Forthcoming – Journal of Mathematical Modelling and Algorithms

A Unified Approach to Algorithms Generating Unrestricted and Restricted Integer Compositions and Integer Partitions

John Douglas (J.D.) Opdyke*

Abstract

An original algorithm is presented that generates both restricted integer compositions and restricted integer partitions that can be constrained simultaneously by a) upper and lower bounds on the number of summands ("parts") allowed, and b) upper and lower bounds on the values of those parts. The algorithm can implement each constraint individually, or no constraints to generate unrestricted sets of integer compositions or partitions. The algorithm is recursive, based directly on very fundamental mathematical constructs, and given its generality, reasonably fast with good time complexity. A general, closed form solution to the open problem of counting the number of integer compositions doubly restricted in this manner also is presented; its formulaic link to an analogous solution for counting doubly-restricted integer partitions is shown to mirror the algorithmic link between these two objects.

Mathematics Subject Classifications: 05A07, 11P82, 11Y16, 11Y55
Keywords: Integer Compositions, Integer Partitions, Bounded Compositions, Bounded Partitions, Pascal's triangle, Fibonacci

© 2008 by John Douglas Opdyke. All rights reserved. Short sections of text, not to exceed two paragraphs, may be quoted without explicit permission provided that full credit, including © notice, is given to the source.

Introduction

A list of integers greater than zero that sum to the positive integer n is an integer partition of n. The set of all such lists is the set of all integer partitions of n. For example, for n=4, the set of integer partitions includes the lists 1 1 1 1, 1 1 2, 2 2, 1 3, and 4. When the ordering of the summands ("parts") matters, these become the integer compositions of n: 1 1 1 1, 2 1 1, 1 2 1, 1 1 2 , 2 2, 3 1, 1 3, and 4. A number of algorithms exist for generating all compositions, and other, quite distinct algorithms exist for generating all partitions. Knuth (1997) is widely known and used for compositions, and others include Ehrlich (1973), Klingsberg (1982), Ruskey (2003), Arndt (2008), and Stojmenovic (2008). For partitions the list is extensive, but a sample includes several presented in Knuth (1997), two developed in Zoghbi & Stojmenivc (1998), which have been the state of the art, and several that have been developed more recently in Yamanaka et al. (2007), including versions that generate restricted integer partitions as well.

When the set of either integer compositions or integer partitions is "restricted," it includes only a subset of the lists that satisfy some restricting conditions. The most common such restrictions include constraining a) the number of parts allowed in any of the lists, and/or b) the values of those parts (other restrictions include constraining the compositions or partitions to fixed numbers of distinct parts, restricting the values of only the leading and trailing parts, identifying only those compositions or partitions containing a particular part or excluding a particular part, forcing consecutive parts to sum to specific values, etc). For example, the set of restricted integer compositions of n=4 that have at least one part and no more than two parts, and parts with values within the range of 2 to 4, includes: 2 2, and 4.

Both restricted integer compositions and restricted integer partitions are fundamental combinatorial objects essential in many mathematical, statistical, and scientific applications, including, respectively, an extensive number of combinatorial problems, efficient enumeration of restricted and unrestricted sample spaces, and atomic behavioral problems in physics. Yet no general algorithm exists to generate either when the two abovementioned

---

* J.D. Opdyke is President of DataMineIt, a statistical data mining consultancy. He also is serving as the Quantitative Director of Correlation Ventures, a venture capital firm. Email: JDOpdyke@DataMineIt.com, JDO@CorrelationVC.com, Website: http://www.DataMineIt.com. The author is grateful to Toyo Johnson for her belief in the "Christmas tree" that is Pascal's triangle, and to Dean "Gordy" Fairchild, PhD, Director, American Express, work for whom motivated inquiry into this topic.

restrictions are applied concurrently. The only similar algorithm for restricted integer compositions appears to be that of Walsh (2000), which restricts individual part values with maximum values, but not minimum values, too (neither does it allow *only* a minimum value to be specified without a maximum value). It also does not explicitly allow the user to restrict the number of parts allowed. For restricted integer partitions, White (1970b) presents an algorithm with only a lower value bound, and no bounds on the number of parts generated. Ruskey (2003), Knuth (1974, 1994), and Yamanaka (2007) present algorithms that concurrently allow an upper bound on the number of parts and an upper value bound, but not lower bounds, too (neither do they allow *only* a lower bound to be specified without a maximum value bound). This paper develops a unified algorithm that does both: simultaneously restricts both integer compositions and integer partitions with upper and lower bounds both the number of parts allowed, as well as the values of those parts. No other algorithm can make this claim. It also generates the unrestricted cases and singly-restricted cases of both objects as well. The algorithm, "RICs_RIPs" (Restricted Integer Compositions, Restricted Integer Partitions), is based on the (Fibonacci series) off-diagonals of Pascal's triangle and composition and partition "paths" found in the triangle via its representation as ordered binomial coefficients. RICs_RIPs is recursive and, given its generality, reasonably fast with good time complexity (approximately $\mathrm{O}(k)$ per composition/partition, where k = # parts). This paper also provides a general, closed form solution to the open problem of counting the number of integer compositions doubly restricted in this manner; its formulaic link to an analogous solution for counting integer partitions is shown to mirror the previously unidentified algorithmic link between these two objects. I initially focus below on restricted integer compositions, and return to partitions later in the paper.

