\mu_2, \nu_3)$. Each of these values $k_{2,3}$ will lead to new row constraints $\mu_2 - k_{2,3}$ and $\mu_1 - \nu_3 + k_{2,3}$ of the 2-by-2 subarray, which will affect $\min(\nu_2, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3})$. Depending on our choice of $k_{2,3}$, either the modified first row constraint $\mu_1 - \nu_3 + k_{2,3}$ or the modified second row constraint $\mu_2 - k_{2,3}$ will be the smallest row constraint of the 2-by-2 subarray. If $\mu_1 - \nu_3 + k_{2,3} < \mu_2 - k_{2,3}$, then we will place our next value, $k_{1,2} = \min(\nu_2, \mu_1 - \nu_3 + k_{2,3})$, in cell $a_{1,2}$, from which the remainder of the array filling will follow automatically. If $\mu_2 - k_{2,3} < \mu_1 - \nu_3 + k_{2,3}$, then we will place our next value, $k_{2,2} = \min(\nu_2, \mu_2 - k_{2,3})$, in cell $a_{2,2}$, from which the remainder of the array filling will follow automatically.

Thus, once $k_{2,3}$ is placed, the placement of a value in the space affected by the new minimum constraint — a value that can range from 0 to $\min(\nu_2, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3})$ — will determine

the entire array. So it is enough to select $k_{2,3}$ and then to select a value ranging from 0 to $\min(\nu_2, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3})$ in order to fill the array. Thus

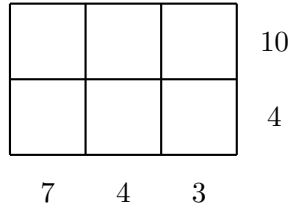
Proposition 4. *The expression*

$$\sum_{k_{2,3}=0}^{\min(\mu_2, \nu_3)} (\min(\nu_2, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3}) + 1)$$

gives the number of fillings for a 2-by-3 array.

Example Illustrating the Number of Fillings for a 2-by-3 Array

We consider the following 2-by-3 array:



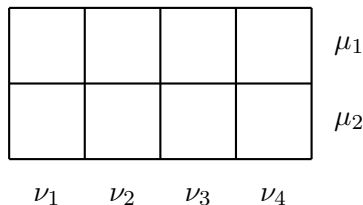
By (Proposition 4), this array has

$$\begin{aligned} & \sum_{k_{2,3}=0}^{\min(\mu_2, \nu_3)} (\min(\nu_2, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3}) + 1) \\ &= \sum_{k_{2,3}=0}^{\min(4, 3)} (\min(4, 4 - k_{2,3}, 10 - 3 + k_{2,3}) + 1) = \sum_{k_{2,3}=0}^3 (\min(4, 4 - k_{2,3}, 7 + k_{2,3}) + 1) = \sum_{k_{2,3}=0}^3 (4 - k_{2,3}) + 1 \\ &= [(4 - 0) + 1] + [(4 - 1) + 1] + [(4 - 2) + 1] + [(4 - 3) + 1] = 5 + 4 + 3 + 2 = 14 \end{aligned}$$

fillings.

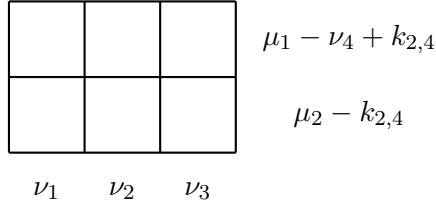
6.3 The Number of Fillings for a 2-by-4 Array

We consider a 2-by-4 array, with row and column constraints in descending order:



First, we find $k_{2,4}$, the value in cell $a_{2,4}$, which can be any of $\min(\mu_2, \nu_4) + 1$ values, ranging from 0 to $\min(\mu_2, \nu_4)$. As before, the choice of $k_{2,4}$ will also determine the contents of cell $a_{1,4}$, which will have value $\nu_4 - k_{2,4}$.

As a result of our choice of $k_{2,4}$, we are left to fill a 2-by-3 array with row constraints $(\mu_1 - \nu_4 + k_{2,4}, \mu_2 - k_{2,4})$ and column constraints (ν_1, ν_2, ν_3) :



By our formula for the 2-by-3 case (Proposition 4), there are

$$\min(\mu_2 - k_{2,4}, \mu_1 - \nu_4 + k_{2,4}, \nu_3) \sum_{k_{2,3}=0} (\min(\nu_2, \mu_2 - k_{2,3} - k_{2,4}, \mu_1 - \nu_4 - \nu_3 + k_{2,3} + k_{2,4}) + 1)$$

fillings of the remaining 2-by-3 array, where we select the value $k_{2,3}$ in cell $a_{2,3}$. If $\mu_2 - k_{2,4} < \mu_1 - \nu_4 + k_{2,4}$, then $a_{2,3}$ will be the cell that will constrain the filling of the third column, and this approach is appropriate.

If $\mu_1 - \nu_4 + k_{2,4} < \mu_2 - k_{2,4}$, then $a_{1,3}$ will be the constraining cell and not $a_{2,3}$, but it will still be appropriate to place our next subscripted k in cell $a_{2,3}$ if $\nu_3 < \mu_1 - \nu_4 + k_{2,4}$, since the value of the column constraint will then be the value limiting our number of choices.

Furthermore, if the column constraints are listed in descending order, then $\nu_4 \leq d/4$, where d is the sum of all the column constraints and the sum of all the row constraints. Also, $\nu_3 \leq d/3$, and $\nu_2 \leq d/2$ (otherwise the descending order would not hold).

Moreover, if the row constraints are listed in descending order, then $\mu_2 \leq d/2$ and $\mu_1 \geq d/2$, so $\mu_1 - \nu_4 \geq d/2 - d/4$, and so $\mu_1 - \nu_4 \geq d/4$.

If ν_4 is as large as possible, namely, $d/4$, then all other column constraints must also be $d/4$, including ν_3 . Thus, in the case where $\mu_1 = d/2$ (i.e., the first row constraint is as small as possible) and $\nu_4 = d/4$, it follows that $\nu_3 = \mu_1 - \nu_4$. In any other case, $\nu_3 < \mu_1 - \nu_4$. Thus, $\nu_3 \leq \mu_1 - \nu_4 + k_{2,4}$ always, and $\mu_1 - \nu_4 + k_{2,4}$ is never the binding constraint. So it is always the case that our second subscripted k in filling a 2-by-4 array will be $k_{2,3}$ in cell $a_{2,3}$.

We sum the number of fillings of the resulting 2-by-3 array over the number of fillings of the 4th column. This gives the number of fillings of an array in the 2-by-4 case:

$$\sum_{k_{2,4}=0}^{\min(\mu_2, \nu_4)} \sum_{k_{2,3}=0}^{\min(\mu_2 - k_{2,4}, \mu_1 - \nu_4 + k_{2,4}, \nu_3)} (\min(\nu_2, \mu_2 - k_{2,3} - k_{2,4}, \mu_1 - \nu_4 - \nu_3 + k_{2,3} + k_{2,4}) + 1)$$

Since we have ascertained that $\nu_3 \leq \mu_1 - \nu_4 + k_{2,4}$ always, we can simplify this expression.

Proposition 5. *There are*

$$\sum_{k_{2,4}=0}^{\min(\mu_2, \nu_4)} \sum_{k_{2,3}=0}^{\min(\mu_2 - k_{2,4}, \nu_3)} (\min(\nu_2, \mu_2 - k_{2,3} - k_{2,4}, \mu_1 - \nu_4 - \nu_3 + k_{2,3} + k_{2,4}) + 1)$$

fillings for a 2-by-4 array.

6.4 The Number of Fillings for a General 2-by- n Array

Now we examine the general 2-by- n case:

For a 2-by-3 array, the formula holds, as there are

$$\sum_{k_{2,3}=0}^{\min(\mu_2, \nu_3)} (\min(\nu_2, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3}) + 1)$$

fillings of the array. Indeed, Proposition 4 is simply a specific case of Proposition 6, where $n = 3$.

Also, the formula holds for a 2-by-4 array, as there are

$$\sum_{k_{2,4}=0}^{\min(\mu_2, \nu_4)} \sum_{k_{2,3}=0}^{\min(\mu_2 - k_{2,4}, \nu_3)} (\min(\nu_2, \mu_2 - k_{2,3} - k_{2,4}, \mu_1 - \nu_4 - \nu_3 + k_{2,3} + k_{2,4}) + 1)$$

fillings for the 2-by-4 case. Indeed, Proposition 5 is simply a specific case of Proposition 6, where $n = 4$.