Background

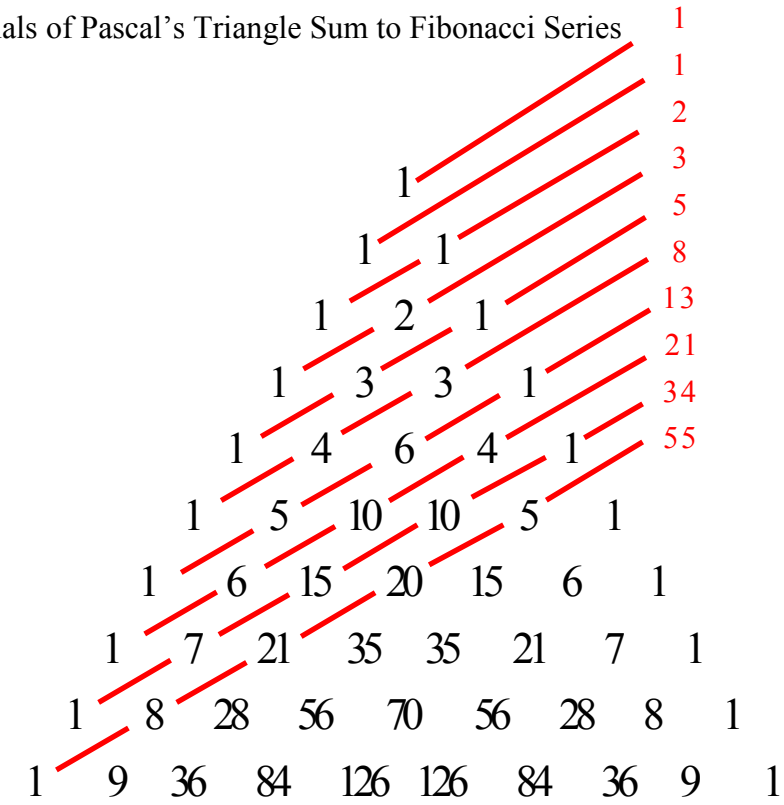
Algorithms for generating all integer compositions of n are well known (see above), as is the formula for counting the number of integer compositions of n, $c(n) = 2^{(n-1)}$ (see Andrews, 1998). For a specific number of parts k, $c(n,k) = \binom{n-1}{k-1}$, and summing every k up to n gives $\sum_{k=1}^{n} c(n,k) = \sum_{k=1}^{n} \binom{n-1}{k-1} = 2^{(n-1)}$, the total number of compositions. Following Kimberling's (2001) notation, if we restrict the values of those compositions by a minimum value of "a" and a maximum value of "b" for $c(n,k,a,b)$, then with $b=\infty$ (in practice, $b=n$) we have $c(n,k,a,\infty) = c(n-ka,k,0,\infty) = \binom{n-ka+k-1}{k-1}$. With a=2, $\sum_{k=1}^{\lfloor n/2 \rfloor} c(n,k,2,\infty) = F_{n-1}$ (see Grimaldi, 2000), where F_n is the well known Fibonacci series defined by $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$. Put differently, $F_{n-1} = c(n \mid \text{all parts} > 1)$, or equivalently $F_n = c(n+1 \mid \text{all parts} > 1)$. Note that when a=3, $F2_n = c(n+1 \mid \text{all parts} > 2)$, where $F2_n = F2_{n-1} + F2_{n-3}$, and when a=4, $F3_n = c(n+1 \mid \text{all parts} > 3)$, where $F3_n = F3_{n-1} + F3_{n-4}$, and so on (with $F\#_{n<a} = 0$ and $F\#_a = 1$; see Andrews, 1998; for a=3, 4, and 5, these are series A078012, A017898, and A017899, respectively, in Sloan's Online Encyclopedia of Integer Sequences). These "Fibonacci-shifted" relationships and their growth rates are shown in Table 1.

Table 1: Counts of Integer Compositions with Minimum-Valued Parts

Asymptotic Growth Rate →	**2.0**	$\phi = \left(1+\sqrt{5}\right)\big/2 = 1.61803$	**1.46557124**	**1.38028287**	**1.32474227**
n	**all parts (*a*=1)**	**all parts > 1 (*a*=2)**	**all parts > 2 (*a*=3)**	**all parts > 3 (*a*=4)**	**all parts > 4 (*a*=5)**
1	1	0	0	0	0
2	2	1	0	0	0
3	4	1	1	0	0
4	8	2	1	1	0
5	16	3	1	1	1
6	32	5	2	1	1
7	64	8	3	1	1
8	128	13	4	2	1
9	256	21	6	3	1
10	512	34	9	4	2
11	1,024	55	13	5	3
12	2,048	89	19	7	4
13	4,096	144	28	10	5
14	8,192	233	41	14	6
15	16,384	377	60	19	8
16	32,768	610	88	26	11
17	65,536	987	129	36	15
18	131,072	1,597	189	50	20
19	262,144	2,584	277	69	26
20	524,288	4,181	406	95	34

One of the many other places the ubiquitous Fibonacci series appears is as the sum of the off-diagonals of Pascal's triangle (see Figure 1 below).

Figure 1: Off-Diagonals of Pascal's Triangle Sum to Fibonacci Series



What is less well known is that the off-diagonals themselves are the number of integer compositions of n with k parts when a=2, that is, $c(n,k,2,\infty)$ (see Kimberling, 2002; in fact, the entire Triangle represents the number of compositions of n with k parts, as shown in Chinn and Heubach (2003)). However, Pascal's triangle not only allows one to count the number of compositions of n with a=2, but also provides the structure for generating these restricted integer compositions via composition "paths" that can be traced efficiently through the triangle. A mathematical construct that allows for the efficient identification of restricted integer compositions is important because, as a comparison of the growth rates of columns 2 vs. 3 through 6 in Table 1 shows, generating all integer compositions of n, and then deleting those with any parts=1 (or any parts<a), would be extremely computationally wasteful and expensive, not to mention runtime prohibitive for large n. So it is necessary to use an algorithm that efficiently and directly identifies the restricted integer compositions of n for specified a and b (and k). We begin with RICs_Base, the special "base case" of a=2 (and b=∞, or in practice, b=n), and easily modify RICs_Base to the more general algorithm "RICs" to accommodate any specified values of k, a and b simultaneously.

The RICs_Base Algorithm – the "Base Case" of a=2

For any positive integer $n \geq 2$, with a=2 (and b=n), $\max(k) = \lfloor n/2 \rfloor$ (more generally, max(k) = $\lfloor n/a \rfloor$, and when $b \leq n$, min(k) = $\lceil n/b \rceil$), and this is the number of off-diagonals for n = (row# + 2) in the "row-column," ordered binomial coefficient representation of Pascal's triangle in Figure 2 below (each $\binom{row\#}{column\#}$ is an "r-choose-c" binomial coefficient, where r=row# and c=column#, and $\binom{r}{c} = \frac{r!}{c!(r-c)!}$ = the value of the node in the triangle as shown in Figure 1).

Figure 2: (Ordered) Binomial Coefficient Representation of Pascal's Triangle

$$\begin{array}{ccccccccccccccccccc}
 & & & & & & & \binom{0}{0} & & & & & & & & & & & \\
 & & & & & & \binom{1}{0} & & \binom{1}{1} & & & & & & & & & & \\
 & & & & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & & & & & & & \\
 & & & & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & & & & & & & \\
 & & & \binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} & & & & & & & \\
 & & \binom{5}{0} & & \binom{5}{1} & & \binom{5}{2} & & \binom{5}{3} & & \mathbf{\binom{5}{4}} & & \binom{5}{5} & & & & & & \\
 & \binom{6}{0} & & \binom{6}{1} & & \binom{6}{2} & & \mathbf{\binom{6}{3}} & & \binom{6}{4} & & \binom{6}{5} & & \binom{6}{6} & & & & & \\
\binom{7}{0} & & \binom{7}{1} & & \mathbf{\binom{7}{2}} & & \binom{7}{3} & & \binom{7}{4} & & \binom{7}{5} & & \binom{7}{6} & & \binom{7}{7} & & & & \\
 & \binom{8}{0} & & \mathbf{\binom{8}{1}} & & \binom{8}{2} & & \binom{8}{3} & & \binom{8}{4} & & \binom{8}{5} & & \binom{8}{6} & & \binom{8}{7} & & \binom{8}{8} & \\
\mathbf{\binom{9}{0}} & & \binom{9}{1} & & \binom{9}{2} & & \binom{9}{3} & & \binom{9}{4} & & \binom{9}{5} & & \binom{9}{6} & & \binom{9}{7} & & \binom{9}{8} & & \binom{9}{9}
\end{array}$$

n = 11 = (9 + 2) = (row# + 2)

$\max(k) = \lfloor n/a \rfloor = \lfloor 11/2 \rfloor$ = 5 off-diagonals

With a=2 (and b=n), the RICs_Base algorithm loops max(k) times and generates all restricted integer compositions corresponding to each value of k on each loop. The movement of the algorithm through the triangle can be described as follows:

Description of RICs_Base:

Start at the leftmost off-diagonal Node of the Triangle corresponding to the specified n. For the first off-diagonal (where Column# = 0), simply output n. Otherwise proceed with the steps below for each off-diagonal.

1) Initialize and keep track of Row# and Column# for each call to the subroutine that moves from Node to Node through the Triangle.

   Let "Cell" be an array that will contain the composition parts.

   Let "Level" = current Row# minus initial off-diagonal Row#. Level tracks how many "left turns" deep into the triangle we move up and away from the off-diagonal. Level #1 represents the first slot containing the first part of each of the compositions; Level #2 represents the second slot, etc. There are max(k) Levels in RICs_Base.

   Initialize Level = 0. Keep track of Level for each call to the subroutine that moves from Node to Node through the Triangle.

2) Always move towards the top of the triangle

   Always turn to the Left Node first

   For every Left turn:
   a) Level ← Level + 1
   b) Cell[Level] ← 2
   c) Column# ← (Column# - 1)
   d) Row# ← (Row# – 1)

   Once away from the off-diagonal, recursively call the Node-to-Node subroutine with the Row#, Column#, and Level values associated with the Node to the Right of the current Node (that is, recursively turn to the Right Node)

   For every Right turn:
   e) Cell[Level] ← (Cell[Level]+1)
   f) Row# ← (Row# – 1)

3) If Column# = 1, then enter an output loop (looping from j = 1 to Row#) that fills in the last two Cell values (with the values of (j+1) and (row + 2 – j), respectively) and outputs the entire composition (i.e. all values of Cell that have been filled in)

Pseudo code implementing RICs_Base is presented below.

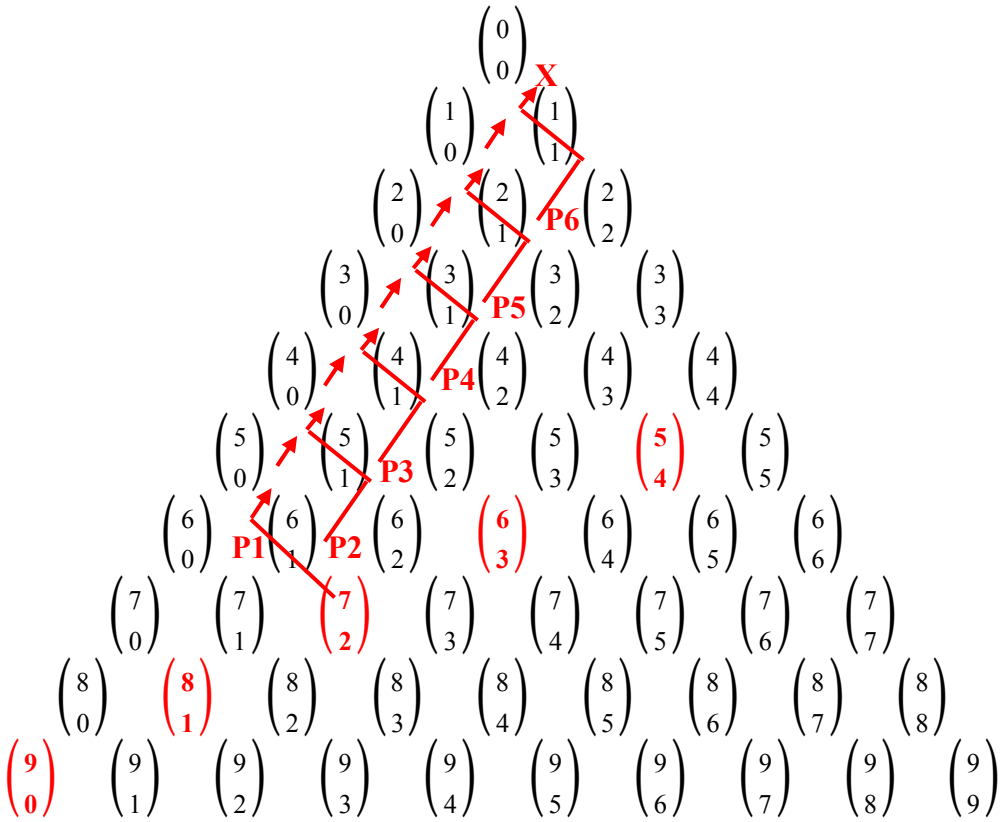
```
BEGIN RICs_Base(n)

Define Subroutine Node2Node(row, col, level)
    if col ≠ 0 then  do
        if Col = 1 then for j=1 to row do
            Cell[i-1] ← (j+1)
            Cell[i] ← (row + 2 - j)
            Print Cell[1] through Cell[i]
        endo
        else do
            Cell[Level+1] ← 2
            Node2Node (row ← row – 1, col ← col – 1, Level ← Level+1)
        endo
    endo
    else Print n
    if Level > 0 & row > 1 then  do
        Cell[Level] ← Cell[Level]+1
        Node2Node (row ← row – 1, col ← col, Level ← Level)
    endo
End Node2Node

for i = 1 to floor(n/2) do
    Node2Node (row ← (n – 1 – i), col ← (i – 1), level ← 0)
endo

END RICs_Base

CALL RICs_Base(n ← 11)
```