Now we show that if for $n = q$, the formula holds, then the formula also holds for $n = q + 1$, where $q \geq 4$.

Assume that the formula holds for a 2-by- q array. Thus, by Proposition 6, the 2-by- q array has

$$\sum_{k_{2,q}=0}^{\min(\mu_2, \nu_q)} \sum_{k_{2,q-1}=0}^{\min(\nu_{q-1}, \mu_2 - k_{2,q})} \sum_{k_{2,q-2}=0}^{\min(\nu_{q-2}, \mu_2 - k_{2,q} - k_{2,q-1})} \cdots \sum_{k_{2,3}=0}^{\min(\nu_3, \mu_2 - k_{2,q} - k_{2,q-1} - \cdots - k_{2,4}, \nu_3)}$$

$$\min(\nu_2, \mu_2 - \sum_{i=3}^q k_{2,i}, \mu_1 - \sum_{j=3}^q \nu_j + \sum_{i=3}^q k_{2,i}) + 1$$

fillings.

Now we try to fill a 2-by- $(q + 1)$ array. We make a placement of value $k_{2,q+1}$ in cell $a_{2,q+1}$, which gives us the outermost summation

$$\sum_{k_{2,q+1}=0}^{\min(\mu_2, \nu_{q+1})}$$

and leaves a 2-by- q array to be filled, with row constraints $\mu_2 - k_{2,q+1}$ and $\mu_1 - \nu_{q+1} + k_{2,q+1}$. Since $q \geq 4$, we know that $q + 1 \geq 5$ and thus $\nu_q \leq \mu_2 - \nu_{q+1} + k_{2,q+1}$ as shown in Subsection 6.3. So we can place $k_{2,q}$ in cell $a_{2,q}$, and the only constraints on the value of $k_{2,q}$ to consider are $\mu_2 - k_{2,q+1}$ and ν_q .

So the number of fillings of the remaining 2-by- q array will (by our assumption that the formula works for the 2-by- q case) be

$$\sum_{k_{2,q}=0}^{\min(\nu_q, \mu_2 - k_{2,q+1})} \sum_{k_{2,q-1}=0}^{\min(\nu_{q-1}, \mu_2 - k_{2,q+1} - k_{2,q})} \sum_{k_{2,q-2}=0}^{\min(\nu_{q-2}, \mu_2 - k_{2,q+1} - k_{2,q} - k_{2,q-1})} \cdots \sum_{k_{2,3}=0}^{\min(\nu_3, \mu_2 - k_{2,q+1} - k_{2,q} - k_{2,q-1} - \cdots - k_{2,4}, \nu_3)}$$

$$\min(\nu_2, \mu_2 - \sum_{i=3}^{q+1} k_{2,i}, \mu_1 - \sum_{j=3}^{q+1} \nu_j + \sum_{i=3}^{q+1} k_{2,i}) + 1$$

We combine this sum with the outermost summation for the $(q + 1)$ st column in order to get the number of fillings for a 2-by- $(q + 1)$ array.

$$\begin{aligned}
& \min(\nu_{q+1}, \mu_2) \min(\nu_q, \mu_2 - k_{2,q+1}) \min(\nu_{q-1}, \mu_2 - k_{2,q+1} - k_{2,q}) \min(\nu_{q-2}, \mu_2 - k_{2,q+1} - k_{2,q} - k_{2,q-1}) \\
& \sum_{k_{2,q+1}=0} \quad \sum_{k_{2,q}=0} \quad \sum_{k_{2,q-1}=0} \quad \sum_{k_{2,q-2}=0} \quad \dots \\
& \min(\nu_3, \mu_2 - k_{2,q+1} - k_{2,q} - k_{2,q-1} \dots - k_{2,4}, \nu_3) \\
& \sum_{k_{2,3}=0} \quad \min(\nu_2, \mu_2 - \sum_{i=3}^{q+1} k_{2,i}, \mu_1 - \sum_{j=3}^{q+1} \nu_j + \sum_{i=3}^{q+1} k_{2,i}) + 1
\end{aligned}$$

This is the formula of Proposition 6, for $n = q + 1$. Thus, if the formula holds for a 2-by- q array, it also holds for a 2-by- $(q + 1)$ array. **Q. E. D.**

7 A More Concise and Convenient Notation

The formula of Proposition 6 contains numerous summations, each of which has lengthy superscripts. Formulas for the number of fillings of larger arrays will have even more summations and even longer superscripts, to the point of becoming visually unwieldy.

To possibly shorten otherwise cumbersome formulas, especially as we are about to embark on the 3-by- n case and then the general m -by- n case, it is clearly to our advantage to introduce short-hand notation for some of the summations involved above.

We will use

$$\triangleright^{\ell;a,b} \text{ to represent } \sum_{t=a}^b k_{\ell,t}.$$

The ℓ in this notation represents the row in which values are being added. Within this row, the values $k_{\ell,a}$ through $k_{\ell,b}$ are being added. For example, $\triangleright^{\ell;2,5} = k_{\ell,2} + k_{\ell,3} + k_{\ell,4} + k_{\ell,5}$.

With this notation, the summation

$$\begin{aligned}
& \min(\nu_3 - k_{3,3}, \mu_2 - k_{2,q+1} - k_{2,q} - k_{2,q-1} \dots - k_{2,4}) \\
& \sum_{k_{2,3}=0} \quad \dots
\end{aligned}$$

would be written

$$\begin{aligned}
& \min(\nu_3 - k_{3,3}, \mu_2 - \triangleright^{2;4,q+1}) \\
& \sum_{k_{2,3}=0} \quad \dots
\end{aligned}$$

The new notation makes it possible to express our formula for the number of fillings of a 2-by- n array much more concisely.

Proposition 7. *The number of fillings for a 2-by- n array where $n \geq 3$ is*

$$\begin{aligned}
& \min(\mu_2, \nu_n) \min(\nu_{n-1}, \mu_2 - k_{2,n}) \min(\nu_{n-2}, \mu_2 - \triangleright^{2;n-1,n}) \quad \dots \quad \min(\nu_3, \mu_2 - \triangleright^{2;4,n}, \nu_3) \\
& \sum_{k_{2,n}=0} \quad \sum_{k_{2,n-1}=0} \quad \sum_{k_{2,n-2}=0} \quad \dots \quad \sum_{k_{2,3}=0} \\
& \min(\nu_2, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n \nu_j + \triangleright^{2;3,n}) + 1
\end{aligned}$$

The \triangleright notation enables us to concisely summarize subtraction of many subscripted k 's from the same row. As we will see later, we also need a way to concisely summarize subtraction of many subscripted k 's from the same column. We will henceforth use

$$\nabla^{\ell;a,b} \text{ to represent } \sum_{t=a}^b k_{t,\ell}.$$

For instance, $k_{2,7} + k_{3,7} + k_{4,7} + k_{5,7} + k_{6,7} = \nabla^{7;2,6}$.

We will see in Section 9.1 that to fill an m -by- n array, it is necessary to use $mn - n - m$ summations. To express $mn - n - m$ summations, each of the form

$$\min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n}) \sum_{k_{q,j}=0}$$

is inconvenient to say the least. To render such expressions concise and manageable, we will introduce an additional shorthand device. In general, we will let

$$\bigsqcup_{q:t;s} \sum_{k_{q,j}=0}^f$$

represent the composition of all the summations where q is any value from s to t , and j is any value from u to v . The summations are applied so that the outermost summation is the one where $q = t$ and $j = v$, where t and v are the *highest* row and column numbers within the given range. Within each summation in the composition above, the value of $k_{q,j}$ can range from 0 to some value f .

For example,

$$\bigsqcup_{q:9;8}^{j:6;2} \sum_{k_{q,j}=0}^f = \sum_{k_{9,6}=0}^f \sum_{k_{9,5}=0}^f \sum_{k_{9,4}=0}^f \sum_{k_{9,3}=0}^f \sum_{k_{9,2}=0}^f \sum_{k_{8,6}=0}^f \sum_{k_{8,5}=0}^f \sum_{k_{8,4}=0}^f \sum_{k_{8,3}=0}^f \sum_{k_{8,2}=0}^f$$

For summations of the sort with which we will be working, we will use notation of the form

$$\bigsqcup_{q:t;s} \sum_{k_{q,j}=0}^f \min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n})$$

to represent a composition of multiple summations. For example,

$$\bigsqcup_{q:5;3}^{j:6;4} \sum_{k_{q,j}=0}^f \min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n})$$

is the composition of all the summations for picking the values of cells in the rectangle whose corner cells are $a_{3,4}$, $a_{3,6}$, $a_{5,4}$, and $a_{5,6}$. The outermost summation will be the one in which $(q, j) = (5, 6)$, and the innermost summation will be the one in which $(q, j) = (3, 4)$.