Actual computer code that implements the RICs_Base algorithm in SAS® can be found in Appendix A (SAS® is the largest privately owned software firm in the world, and with over 44,000 user sites, the SAS® platform is ubiquitous. Only the Base SAS® module is needed to run the code presented in this paper. C++ code, too, will be made available on the author's website at http://www.DataMineIt.com). An example of its output for n=11 can be seen in Table 2 below. The total number of restricted integer compositions is, as expected, $c\left(n=11, k=\text{unrestricted}, a=2, b=\infty\right)=F_{n-1}=F_{10}=55$. The corresponding paths RICs_Base traces through Pascal's triangle (only for k=3 to maintain visual clarity) are shown in Figure 3 below. Each path is numbered and the corresponding compositions are identified in Table 2.

Table 2: All Restricted Integer Compositions of $n = 11$ with parts > 1 (i.e. a=2, b=∞)
Output from RICs_Base (n=11)

Composition.#	Level1	Level2	Level3	Level4	Level5	
1	11					
2	2	9				
3	3	8				
4	4	7				
5	5	6				
6	6	5				
7	7	4				
8	8	3				
9	9	2				
10	**2**	**2**	**7**			
11	**2**	**3**	**6**			
12	**2**	**4**	**5**			→ Path 1
13	**2**	**5**	**4**			
14	**2**	**6**	**3**			
15	**2**	**7**	**2**			
16	3	2	6			
17	3	3	5			
18	3	4	4			→ Path 2
19	3	5	3			
20	3	6	2			
21	**4**	**2**	**5**			
22	**4**	**3**	**4**			→ Path 3
23	**4**	**4**	**3**			
24	**4**	**5**	**2**			
25	5	2	4			
26	5	3	3			→ Path 4
27	5	4	2			
28	**6**	**2**	**3**			→ Path 5
29	**6**	**3**	**2**			
30	7	2	2			→ Path 6
31	2	2	2	5		
32	2	2	3	4		
33	2	2	4	3		
34	2	2	5	2		
35	2	3	2	4		
36	2	3	3	3		
37	2	3	4	2		
38	2	4	2	3		
39	2	4	3	2		
40	2	5	2	2		
41	3	2	2	4		
42	3	2	3	3		
43	3	2	4	2		
44	3	3	2	3		
45	3	3	3	2		
46	3	4	2	2		
47	4	2	2	3		
48	4	2	3	2		
49	4	3	2	2		
50	5	2	2	2		
51	2	2	2	2	3	
52	2	2	2	3	2	
53	2	2	3	2	2	
54	2	3	2	2	2	
55	3	2	2	2	2	

Figure 3: Composition Paths Traced by RICs_Base(n=11, k=3)



The RICs_Base algorithm is efficient in that every restricted integer composition for n (with a=2 and b=∞) is identified once and only once by a unique path in the Triangle, and no extra steps are necessary or taken. This also is (almost always) true of the RICs algorithm, which generates restricted integer compositions for any a, b, and k generally.

But first, note that for the case of a=1 (and b=∞), that is, the completely unrestricted case, the RICs algorithm still can be used if the entire row of row# = (n – 1), rather than the off-diagonal of row# = (n – 2), is traversed in the same manner. This is easily incorporated into RICs below to increase its range of application to any value of a – that is, to generate both restricted and unrestricted integer compositions (of course, where $1 \le a \le b \le n$ (b>n is ignored) and $\lceil n/b \rceil \le k \le \lfloor n/a \rfloor$).