We will later encounter two compositions of summations, where the summations within the composition have the same form. Instead of writing a sum (\sum) within each \bigsqcup operator, we will write the overall composition as

$$\bigsqcup_{q:t_1;s_1}^{j:v_1;u_1} \bigsqcup_{q:t_2;s_2}^{j:v_2;u_2} \sum_{k_{q,j}=0}^f$$

For instance,

$$\bigsqcup_{q:9;8}^{j:6;2} \sum_{k_{q,j}=0}^f \bigsqcup_{q:5;3}^{j:7;3} \sum_{k_{q,j}=0}^f$$

will be written as

$$\bigsqcup_{q:9;8}^{j:6;2} \bigsqcup_{q:5;3}^{j:7;3} \sum_{k_{q,j}=0}^f$$

In the formulas we will be working with, instead of writing

$$\sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,m})}$$

after each \bigsqcup operator, we will write the overall composition as

$$\bigsqcup_{q:t_1;s_1}^{j:v_1;u_1} \bigsqcup_{q:t_2;s_2}^{j:v_2;u_2} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,m})}$$

In this case, the summations

$$\bigsqcup_{q:t_1;s_1}^{j:v_1;u_1} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,m})}$$

are the outer summations and the summations

$$\bigsqcup_{q:t_2;s_2}^{j:v_2;u_2} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,m})}$$

are the inner summations.

Using this notation, we can express the formula for the number of fillings of a 2-by- n array even more concisely than before.

Proposition 8. *The number of fillings for a 2-by- n array where $n \geq 3$ is*

$$\bigsqcup_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n \nu_j + \triangleright^{2;3,n}) + 1.$$

8 The Number of Fillings for a 3-by- n Array

8.1 The Flexibility of Cell Placements for Arrays With More Than 2 Rows and 2 Columns

In 2-by-2 arrays and 2-by-2 subarrays of larger arrays, we found that the decision of which cell to place a value in first affects whether a filling with such a placement is a possible. While this is the case for the final 2-by-2 subarray of any array, it is not the case in any other place in any larger array. Provided that some cell in the larger array can be filled legitimately (i.e., within its row and column constraints), the cell above it and the cell to the left of it can be filled in any order, subject only to the revised row and column constraints pertaining to those individual cells.

This result will enable us to fill larger arrays one row at a time without fearing that legitimate placements of values in cells within any row will render other rows of the array impossible to fill. Additionally, this result describes a procedure in which all possible fillings may be obtained. We state it as a proposition.

Proposition 9. *In any m -by- n array, where both m and n are greater than 2 and the row and column constraints are each arranged in descending order, it is possible to always fill the array by first placing any value of $k_{m,n}$ from 0 to $\min(m,n)$ in cell $a_{m,n}$ and then placing, in any order, the values $k_{i,j-1}$ or $k_{i-1,j}$ in cells $a_{i,j-1}$ or $a_{i-1,j}$, provided that a legitimate placement of $k_{i,j}$ in cell $a_{i,j}$ has been made and that all cells $a_{p,q}$, where $p > i$ or $q > j$ have been legitimately filled. Each $k_{s,t}$ thus placed is subject to no restrictions besides the minimum of the revised row constraint and the revised column constraint of cell $a_{s,t}$. This procedure of filling the array can be performed until the original array has been reduced to a 2-by-2 subarray.*

We show Proposition 9 to be true as follows.

Let i and j both be greater than or equal to 3 and assume that cell $a_{i,j}$ and all cells below it and all cells to the right of it legitimately have been filled. Suppose that cell $a_{i,j}$, prior to being filled, is subject to revised constraints μ'_i and ν'_j . Cell $a_{i,j}$ is then filled with $k_{i,j}$ ranging from 0 to $\min(\mu'_i, \nu'_j)$. After cell $a_{i,j}$ is filled, cell $a_{i-1,j}$ will be subject to revised constraints μ'_{i-1} and $\nu'_j - k_{i,j}$. Likewise, after cell $a_{i,j}$ is filled, cell $a_{i,j-1}$ will be subject to revised constraints $\mu'_i - k_{i,j}$ and ν'_{j-1} . The value of μ'_{i-1} is not dependent on the value of $\mu'_i - k_{i,j}$. Likewise, the value of $\nu'_j - k_{i,j}$ is not dependent on the value of ν'_{j-1} . Since the constraints on $k_{i-1,j}$ and $k_{i,j-1}$ are independent, it is possible to place $k_{i-1,j}$ or $k_{i,j-1}$ in any order without the order of placement affecting how many fillings of the array are obtained.

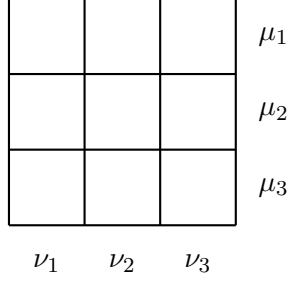
Since Proposition 9 holds for all i and j which are both greater than or equal to 3, it also holds for filling cell $a_{m,n}$ in an m -by- n array. Since cell $a_{m,n}$ can be legitimately filled by choosing $k_{m,n}$ ranging from 0 to $\min(\mu_m, \nu_n)$, it follows that $a_{m-1,n}$ and $a_{m,n-1}$ can be legitimately filled in any order. Once $a_{m-1,n}$ is filled, it is possible next to fill $a_{m-2,n}$ and $a_{m-1,n-1}$ in any order. Once $a_{m,n-1}$ is filled, it is possible to then fill $a_{m-1,n-1}$ and $a_{m,n-2}$ in any order. Indeed, any cell in the array can be filled provided that all cells below it and all the cells to the right of it have been filled. Thus, filling one column at a time or one row at a time and then moving on to the next column or row is entirely legitimate. We can continue filling cells in the array in this fashion until we reach the final 2-by-2 subarray.

Proposition 9 will not hold for any final 2-by-2 subarray of an array. This is because the placement of a value within any cell of a 2-by-2 subarray will determine the values within every other cell of the subarray. So it is not possible to freely place $k_{1,2}$ or $k_{2,1}$ after $k_{2,2}$ has been placed. Likewise, if the subscripted k to be placed is $k_{1,2}$ instead of $k_{2,2}$, this choice may affect whether a filling is possible. Such a placement decision may (though not always will) bring about the impossibility of filling the rest of the array if $\mu'_2 < \mu'_1$ or make a filling possible if $\mu'_1 < \mu'_2$.

8.2 The Number of Fillings for a 3-by-3 Array

We continue to find formulas for the numbers of fillings of increasingly larger arrays. We note that any m -by-2 array can be filled by filling the corresponding 2-by- m array and transposing the result so that the rows become the columns and vice versa. So the smallest array size for whose number of fillings we have yet to develop a formula is a 3-by-3 array.

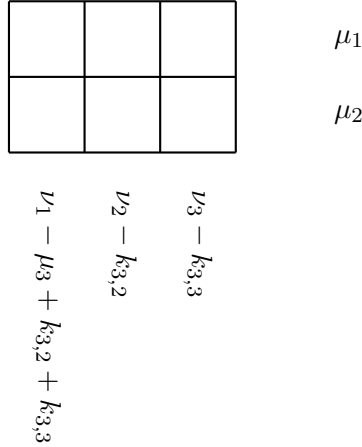
We consider a 3-by-3 array, with row and column constraints in descending order:



We first fill cell $a_{3,3}$ of the array with $k_{3,3}$, which can have any value from 0 to $\min(\mu_3, \nu_3)$. Then we fill cell $a_{3,2}$ of the array with $k_{3,2}$, which can have any value from 0 to $\min(\mu_3 - k_{3,3}, \nu_2)$. By Proposition 9, we can place $k_{3,2}$ without fear that this will impair our ability to fill the array later on. The value in cell $a_{3,1}$ will be determined by these two placements and will be $\mu_3 - k_{3,3} - k_{3,2}$. This determines the third row of the array, leaving us with a 2-by-3 array. Our outer summations are therefore

$$\sum_{k_{3,3}=0}^{\min(\mu_3, \nu_3)} \sum_{k_{3,2}=0}^{\min(\mu_3 - k_{3,3}, \nu_2)}$$

The resulting 2-by-3 array has row constraints μ_1 and μ_2 and column constraints $\nu_1 - \mu_3 + k_{3,2} + k_{3,3}$, $\nu_2 - k_{3,2}$, and $\nu_3 - k_{3,3}$:



By our previously derived formula for the number of fillings of a 2-by-3 array (Proposition 4), this array has $\sum_{k_{2,3}=0}^{\min(\mu_2, \nu_3 - k_{3,3})} (\min(\nu_2 - k_{3,2}, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3} + k_{3,3}) + 1)$ fillings.

Putting these summations together, we find that there are

$$\sum_{k_{3,3}=0}^{\min(\mu_3, \nu_3)} \sum_{k_{3,2}=0}^{\min(\mu_3 - k_{3,3}, \nu_2)} \sum_{k_{2,3}=0}^{\min(\mu_2, \nu_3 - k_{3,3})} (\min(\nu_2 - k_{3,2}, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3} + k_{3,3}) + 1)$$

fillings for a 3-by-3 array. We can express this result using our short-hand notation.

Proposition 10. *There are*

$$\prod_{q:3;3} \prod_{q:2;2}^{j:3;2} \sum_{k_{q,j}=0}^{j:3;2} \min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n}) \min(\nu_2 - k_{3,2}, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + \nabla^{3;2,3}) + 1$$

fillings for a 3-by-3 array.

or, using our short-hand notation,

$$\bigsqcup_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu'_j, \mu_q - \triangleright^{q;j+1,n})} \min(\nu'_2, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n \nu'_j + \triangleright^{2;3,n}) + 1.$$

For the inner summations representing the number of fillings of the 2-by- n subarray here, we must take into account the revised column constraints. Therefore, the number of fillings of a 2-by- n array with row constraints μ_1 and μ_2 and column constraints $\nu_1 - \mu_3 + \triangleright^{3;2,n}, \nu_2 - k_{3,2}, \nu_3 - k_{3,3}, \dots, \nu_n - k_{3,n}$ is

$$\begin{aligned} & \min(\nu_n - k_{3,n}, \mu_2) \min(\nu_{n-1} - k_{3,n-1}, \mu_2 - k_{2,n}) \min(\nu_{n-2} - k_{3,n-2}, \mu_2 - k_{2,n} - k_{2,n-1}) \cdots \min(\nu_3 - k_{3,3}, \mu_2 - k_{2,n} - k_{2,n-1} \cdots - k_{2,4}) \\ & \sum_{k_{2,n}=0} \sum_{k_{2,n-1}=0} \sum_{k_{2,n-2}=0} \cdots \sum_{k_{2,3}=0} \\ & \min(\nu_2 - k_{3,2}, \mu_2 - \sum_{i=3}^n k_{2,i}, \mu_1 - \sum_{j=3}^n \nu_j + \sum_{g=3}^n k_{2,g} + \sum_{h=3}^n k_{3,h}) + 1 \end{aligned}$$

or, using our short-hand notation,

$$\bigsqcup_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2 - k_{3,2}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n \nu_j + \triangleright^{2;3,n} + \triangleright^{3;3,n}) + 1$$

Putting together the outer and the inner summations above, we find that the number of fillings of a 3-by- n array, where $n \geq 3$, is

$$\begin{aligned} & \sum_{k_{3,n}=0}^{\min(\mu_3, \nu_n)} \sum_{k_{3,n-1}=0}^{\min(\mu_3 - k_{3,n}, \nu_{n-1})} \cdots \sum_{k_{3,3}=0}^{\min(\mu_3 - k_{3,n} - k_{3,n-1} - \cdots - k_{3,4}, \nu_3)} \sum_{k_{3,2}=0}^{\min(\mu_3 - k_{3,n} - k_{3,n-1} - \cdots - k_{3,3}, \nu_2)} \\ & \min(\nu_n - k_{3,n}, \mu_2) \min(\nu_{n-1} - k_{3,n-1}, \mu_2 - k_{2,n}) \min(\nu_{n-2} - k_{3,n-2}, \mu_2 - k_{2,n} - k_{2,n-1}) \cdots \min(\nu_3 - k_{3,3}, \mu_2 - k_{2,n} - k_{2,n-1} \cdots - k_{2,4}) \\ & \sum_{k_{2,n}=0} \sum_{k_{2,n-1}=0} \sum_{k_{2,n-2}=0} \cdots \sum_{k_{2,3}=0} \\ & \min(\nu_2 - k_{3,2}, \mu_2 - \sum_{i=3}^n k_{2,i}, \mu_1 - \sum_{j=3}^n \nu_j + \sum_{g=3}^n k_{2,g} + \sum_{h=3}^n k_{3,h}) + 1 \end{aligned}$$

We can express this result in our short-hand notation.

Proposition 11. *The number of fillings of a 3-by- n array, where $n \geq 3$, is*

$$\bigsqcup_{q:3;3}^{j:n;2} \bigsqcup_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2 - k_{3,2}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n \nu_j + \triangleright^{2;3,n} + \triangleright^{3;3,n}) + 1$$

We note that, fortunately, the revised first column constraint $\nu_1 - \mu_3 + k_{3,2} + k_{3,3} + \cdots + k_{3,n}$ is never present in this formula; nor is ν_1 present in the original formula for the number of fillings of a 2-by- n array. This is so because the values in cells $a_{1,1}$ and $a_{2,1}$ will always be determined by our prior choices of values in cells outside the first column; thus, the first column constraint need not come into the formula.

8.4 Proof of the Formula for the Number of Fillings for a 3-by- n Array

We argue by induction of the number of columns, n . When $n = 3$, it has been shown that there are

$$\sum_{k_{3,3}=0}^{\min(\mu_3, \nu_3)} \sum_{k_{3,2}=0}^{\min(\mu_3 - k_{3,3}, \nu_2)} \sum_{k_{2,3}=0}^{\min(\mu_2, \nu_3 - k_{3,3})} (\min(\nu_2 - k_{3,2}, \mu_2 - k_{2,3}, \mu_1 - \nu_3 + k_{2,3} + k_{3,3}) + 1)$$

fillings for a 3-by-3 array (Proposition 10). This agrees with the formula which we want to prove.

Now, we assume that that the formula holds for a 3-by- b array. By assumption, a 3-by- b array with row constraints μ_1, μ_2, μ_3 and column constraints $\nu'_1, \nu'_2, \nu'_3, \dots, \nu'_b$ has

$$\prod_{q:3:3} \prod_{q:2:2}^{j:b;2 \ j:b;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \triangleright^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,b})} \min(\nu'_2 - k_{3,2}, \mu_2 - \triangleright^{2;3,b}, \mu_1 - \sum_{j=3}^b \nu'_j + \triangleright^{2;3,b} + \triangleright^{3;3,b}) + 1$$

fillings.

We show that the formula holds for a 3-by- $(b+1)$ array with row constraints μ_1, μ_2, μ_3 and column constraints $\nu_1, \nu_2, \nu_3, \dots, \nu_b, \nu_{b+1}$. We examine a 3-by- $(b+1)$ array and attempt to fill the $(b+1)$ st column, so that we are left with a 3-by- b array to work with.

We fill cell $a_{3,b+1}$ of the array with $k_{3,b+1}$, which can have any value from 0 to $\min(\mu_3, \nu_{b+1})$. Then we fill cell $a_{2,b+1}$ of the array with $k_{2,b+1}$, which can have any value from 0 to $\min(\mu_2, \nu_{b+1} - k_{3,b+1})$. These two selections determine the $(b+1)$ st column.