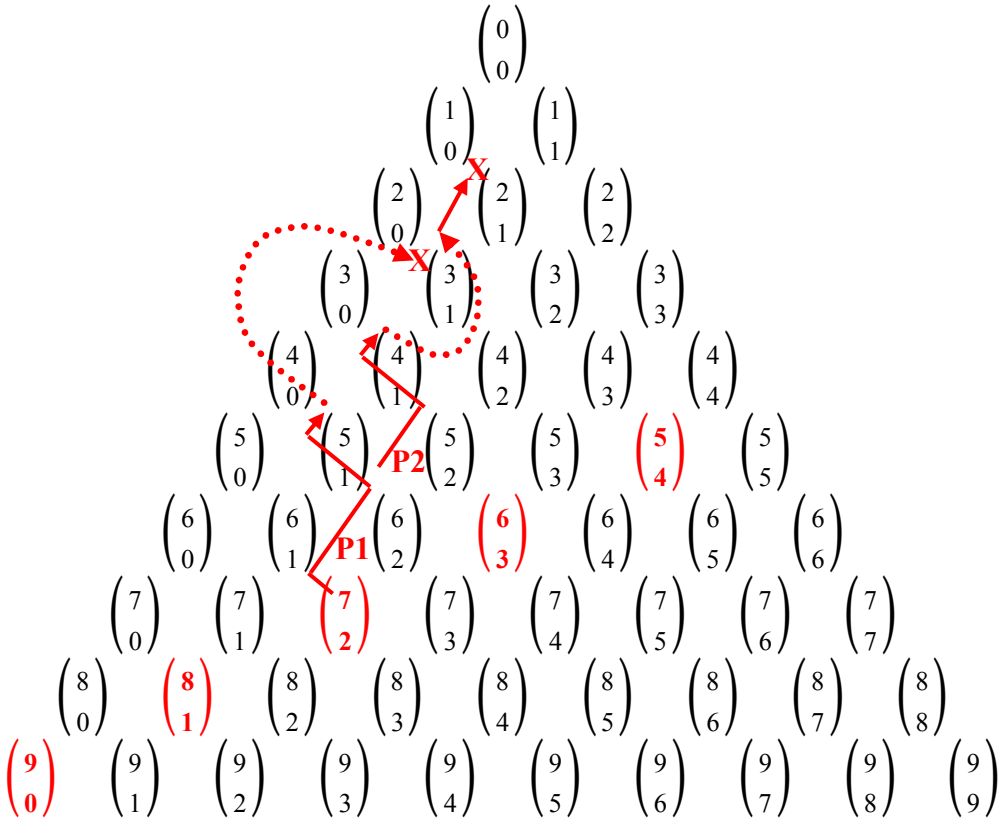
The RICs Algorithm – the General Case of any *a*, *b*, and *k*

The "Base Case" RICs algorithm for a=2 and b=∞ (RICs_Base above) is easily generalized for any values of a, b, and k with only two additional rules. RICs uses the same structure of Pascal's triangle, loops on each of its off-diagonals, and follows the same paths except that it only continues up any specific path if two conditions are satisfied: i) $a \le$ Cell[Level] $\le b$, and ii) $a \le$ average amount left $\le b$, where "average amount left" = (n – the sum of the parts already assigned in Cell[]) / (the number of parts yet to be assigned in that path). The min$k$ and max$k$ restrictions are trivially satisfied because these values simply become the low and high values, respectively, of the program loop on the Node2Node subroutine (when Rule ii) is applied to the off-diagonal node, it prevents the algorithm from running on a user-misspecified value of $k > \lfloor n/a \rfloor$, which is not possible; misspecified values of $k < \lceil n/b \rceil$ simply are ignored). Table 3 shows the example of all the restricted integer compositions of n=11, min$k$=2, max$k$=5, a=2, and b=4, and Figure 4 shows the corresponding paths RICs traces through the Triangle (only for k=3 to maintain visual clarity).

Table 3: All Restricted Integer Compositions of n = 11 with min$k$=2, max$k$=5, a=2, b=4

Composition.#	Level1	Level2	Level3	Level4	Level5	
1	3	4	4			→ Path 1
2	4	3	4			
3	4	4	3			→ Path 2
4	2	2	3	4		
5	2	2	4	3		
6	2	3	2	4		
7	2	3	3	3		
8	2	3	4	2		
9	2	4	2	3		
10	2	4	3	2		
11	3	2	2	4		
12	3	2	3	3		
13	3	2	4	2		
14	3	3	2	3		
15	3	3	3	2		
16	3	4	2	2		
17	4	2	2	3		
18	4	2	3	2		
19	4	3	2	2		
20	2	2	2	2	3	
21	2	2	2	3	2	
22	2	2	3	2	2	
23	2	3	2	2	2	
24	3	2	2	2	2	

Figure 4: Composition Paths Traced by RICs(n=11, k=3, a=2, b=4)



RICs remains efficient in the same sense that RICs_Base is efficient in that it still uniquely identifies every valid restricted integer composition once and only once, and takes no unnecessary "exploratory" left turns. In other words, it does not proceed up any left-path to the next Level unless it contains at least one restricted integer composition that satisfies the specified values of min$k$, max$k$, a, and b (Rule ii) ensures this). However, whenever Rule i) is not satisfied immediately upon turning right (which is fairly rare), RICs must make "exploratory" right turns along the same Level (from Cell[Level]=a to either b, or a+[Row# – Col#] – the end of the Level – whichever comes first) to know whether it needs to make any more left turns further up into the Triangle (an example of this is