Our outermost summations will thus be

$$\sum_{k_{3,b+1}=0}^{\min(\mu_3, \nu_{b+1})} \sum_{k_{2,b+1}=0}^{\min(\mu_2, \nu_{b+1} - k_{3,b+1})}$$

Now we are left with a 3-by- b array whose row constraints are $\mu_1 - \nu_3 + k_{2,b+1} + k_{3,b+1}$, $\mu_2 - k_{2,b+1}$, and $\mu_3 - k_{3,b+1}$, and whose column constraints are $\nu_1, \nu_2, \nu_3, \dots, \nu_n$. Therefore, this is a valid 3-by- b array. By our assumption, the number of ways to fill this array is

$$\prod_{q:3:3} \prod_{q:2:2}^{j:b;2 \ j:b;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \triangleright^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,b+1})} \min(\nu_2 - k_{3,2}, \mu_2 - \triangleright^{2;3,b+1}, \mu_1 - \sum_{j=3}^{b+1} \nu_j + \triangleright^{2;3,b+1} + \triangleright^{3;3,b+1}) + 1$$

We put together the outer and inner summations to get

$$\sum_{k_{3,b+1}=0}^{\min(\mu_3, \nu_{b+1})} \sum_{k_{2,b+1}=0}^{\min(\mu_2, \nu_{b+1} - k_{3,b+1})}$$

$$\prod_{q:3:3} \prod_{q:2:2}^{j:b;2 \ j:b;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \triangleright^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,b+1})} \min(\nu_2 - k_{3,2}, \mu_2 - \triangleright^{2;3,b+1}, \mu_1 - \sum_{j=3}^{b+1} \nu_j + \triangleright^{2;3,b+1} + \triangleright^{3;3,b+1}) + 1$$

This is not quite the order of summations in the formula we desire to prove. However, it is possible to rearrange the summations above in the desired order. By Proposition 9, it is possible

to place $k_{2,b+1}$ either immediately after $k_{3,b+1}$ has been placed or to wait to place $k_{2,b+1}$ until the entire third row has been placed. Making either decision will impair one's ability to fill the array successfully. Thus, the summation

$$\sum_{k_{2,b+1}=0}^{\min(\mu_2, \nu_{q+1}-k_{3,b+1})}$$

is not dependent on the choice of variables in the summations

$$\bigsqcup_{q:3:3}^{j:b;2} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,b+1})}$$

That is, the choices made in the former summation do not constrain our choices in any of the others.

Thus, it is possible to rearrange the order of these summations in the following way:

$$\sum_{k_{3,b+1}=0}^{\min(\mu_3, \nu_{b+1})} \bigsqcup_{q:3:3}^{j:b;2} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,b+1})} \min(\nu_2 - k_{3,2}, \mu_2 - \triangleright^{2;3,b+1}, \mu_1 - \sum_{j=3}^{b+1} \nu_j + \triangleright^{2;3,b+1} + \triangleright^{3;3,b+1}) + 1$$

fillings.

This expression can be simplified to give

$$\bigsqcup_{q:3:3}^{j:b+1;2} \bigsqcup_{q:2:2}^{j:b+1;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,b+1})} \min(\nu_2 - k_{3,2}, \mu_2 - \triangleright^{2;3,b+1}, \mu_1 - \sum_{j=3}^{b+1} \nu_j + \triangleright^{2;3,b+1} + \triangleright^{3;3,b+1}) + 1$$

fillings.

This is exactly our formula for the number of fillings for a 3-by- $(b+1)$ array. Thus, by the Principle of Mathematical Induction, the formula holds for any 3-by- n array. **Q. E. D.**

9 The Number of Fillings for a m -by- n Array

From now on, we assume that we are given an m -by- n array in which $m \geq 3$ and $n \geq 3$.

9.1 A Formula for the Number of Fillings of a m -by- n Array

We are now ready to find the formula for the number of fillings for a general m -by- n array. We will first develop a formula and then prove it by induction. We have thus far established the following regularities.

1. The second row requires the selection of $n-2$ values (of subscripted k 's) before a 2-by- n array becomes reduced to a 2-by-2 array. Once we reduce the array to a 2-by-2 subarray, we can no longer rely on Proposition 9 and must instead pick either $k_{1,2}$ or $k_{2,2}$ to finish filling the array.

2. The third row requires the selection of $n - 1$ values before a 3-by- n array becomes reduced to a 2-by- n array. Likewise, for any $m \geq 3$, the m th row requires the selection of $n - 1$ values before a m -by- n array becomes reduced to a $(m - 1)$ -by- n array. This makes sense, because choosing $n - 1$ values in a row of n cells will determine the value that fills the remaining cell. So for rows 3 through m , $n - 1$ values in each row will be chosen to fill those rows.
3. Thus, a m -by- n array will require choosing $(n - 2) + (m - 2)(n - 1) = n - 2 + mn - 2n - m + 2 = mn - n - m$ values of subscripted k 's before the array is reduced to a 2-by-2 array. Thus, the formula for the number of fillings of a m -by- n array will have $mn - n - m$ summations in it.

For instance, the formula for the number of fillings of a 2-by-3 array (Proposition 4) involves $2 \times 3 - 3 - 2 = 1$ summation. The formula for the number of fillings of a 3-by-7 array has $3 \times 7 - 7 - 3 = 11$ summations in it.

Also, when examining the formula for the number of fillings of a 3-by- n array (Proposition 11), we note that in the summations applied over the third row, the column constraints ν_j are present without modification. However, in the summations applied over the second row, the column constraints ν_j are modified by subtraction of $k_{3,j}$ from each ν_j . When filling the q th row of a m -by- n array, the column constraints will need to be modified by subtracting $\nabla^{j;q+1,m}$ from each ν_j to account for the cells already filled below $a_{q,j}$.

By examining the summations within each row in the formula for the number of fillings of a 3-by- n array (Proposition 11), we note that the first summation takes the minimum of the original row constraint and the appropriately modified column constraint. Going through the row, we modify the row constraint in each subsequent summation by subtracting from it the subscripted k 's already placed in that row. So when filling the cell $a_{q,j}$ of a m -by- n array the row constraint will need to be modified by subtracting $\triangleright^{q;j+1,n}$ from μ_q . In other words, the number of options for filling cell $a_{q,j}$ with some value $k_{q,j}$ corresponds to the summation

$$\min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n}) \sum_{k_{q,j}=0}^{\cdot} .$$

After the array has been reduced to a 2-by-2 case, we will need to consider the quantity within the composition of summations. This quantity will be one plus the minimum of three values. What values will we be taking the minimum of in the quantity within the summation?

Examining the formula for the number of fillings of a 3-by- n array (Proposition 11), we note that we take the minimum of the modified constraints ν_2 , μ_2 , and μ_1 . When the array is reduced to a 2-by-2 case, ν_2 is modified by subtracting all of the values placed in the second column in rows below the second row. Thus, $\nabla^{2;3,m}$ must be subtracted from ν_2 . The value of μ_2 is modified by subtracting all the values placed in the second row in columns to the right of the second column. Thus, $\triangleright^{2;3,n}$ must be subtracted from μ_2 . Similarly, μ_1 is modified by subtracting all the values placed in the first row in columns to the right of the second column. But these values are not $k_{1,j}$'s that we have placed; rather, each of these values is the difference of some ν_j and the sum of the subscripted k 's that have been placed in the second through the m th row of the j th column as dictated by the column constraints. Thus, μ_1 is modified by subtracting $\sum_{j=3}^n (\nu_j - \nabla^{j;2,m})$.

Hence, our last minimum within the summations, the minimum which represents the cell in which we place a value after only a 2-by-2 array remains to be filled within the original m -by- n array, is

$$\min(\nu_2 - \nabla^{2;3,m}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,m}))$$

and the quantity within the summations is

$$\min(\nu_2 - \nabla^{2;3,m}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,m})) + 1.$$

A single application of the \sqcup operator will suffice to describe the composition of summations pertaining to a *rectangle* of cells. In placing our subscripted k 's, we are faced with two distinct rectangles, since we place $(n-2)$ subscripted k 's in the second row, and $(n-1)$ subscripted k 's in each row from the third to the m th. Thus, we have a 1-by- $(n-2)$ rectangle and a $(m-2)$ -by- $(n-1)$ rectangle to fill with subscripted k 's.

We fill the $(m-2)$ -by- $(n-1)$ rectangle using the following composition of summations:

$$\bigsqcup_{q:m;3}^{j:n;2} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n})}$$

These will be the outer summations of our formula.

We fill the 1-by- $(n-2)$ rectangle using the following composition of summations:

$$\bigsqcup_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n})}$$

These will be the inner summations of our formula.

As we already determined, the quantity within the summations is

$$\min(\nu_2 - \nabla^{2;3,m}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,m})) + 1$$

Putting these together, we get the following formula.

Proposition 12. *The number of fillings of an m -by- n array, where $m \geq 3$ and $n \geq 3$, is*

$$\bigsqcup_{q:m;3}^{j:n;2} \bigsqcup_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,m}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2 - \nabla^{2;3,m}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,m})) + 1.$$

9.2 Proof of the Formula for the Number of Fillings of a General m -by- n Array

We will use induction on m , the number of rows in the array.

Part 1: The 3-by- n case is the simplest instance of our general formula. We already showed (Proposition 11) that the number of fillings for a 3-by- n array is

$$\bigsqcup_{q:3;3}^{j:n;2} \bigsqcup_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,3}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2 - \nabla^{2;3,3}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,3})) + 1.$$

Thus, our formula in Proposition 12 holds for a 3-by- n array.

Part 2: Now we assume that the formula in Proposition 12 holds for a r -by- n array and show that it also holds for a $(r + 1)$ -by- n array. This will suffice to prove the formula in Proposition 12 because we are performing induction on rows without regard to the number of columns.

By assumption, the number of fillings for a r -by- n array is

$$\prod_{q:r;3}^{j:n;2} \prod_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu'_j - \nabla^{j;q+1,r}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu'_2 - \nabla^{2;3,r}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu'_j - \nabla^{j;2,r})) + 1$$

We fill a $(r + 1)$ -by- n array by first filling the $(r + 1)$ st row. We know from prior reasoning that this requires $(n - 1)$ placements of subscripted k 's, the summation for each of which will be of the form

$$\sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,r+1}, \mu_q - \triangleright^{q;j+1,n})}$$

In the expression we will develop for filling the $(r + 1)$ st row, for all places where r was present in the assumption, $(r + 1)$ will be present instead. This is done to account for the fact that this r -by- n array has different column constraints from the array in the assumption. The r -by- n array in the assumption had constraints $\nu'_1, \nu'_2, \nu'_3, \dots, \nu'_n$, whereas the r -by- n array left over after the $(r + 1)$ st row of $(r + 1)$ -by- n array has been filled has column constraints $\nu_1 - \mu_n + \triangleright^{r+1;2,n}, \nu_2 - k_{r+1,2}, \nu_3 - k_{r+1,3}, \dots, \nu_n - k_{r+1,n}$. So an additional subscripted k needs to be subtracted from ν_j within each of the summations to account for the presence of that k in the j th column of the $(r + 1)$ st row. This transforms $\nabla^{j;q+1,r}$ into $\nabla^{j;q+1,r+1}$.

Using our concise notation, the expression for filling the $(r + 1)$ st row is thus

$$\prod_{q:r+1;r+1}^{j:n;2} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,r+1}, \mu_q - \triangleright^{q;j+1,n})}$$

Now we find the number of fillings of the left-over r -by- n array. Compared with the formula for the number of fillings of the r -by- n array in the assumption, an additional subscripted k needs to be subtracted from ν_2 in the minimum within all of the summations, turning $\nabla^{2;3,r}$ into $\nabla^{2;3,r+1}$. Finally, the addition of a new row means that the expression for the value of each entry in the first row to the right of the second column must now take into account the value of the subscripted k 's in the newly added row. Thus,

$$\sum_{j=3}^n (\nu_j - \nabla^{j;2,r})$$

becomes

$$\sum_{j=3}^n (\nu_j - \nabla^{j;2,r+1})$$

and this latter term is subtracted from μ_1 in the minimum within all of the summations.

This leaves us with a r -by- n array whose number of fillings, by assumption, is

$$\prod_{q:r;3}^{j:n;2} \prod_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,r+1}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2 - \nabla^{2;3,r+1}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,r+1})) + 1$$

Composing the outer summations that fill the $(r + 1)$ st row with the expression for the number of fillings of the resultant r -by- n array, we get

$$\prod_{q:r+1;r+1}^{j:n;2} \prod_{q:r;3}^{j:n;2} \prod_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,r+1}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2 - \nabla^{2;3,r+1}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,r+1})) + 1$$

which can be rewritten as

$$\prod_{q:r+1;3}^{j:n;2} \prod_{q:2;2}^{j:n;3} \sum_{k_{q,j}=0}^{\min(\nu_j - \nabla^{j;q+1,r+1}, \mu_q - \triangleright^{q;j+1,n})} \min(\nu_2 - \nabla^{2;3,r+1}, \mu_2 - \triangleright^{2;3,n}, \mu_1 - \sum_{j=3}^n (\nu_j - \nabla^{j;2,r+1})) + 1$$

since we have simply added another row to the larger $(m-2)$ -by- $(n-1)$ rectangle instead of adding a new rectangle that is different in all dimensions.

The above formula is exactly our formula in Proposition 12 for the number of fillings of a $(r+1)$ -by- n array. Thus, if the formula holds for a r -by- n array, it holds for a $(r+1)$ -by- n array. We have just established a general formula for the number of fillings of a m -by- n array, where both m and n are at least 3. **Q. E. D.**

10 Questions for Further Exploration

10.1 The Symmetric Polynomial Approach to Filling Arrays

Here, we will briefly examine a possible line of inquiry for which Proposition 12 might be useful. Before we can articulate these further questions, we will need to introduce some definitions. We refer the reader to Cameron [2] for more details.

A *partition* of n is “an expression for n as a sum of positive integers where the order of the summands is unimportant. We can arrange the parts in order, with the largest first” [2]. For instance, the partitions of 3 are 3, 2 + 1, and 1 + 1 + 1, so there are three partitions of 3 in total. We will write “ λ is a partition of n ” as $\lambda \vdash n$.

For some variables x_1, x_2, \dots, x_N , a polynomial $f(x_1, x_2, \dots, x_N)$ is called a *symmetric polynomial* if permuting the variables x_1, x_2, \dots, x_N does not change the polynomial. For any altered ordering $x_{1g}, x_{2g}, \dots, x_{Ng}$ of the variables x_1, x_2, \dots, x_N , it will be the case that $f(x_1, x_2, \dots, x_N) = f(x_{1g}, x_{2g}, \dots, x_{Ng})$.

We now consider several specific kinds of symmetric polynomials. Each of these polynomials will be considered with respect to the partition λ of n , where λ is the partition $n = n_1 + n_2 + \dots + n_k$.

The *basic polynomial*, denoted m_λ , is the “sum of the term $x_1^{n_1} x_2^{n_2} \dots x_k^{n_k}$ and all the other terms which be obtained from this one by permuting the indeterminates” [2]. For instance, if λ is the partition $3 = 2 + 1$ and $N = 3$, then $m_\lambda = x_1^2 x_2 + x_1 x_2^2 + x_1^2 x_3 + x_1 x_3^2 + x_2^2 x_3 + x_2 x_3^2$.

The *elementary symmetric polynomial*, denoted e_n , is “the sum of all products of n distinct indeterminates” [2]. If λ is the partition $n = n_1 + n_2 + \dots + n_k$, we define $e_\lambda = e_{n_1} e_{n_2} \dots e_{n_k}$. For instance, if λ is the partition $3 = 2 + 1$ and $N = 3$, then $e_\lambda = e_2 e_1 = (x_1 x_2 + x_1 x_3 + x_2 x_3)(x_1 + x_2 + x_3)$.

Proposition 13. *We consider a matrix A whose entries can only have values of 0 or 1. We let the row sums of A be $n_1 \geq n_2 \geq \dots \geq n_k > 0$, and we let the column sums of A be $m_1 \geq m_2 \geq \dots \geq m_l > 0$. We also let λ and μ be the partitions $\lambda : n = n_1 + n_2 + \dots + n_k$ and $\mu : n = m_1 + m_2 + \dots + m_l$.*

If these conditions hold, then, according to Cameron [2] in Section 13.6, Exercise 11, the polynomial e_λ is expressible as

$$e_\lambda = \sum_{\mu \vdash n} a_{\lambda\mu} m_\mu,$$

where $a_{\lambda\mu}$ is the number of matrices A which satisfy the conditions given above.

The value of $a_{\lambda\mu}$ is *not* the same as the number of fillings for a k -by- l array as given by Proposition 12. The formula of Proposition 12 permits the placement of *any* numerical values within the cells of the array, provided that those values satisfy all row and column constraints. The value $a_{\lambda\mu}$, however, only counts the number of ways to fill a k -by- l array with only 0's and 1's present in each cell.