RICs(n=12, k=3, a=3, b=4) – upon RICs' first right turn, Cell[Level]=3, which is not valid, so another right turn must be made to check to see whether Cell[Level]+1=4 is valid, and it is, leading to the composition 4 4 4). SAS® code implementing the general case of RICs can be found in Appendix B, and pseudo code is presented below.

```
BEGIN RICs(n, mink, maxk, a, b)

Define Subroutine Node2Node(row, col, level, cum_sum_parts)
    if col ≠ 0 then do
        if col = 1 then for j = max[a, (n – cum_sum_parts – b)]   to   min[b, (n – cum_sum_parts – a)] do
            Cell[i – 1] ← j
            Cell[i] ← (n – cum_sum_parts – j)
            Print Cell[1] through Cell[i]
        endo
        else do
            Cell[Level+1] ← a
            cum_sum_parts_temp ← cum_sum_parts + a
            if ( a≤ [(n – cum_sum_parts_temp)/(i – Level – 1)] ≤b   &  Cell[Level+1] ≤b ) then
                Node2Node (row ← row–a+1, col ← col–1, Level ← Level+1, cum_sum_parts ← cum_sum_parts_temp)
            else for q=1 to min[(b – a), (row – a) – (col – 1)] do
                Cell[Level+1] ← Cell[Level+1] + 1
                cum_sum_parts_temp ← cum_sum_parts_temp + 1
                if ( a≤ [(n – cum_sum_parts_temp)/(i – Level – 1)] ≤b   &   Cell[Level+1] ≤b ) then do
                    q2  ← q
                    q  ← min[(b – a), (row – a) – (col – 1)]
                    Node2Node (row←row–a+1–q2, col←col–1, Level←Level+1, cum_sum_parts←cum_sum_parts_temp)
                endo
            endo
        endo
    endo
    else Print n
    if Level > 0 & row > 1 then  do
        Cell[Level] ← Cell[Level]+1
        cum_sum_parts ← cum_sum_parts + 1
        if Cell[Level]<a then do
            cum_sum_parts ← cum_sum_parts + (a – Cell[Level])
            Cell[Level] ← a
            row ← row  –  (a – Cell[Level])
        endo
        toploop ← min[(b – Cell[Level]), (row – 1 – col)]
        if ( a≤ [(n – cum_sum_parts)/(i – Level)] ≤b   &   Cell[Level] ≤b ) then
            Node2Node (row←row – 1, col←col, Level← Level, cum_sum_parts ← cum_sum_parts)
        else for p=1 to toploop do
            Cell[Level] ← Cell[Level]+1
            cum_sum_parts ← cum_sum_parts + 1
            if ( a≤ [(n – cum_sum_parts)/(i – Level)] ≤b   &   Cell[Level] ≤b ) then do
                p2  ← p
                p  ← toploop
                Node2Node (row←row – p2, col←col, Level← Level, cum_sum_parts ← cum_sum_parts)
            endo
        endo
    endo
End Node2Node

rowdec ← 0
for i = mink to maxk do
    if a≠1 then rowdec ← i
    if (a≤ (n/i) ≤b) then Node2Node (row ← n–1–rowdec, col ← i–1, level ← 0, cum_sum_parts ← 0)
endo

END RICs

CALL RICs(n←11, mink←2, maxk←5, a←2, b←4)
```

Note that Rule ii) is applied at the off-diagonal nodes as well, in the main loop on the Node2Node subroutine, so that if no restricted integer compositions of, say, k=2 exist for n=11, a=2, and b=4 (which is true), then even if min$k$=2 is specified, Node2Node is not called on the second off-diagonal, which corresponds to k=2.

Also note that the algorithm assigns a value of Cell[Level] = a (instead of 2 as in RICs_Base) for every left turn so that RICs immediately "jumps" to the node to the right along the same Level where Cell[Level] = a, while incrementing Cum_Sum_Parts (the sum of the parts already assigned to Cell[]) accordingly. This efficiently eliminates unnecessary checks of Rules i) and ii) at every Node on a Level. But along a given Level, RICs still must explicitly check whether Rule ii) is satisfied for every value of Cell[Level] = a through either b, or a+(Row# – Col#) – the end of the Level – whichever comes first. In this sense, whenever Rule i) is not satisfied immediately upon turning right (or after a right "jump"), which is rare, RICs must make additional "exploratory" right turns on the same Level to know whether it needs to make any further left turns deeper up into the Triangle.

A final note on the efficiency of RICs: for this general case valid for any values for a and b (and k), the endpoints of the output loop in RICs are defined exactly by:

Low value = max[a, (n – sum of the parts already assigned – b)]
High value = min[b, (n – sum of the parts already assigned – a)]

so no unnecessary looping is performed in the output loop.

Counting the Number of Restricted Integer Compositions – the General Case

Although for b=∞ (in practice, b=n), both $c(n,k,a,\infty)=c(n-ka,k,0,\infty)=\binom{n-ka+k-1}{k-1}$ and $c(n,a,\infty)=\sum_{k=1}^{\lfloor n/a\rfloor}\binom{n-ka+k-1}{k-1}$ are well known, to the best of this author's knowledge (and that of C. Kimberling via email correspondence, 07/24/08), no closed-form solutions exist for $c(n,k,a,b)$ and $c(n,a,b)$ generally, that is, for any a, b, and k, (of course, where $1 \le a \le b \le n$ (b>n is ignored) and $\lceil n/b\rceil \le k \le \lfloor n/a\rfloor$) and they remain open problems. Some related results include Heubach and Mansour's (2004) presentation of the generating function for the number of compositions of n with k parts in the set A. They use the generating function to solve for some examples of specifically defined A, but do not solve for A generally as defined by an arbitrary range (that is, A including all positive integers x such that $a \le x \le b$, which is what is needed here). Chinn and Heubach (2003) present recursions for counting the number of compositions with k parts that exclude a particular part x, but it is not obvious how this can be utilized to solve the complement of the general case of $a \le x \le b$ by excluding entire sets of parts where $x < a$ and $x > b$ (i.e. by solving for A^c, where $A^c \cup A = \mathbb{N}$). Conversely, Knopfmacher and Mays (1996) identified a convenient recursive relationship for counting the number of compositions of n that contain at least one part = x, but again, some of those compositions counted by $2^{(n-1)} - c(n \mid \text{one or more parts} = x)$ will include compositions with, for example, one or more parts = x+1, and if (x+1)<b, we cannot double count these compositions when subtracting both counts from $2^{(n-1)}$ in $2^{(n-1)} - c(n \mid \text{one or more parts} = x) - c(n \mid \text{one or more parts} = (x+1))$.

All of these results come tantalizingly close to a general solution, but the results of RICs in Table 4 below reveals a very simple recursion as a solution to $c(n,a,b)$ (zeros are excluded from all Tables of counts to enhance visual clarity):

For $n \le b$, b has no effect on the outcome, so $c(n,a,b) = c(n,a,\infty) = \sum_{k=1}^{\lfloor n/a \rfloor} \binom{n-ka+k-1}{k-1}$ (1)

but for $n > b$, $c(n,a,b) = \sum_{i=(n-b)}^{(n-a)} c(i,a,b)$

(1) can be combined into a single formula:

$$c(n,a,b) \;=\; I(n \le b) + \sum_{i=\max[1,(n-b)]}^{(n-a)} c(i,a,b) \qquad (2)$$

where the indicator function $I( ) = 1$ if $n \le b$, and $I( ) = 0$ otherwise, and $1 \le a \le b \le n$ ($b > n$ is ignored).