We have seen in Section 2 that it is always possible to fill a k -by- l array for which the sum of the row constraints is equal to the sum of the column constraints, provided that one is permitted to choose any numerical values in each cell which satisfy all the row and column constraints. When we are restricted to only placing 0's and 1's in cells of the array, however, it is possible that a given array might not have a valid filling at all. For instance, we consider the following array.

		3
		3
3	3	

There is no way to fill this array using only 0's and 1's, since a 0 placed in a cell of any row would require 3 in the other cell of that row to satisfy its row constraint. Moreover, a 1 placed in a cell of any row would require a 2 in the the other cell of that row to satisfy its row constraint. Thus, the value of $a_{\lambda\mu}$ associated with this array is 0. This agrees with the result of Proposition 13.

A possibly fruitful avenue for future exploration might involve developing a formula akin to that of Proposition 13 for using symmetric polynomials to count the number of fillings of *any* m -by- n array for which the sum of the row constraints is equal to the sum of the column constraints and the value in each cell is *not* restricted to 0 or 1. This formula, if it can be attained, would yield another way, in addition to Proposition 12, of counting the number of fillings of such an array.

Another possible approach to this question might involve trying to find a formula akin to Proposition 12 for the number of fillings of numerical arrays where each cell value *is* restricted to either 0 or 1. This approach, if successful, would result in a way to find $a_{\lambda\mu}$ using summations rather than symmetric polynomials.

10.2 Sensitivity to Constraints and Complexity of the Problem

Still other questions for further exploration concern the sensitivity of m -by- n numerical arrays to the values of the row and column constraints. For instance, if a particular row constraint is increased by some value s and the sum of the column constraints is also increased by s , how many more fillings will the new array have as compared to the previous array? Would increasing an already larger constraint produce more fillings than increasing an already smaller constraint, or is the reverse true?

It may also be fruitful to explore how the number of fillings is affected when one vector of constraints (say, the vector of row constraints) is held constant while the other vector of constraints (say, the vector of column constraints) is allowed to vary, provided that the sum of the row constraints is equal to the sum of the column constraints. Would a balanced vector of column constraints (say, $(3, 3, 3, 3, 3)$ for $d = 15$) result more fillings, or would a skewed vector of column constraints (say, $(5, 4, 3, 2, 1)$ for $d = 15$) render a greater number of fillings possible?

Moreover, it may be useful to investigate the complexity of the formula of Proposition 12, i.e., how many computations would be required to determine the number of fillings for any particular m -by- n numerical array. Applying the formula of Proposition 12 to many larger arrays requires too many computations for a human being to perform by hand in any reasonable amount of time. Therefore, using a computer to apply the formula to such larger arrays would be necessary. To estimate the amount of time the computer can be expected to take in solving the problem, some understanding of our formula's complexity would be helpful.

We have already noted in Subsection 9.1 that the formula for the number of fillings for an m -by- n numerical array will have $mn - n - m$ summations. This insight may be useful in determining the number of calculations required to obtain the number of fillings for a given numerical array. An upper estimate of the formula's complexity is $(mn - n - m)(d + 1)$, since the subscripted k pertaining to each summation can at most assume integer values ranging from 0 to d , the sum of the row/column constraints.

10.3 Extensions of the Array-Filling Problem

Further restrictions can be imposed on our array-filling problem. For instance, certain kinds of array fillings might be disallowed to reflect a situation modified from that of the scenario described in Subsection 1.4. If, for instance, certain groups of workers refuse to work together on the same task, this would restrict our options for assigning them. For example, workers from team i and workers from team f have a mutual animosity. If workers from team i have been assigned to project j , then workers from team f should not be assigned to the same project, and vice versa. In a numerical array, this restriction would be reflected as follows. If column j has a non-zero entry in row i , then it must have a zero entry in row f , and vice versa. It would be interesting to examine how such a restriction would affect the number of possible array fillings and whether there exists a systematic way of determining this effect.

Likewise, it may be instructive to consider how further restricting the values of array cells would affect the number of possible fillings. For instance, if a particular cell or cells were required to have a certain value, how many fillings would be possible with that restriction? How would the number of fillings be affected if a given cell or cells were required to have values greater than a certain non-zero number? For instance, if cell $a_{i,j}$ were required to have a value greater than or equal to p , where p is an integer greater than 0, how many possible fillings of the given array would exist under such a condition?

A Permutations of Rows and Columns With Equal Constraints

A.1 Motivation for the Permutation Approach

Prior to the approach illustrated in this thesis, another approach was attempted in the effort to solve the array-filling problem. This method sought to combine directional filling algorithms with permutations of rows and columns whose constraints were of equal value.

For instance, we consider the following array.

				1
				1
				1
				1
1	1	1	1	

0	0	0	1
0	1	0	0
0	0	1	0
1	0	0	0

In Section 4, we showed that the filling cannot be obtained solely from directional

0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

row-by-row filling algorithms applied to this array. However, we can obtain the filling by filling the array row by row from right to left. Then we can exchange rows 2 and 3 to get the desired filling.

This example suggested that it might be fruitful to attempt to obtain all the possible fillings of a numerical array via permutations applied to various fillings obtained by means of directional filling algorithms. Ultimately, however, insurmountable difficulties with this method prevented it from being used to obtain the final result.

A.2 Row Permutations Always Commute With Column Permutations

In an attempt to reduce the number of permutation compositions needing to be examined for each array, an effort was undertaken to prove the commutativity of as many of these permutation compositions as possible. This effort was successful in showing that row permutations always commute with column permutations.

For some arbitrary filled array Q , we let $\rho(x, y)$ be a permutation that switches the positions of rows x and y . Furthermore, we let $\chi(d, f)$ be a permutation that switches the positions of columns d and f in a numerical array. We show that these two permutations always commute. The only cells affected by both of the permutations are $\alpha = a_{x,d}$, $\beta = a_{x,f}$, $\gamma = a_{y,d}$, and $\delta = a_{y,f}$.

First, we consider $\chi(d, f)\rho(x, y)Q$.

$\rho(x, y)$ moves α to $a_{y,d}$ and β to $a_{y,f}$.

Thus, the configuration for $\rho(x, y)Q$ is $\gamma = a_{x,d}$, $\delta = a_{x,f}$, $\alpha = a_{y,d}$, and $\beta = a_{y,f}$.

Now we apply $\chi(d, f)$ to $\rho(x, y)Q$. $\chi(d, f)$ switches the positions of γ and δ ; it also switches the positions of α and β . Thus, for $\chi(d, f)\rho(x, y)Q$, the configuration is $\delta = a_{x,d}$, $\gamma = a_{x,f}$, $\beta = a_{y,d}$, and $\alpha = a_{y,f}$.

Next, we consider $\rho(x, y)\chi(d, f)Q$.

The original Q arrangement is $\alpha = a_{x,d}$, $\beta = a_{x,f}$, $\gamma = a_{y,d}$, and $\delta = a_{y,f}$.

Now we apply $\chi(d, f)$ to Q . $\chi(d, f)$ switches the positions of α and β and those of γ and δ . Thus, the arrangement for $\chi(d, f)Q$ is $\beta = a_{x,d}$, $\alpha = a_{x,f}$, $\delta = a_{y,d}$, and $\gamma = a_{y,f}$.

Next, we apply $\rho(x, y)$ to $\chi(d, f)Q$. $\rho(x, y)$ switches the positions of β and δ and those of α and γ . Thus, the arrangement for $\rho(x, y)\chi(d, f)Q$ is $\delta = a_{x,d}$, $\gamma = a_{x,f}$, $\beta = a_{y,d}$, and $\alpha = a_{y,f}$.

which is identical to the arrangement for $\chi(d, f)\rho(x, y)Q$ in terms of the only cells whose values are affected by both permutations. Thus, in every case, $\rho(x, y)\chi(d, f)Q = \chi(d, f)\rho(x, y)Q$ and a row permutation always commutes with a column permutation of a filled array. **Q. E. D.**

A.3 Two Row Permutations Do Not Necessarily Commute

While a row permutation always commutes with a column permutation, this cannot be said of two row permutations or two column permutations, a fact that complicates the attempt to find array fillings using this approach.

As a counterexample to the commutativity of two row permutations, we consider the row permutations $\rho(3, 4)$ and $\rho(4, 5)$ applied to some array Q , where Q has 5 or more rows, and rows 3, 4, and 5 have identical constraints. In the given array Q , we let $c = \text{row } 3$, $d = \text{row } 4$, and $e = \text{row } 5$. Then $\rho(3, 4)\rho(4, 5)Q$ results in the following arrangement: $e = \text{row } 3$, $c = \text{row } 4$, $d = \text{row } 5$, while $\rho(4, 5)\rho(3, 4)Q$ results in the following arrangement: $d = \text{row } 3$, $e = \text{row } 4$, $c = \text{row } 5$.

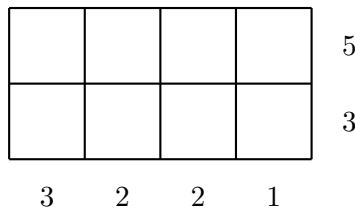
These arrangements are not the same, so $\rho(4, 5)\rho(3, 4)Q \neq \rho(3, 4)\rho(4, 5)Q$. Therefore, two row-permutations that leave the row-constraint vector invariant must not necessarily commute with one another. **Q. E. D.**

However, commutativity always holds whenever the row permutations to be commuted are $\rho(w, x)$ and $\rho(y, z)$, with w, x, y , and z being distinct from one another. Thus, permutations for rows or columns with distinct constraints always commute. Commutativity does not hold when the permutations to be composed are $\rho(w, x)$ and $\rho(w, z)$, i.e., when one of the rows is affected by each of the two permutations. (It suffices to substitute w for 3, x for 4, z for 5 in the counterexample above to prove the generic case.)

A.4 Directional Filling Algorithms and Permutations Do Not Suffice to Find All Fillings

Given that row permutations do not always commute with each other and that the same applies to column permutations, considerable work can be involved in examining every possible distinct composition of permutations applied to an array filled via a certain directional algorithm. Even if this work were carried out, however, it might not produce all fillings of an array.

We consider the following array.



By the formula for the number of fillings of a 2-by-4 array (Proposition 5), this array has the following number of fillings.

$$\begin{aligned}
& \sum_{k_{2,4}=0}^{\min(\mu_2, \nu_4)} \sum_{k_{2,3}=0}^{\min(\mu_2 - k_{2,4}, \nu_3)} (\min(\nu_2, \mu_2 - k_{2,3} - k_{2,4}, \mu_1 - \nu_4 - \nu_3 + k_{2,3} + k_{2,4}) + 1) \\
&= \sum_{k_{2,4}=0}^{\min(3, 1)} \sum_{k_{2,3}=0}^{\min(3 - k_{2,4}, 2)} (\min(2, 3 - k_{2,3} - k_{2,4}, 5 - 1 - 2 + k_{2,3} + k_{2,4}) + 1)
\end{aligned}$$

$$\begin{aligned}
&= \sum_{k_{2,4}=0}^1 \sum_{k_{2,3}=0}^2 (\min(2, 3 - k_{2,3} - k_{2,4}, 2 + k_{2,3} + k_{2,4}) + 1) \\
&= \sum_{k_{2,4}=0}^1 \sum_{k_{2,3}=0}^2 (\min(2, 3 - k_{2,3} - k_{2,4}) + 1)
\end{aligned}$$

The values $k_{2,4} = 0$ and $k_{2,3} = 0$, produce $2 + 1 = 3$ fillings.

The values $k_{2,4} = 1$ and $k_{2,3} = 0$, produce $2 + 1 = 3$ fillings.

The values $k_{2,4} = 0$ and $k_{2,3} = 1$, produce $2 + 1 = 3$ fillings.

The values $k_{2,4} = 1$ and $k_{2,3} = 1$, produce $1 + 1 = 2$ fillings.

The values $k_{2,4} = 0$ and $k_{2,3} = 2$, produce $1 + 1 = 2$ fillings.

The values $k_{2,4} = 1$ and $k_{2,3} = 2$, produce $0 + 1 = 1$ filling.

Thus, there are total of 14 fillings to this array. This result was confirmed by using the Array Filler program whose source code is included in Appendix B.

However, considering only directional filling algorithms and the application to these fillings of the column permutation $\chi(2, 3)$, which is the only applicable permutation here, we can only get the following three fillings.

$$\begin{array}{|c|c|c|c|} \hline 3 & 2 & 0 & 0 \\ \hline 0 & 0 & 2 & 1 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 0 & 2 & 2 & 1 \\ \hline 3 & 0 & 0 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 3 & 0 & 2 & 0 \\ \hline 0 & 2 & 0 & 1 \\ \hline \end{array}$$

The following eleven fillings are still left unaccounted for:

$$\begin{array}{|c|c|c|c|} \hline 3 & 0 & 1 & 1 \\ \hline 0 & 2 & 1 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 3 & 1 & 0 & 1 \\ \hline 0 & 1 & 2 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 1 & 1 & 2 & 1 \\ \hline 2 & 1 & 0 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 1 & 2 & 1 & 1 \\ \hline 2 & 0 & 1 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 2 & 2 & 1 & 0 \\ \hline 1 & 0 & 1 & 1 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 2 & 1 & 2 & 0 \\ \hline 1 & 1 & 0 & 1 \\ \hline \end{array}, \\
\begin{array}{|c|c|c|c|} \hline 1 & 2 & 2 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 2 & 0 & 2 & 1 \\ \hline 1 & 2 & 0 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 2 & 2 & 0 & 1 \\ \hline 1 & 0 & 2 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 2 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}, \quad
\begin{array}{|c|c|c|c|} \hline 3 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 1 \\ \hline \end{array}.$$

This counterexample shows that even when we apply all possible directional filling algorithms for an array and obtain all possible permutations of these fillings, this will not suffice to describe all the fillings of even fairly small arrays. Thus, such an approach was ultimately abandoned in favor of the approach discussed in the body of this paper.

B Source Code for Nathan Sprague's Array Filler Program

This is the source code for an array filler program written in Python by Dr. Nathan Sprague. This program, when executed, can list and count the fillings of any given m -by- n array. The program was used to test the results for the examples displayed in this paper.

```

def subtractArray(a,b):
    res = [0] * len(a)
    for i in range(len(a)):
        res[i] = a[i] - b[i]
    return res

def printMatrix(a):
    for i in range(len(a)):

```

```

        for j in range(len(a[0])):
            print a[i][j],
        print
    print

COUNT = 0

#countSolutions -----
#row_goals, col_goals - list of values that rows/cols must sum to.
#display - 1 if solutions should be printed out, 0 otherwise.
#No error checking of inputs is performed.

def countSolutions(row_goals, col_goals, display):
    global COUNT
    COUNT= 0
    nrows =len(row_goals)
    ncols =len(col_goals)
    solution = [[0] * ncols for i in range(nrows)] #python way of creating rows x cols matrix
    countSolutionsIn(0, 0, nrows, ncols, row_goals[0], row_goals, col_goals, solution, display)
    print "Number of solutions: " + str(COUNT)

#Don't call this directly. This is the recursive bit of countSolutions.
def countSolutionsIn(row, col, nrows, ncols, num_left_row, row_goals, col_goals,
                    solution, display):
    global COUNT

    if col == ncols:    #finished a row
        if num_left_row == 0: #successful row!
            if (row == nrows - 1): #SUCCESS!!
                COUNT = COUNT+1
                if display != 0:
                    printMatrix(solution)
                return
            else:
                #Move to the next row...
                col_goals = subtractArray(col_goals, solution[row])
                countSolutionsIn(row+1, 0, nrows, ncols, row_goals[row+1], row_goals, col_goals,
                                solution, display)
                return
        else:
            #failed row.
            return

    #try every valid setting for the current cell...
    for i in range(min(num_left_row, col_goals[col])+1):
        solution[row][col] = i;
        countSolutionsIn(row, col+1, nrows, ncols, num_left_row - i, row_goals, col_goals,
                        solution, display)

```

References

- [1] Brualdi, Richard. *Introductory Combinatorics*, 1st Edition. North-Holland. 1977.
- [2] Cameron, Peter J. *Combinatorics: Topics, Techniques, Algorithms*, 1st Edition. Cambridge University Press, Cambridge. 1994.
- [3] Felgenhauer, Bertram and Frazer Jarvis. “Mathematics of Sudoku I.” Sudoku enumeration problems. 14 April 2008. <<http://www.afjarvis.staff.shef.ac.uk/sudoku/>>.
- [4] Jarvis, Frazer and Ed Russell. “Mathematics of Sudoku II.” Sudoku enumeration problems. 14 April 2008. <<http://www.afjarvis.staff.shef.ac.uk/sudoku/>>.