Table 4: Counts of Restricted Integer Compositions for Specified Values of n, a, and b: $c(n,a,b)$

	a=	2	3	4	2	2	4	4	4	4	4	4	4	4
n	b=	3	4	5	4	5	5	6	7	8	9	10	11	12
1														
2		**1**			**1**	**1**								
3		**1**	**1**		**1**	**1**								
4		1	**1**	**1**	**2**	**2**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
5		2		**1**	2	**3**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
6		2	1		4	4		**1**	**1**	**1**	**1**	**1**	**1**	**1**
7		3	2		5	7			**1**	**1**	**1**	**1**	**1**	**1**
8		4	1	1	8	10	1	1	1	**2**	**2**	**2**	**2**	**2**
9		5	1	2	11	16	2	2	2	2	**3**	**3**	**3**	**3**
10		7	3	1	17	24	1	3	3	3	3	**4**	**4**	**4**
11		9	3		24	37		2	4	4	4	4	**5**	**5**
12		12	2	1	36	57	1	2	4	6	6	6	6	**7**
13		16	4	3	52	87	3	3	5	7	9	9	9	9
14		21	6	3	77	134	3	6	7	9	11	13	13	13
15		28	5	1	112	205	1	7	10	12	14	16	18	18
16		37	6	1	165	315	1	7	13	17	19	21	23	25
17		49	10	4	241	483	4	7	16	22	27	29	31	33
18		65	11	6	354	741	6	11	20	29	36	41	43	45
19		86	11	4	518	1,137	4	16	26	38	47	55	60	62
20		114	16	2	760	1,744	2	20	35	51	63	73	81	86
21		151	21	5	1,113	2,676	5	21	46	67	86	98	109	117
22		200	22	10	1,632	4,105	10	25	59	89	116	135	148	159
23		265	27	10	2,391	6,298	10	34	75	118	154	184	203	217
24		351	37	6	3,505	9,662	6	47	97	157	206	248	278	298
25		465	43	7	5,136	**14,823**	7	57	127	207	278	333	378	408

We can see a similar recursion for $c(n,k,a,b)$ in Tables 5 and 6 below, such that:

$$c(n,k,a,b) = \sum_{i=\max\left[1,(n-b)\right]}^{(n-a)} c(i,k-1,a,b) \tag{3}$$

where $c(i,0,a,b) = 1$, $1 \le a \le b \le n$ ($b > n$ is ignored), and $\lceil n/b \rceil \le k \le \lfloor n/a \rfloor$.

Table 5: Counts of Restricted Integer Compositions for Specified Values of n, k, a=2, b=5: $c(n,k,a,b)$

	a=	2	2	2	2	2	2	2	2	2	2	2	2
	b=	5	5	5	5	5	5	5	5	5	5	5	5
n	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2		1											
3		1											
4		1	1										
5		1	2										
6			3	1									
7			4	3									
8			3	6	1								
9			2	10	4								
10			1	12	10	1							
11				12	20	5							
12				10	31	15	1						
13				6	40	35	6						
14				3	44	65	21	1					
15				1	40	101	56	7					
16					31	135	120	28	1				
17					20	155	216	84	8				
18					10	155	336	203	36	1			
19					4	135	456	413	120	9			
20					1	101	546	728	322	45	1		
21						65	580	1,128	728	165	10		
22						35	546	1,554	1,428	486	55	1	
23						15	456	1,918	2,472	1,206	220	11	
24						5	336	2,128	3,823	2,598	705	66	1
25						1	216	2,128	**5,328**	4,950	1,902	286	12

So a solution to the problem of counting the number of compositions simultaneously restricted both in the number of parts allowed, and the values of those parts, i.e. $c(n,\min k \le k \le \max k,a,b)$, is simply

$$c(n,\min k \le k \le \max k,a,b) = \sum_{k=\min k}^{\max k} \left[\sum_{i=\max\left[1,(n-b)\right]}^{(n-a)} c(i,k-1,a,b) \right] \tag{4}$$

where $c(i,0,a,b) = 1$, $1 \le a \le b \le n$ ($b > n$ is ignored), and $\lceil n/b \rceil \le k \le \lfloor n/a \rfloor$ (Mathematica® code for generating all Tables of counts contained herein is provided in Appendix D).

Formulae (2), (3), and (4) mirror the analogous solutions for counting doubly-restricted integer partitions presented later in the paper. Although their recursive nature makes these formulae less convenient than, say, a simple combinatoric equation or sum, they still provide closed form solutions to problems which had none before, and their calculation is not onerous. Derivation of straightforward combinatoric equivalents of each is the topic of continuing research.

Table 6: Counts of Restricted Integer Compositions for Specified Values of n, k, a=3, b=7: $c(n,k,a,b)$

	a=	3	3	3	3	3	3
	b=	7	7	7	7	7	7
n	k=	1	2	3	4	5	6
1							
2							
3		1					
4		1					
5		1					
6		1	1				
7		1	2				
8			3				
9			4	1			
10			5	3			
11			4	6			
12			3	10	1		
13			2	15	4		
14			1	18	10		
15				19	20	1	
16				18	35	5	
17				15	52	15	
18				10	68	35	1
19				6	80	70	6
20				3	85	121	21
21				1	80	185	56
22					68	255	126
23					52	320	246
24					35	365	426
25					20	381	**666**

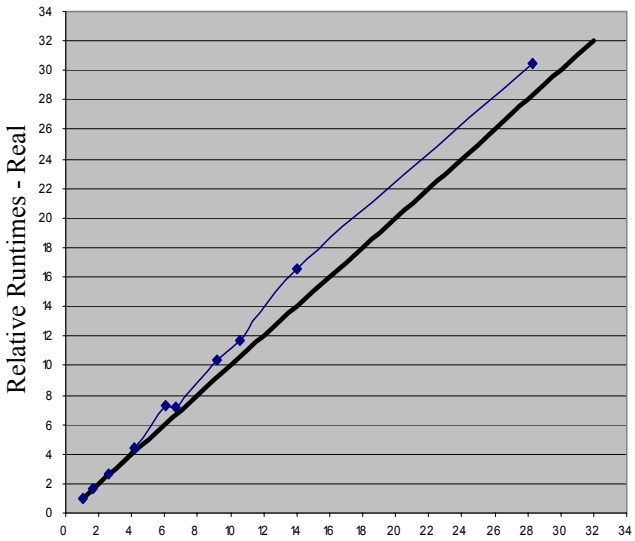
Time Complexity of RICs

If k is not explicitly bounded by the RICs user then an estimate of the time complexity of RICs is $\mathrm{O}\left(\sum_{k=\lceil n/b \rceil}^{\lfloor n/a \rfloor}\left(k \bullet \sum_{i=(n-b)}^{(n-a)} c(i,k-1,a,b)\right)\right)$ to calculate $c(n,a,b)$ compositions (if k is explicitly bounded by the RICs user then for $c(n,\min k \le k \le \max k,a,b)$ this estimate is $\mathrm{O}\left(\sum_{k=\min k}^{\max k}\left(k \bullet \sum_{i=(n-b)}^{(n-a)} c(i,k-1,a,b)\right)\right)$). This estimate assumes that $\mathrm{O}(\mathrm{RICs})$ is a function of only two parameters: 1) the number of compositions that exist given the specified values of a and b (and min$k$ and max$k$), and 2) the number of parts in each of these compositions. Of

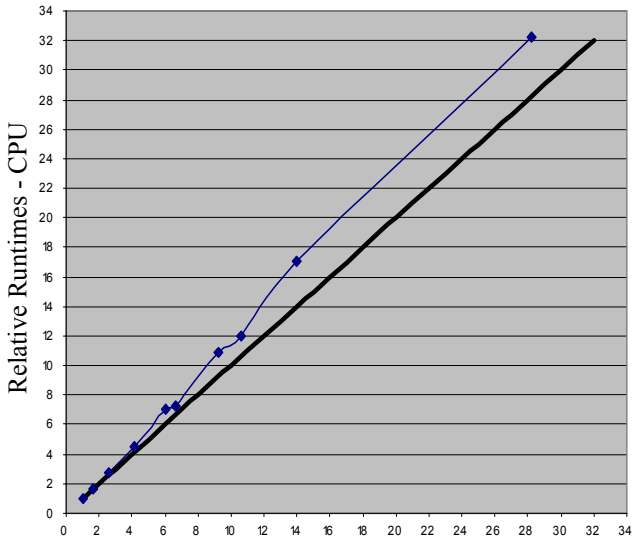
course, the algorithm does need to enforce Rules i) and ii) before calling the Node2Node subroutine, and it also occasionally must make exploratory "right turns" (on a given Level) before calling Node2Node to decide whether to turn left, deeper up into the Triangle, so on the one hand this may be an underestimate of the magnitude of $\mathrm{O}(\mathrm{RICs})$. On the other hand, this may be an overestimate in that the number of times Node2Node is called for a given composition is rarely as large as $k - 1$, because most "paths," up to the output loop, contain many compositions, with only the last two parts changing via the output loop; in other words, only when there exists a single composition for a given k will it be necessary to call Node2Node the full $k - 1$ times, as assumed by this estimate of $\mathrm{O}(\mathrm{RICs})$ (for example, compare Path 1 vs. Path 6 in Table 2). Graphs 1 and 2 below plot the relative runtime (real and cpu) for the algorithm by the relative number of steps as estimated by $\mathrm{O}\left(\sum_{k=\lceil n/b\rceil}^{\lfloor n/a\rfloor}\left(k \bullet \sum_{i=(n-b)}^{(n-a)} c(i,k-1,a,b)\right)\right)$, and a relationship that is very close to linear with unit slope is evident for both real and cpu runtimes. For cpu time, the slight increase in slope for larger values of $c(n,a,b)$ (here the largest is $c(n,a,b)=c(n=25,a=2,b=n)=46{,}386$) indicates that this approximation may still be underestimating RICs' time complexity slightly, at least for moderate to large values of $c(n,a,b)$. But for most practical applications, especially those focused on real runtimes, this appears to be a good approximation of $\mathrm{O}(\mathrm{RICs})$. I compare $\mathrm{O}(\mathrm{RICs})$ to composition growth rates in Table 7 below. Note in column 5 that $\mathrm{O}(\mathrm{RICs})/c(n,a,b)$ is simply the weighted average number of parts per composition, or average k for a given $c(n,a,b)$. So to the extent that this overall estimate of $\mathrm{O}(\mathrm{RICs})$ is accurate as shown in Graphs 1 and 2, the asymptotic time complexity of RICs *per composition* is simply $\sim \mathrm{O}(k)$.

Relative Runtimes (Real and CPU) by Relative Approximation of $\mathrm{O}(\mathrm{RICs})$

Graph 1



Graph 2



Relative Values of Calculated Estimate of $\mathrm{O}(\mathrm{RICs})$

Table 7: Counts of (Un)restricted Integer Compositions vs. $O(\text{RICs}(n, k = unrestricted, a = 2, b = 5))$

n	**Unrestricted Integer Compositions** $c(n) = 2^{(n-1)}$	**Restricted Integer Compositions** $c(n,a,b)$	$O(\text{RICs})$	$O(\text{RICs})/c(n,a,b)$ = average *k*	$O(\text{RICs})/2^{(n-1)}$
1	1	0	0		
2	2	1	1	1.000	0.500
3	4	1	1	1.000	0.250
4	8	2	3	1.500	0.375
5	16	3	5	1.667	0.313
6	32	4	9	2.250	0.281
7	64	7	17	2.429	0.266
8	128	10	28	2.800	0.219
9	256	16	50	3.125	0.195
10	512	24	83	3.458	0.162
11	1,024	37	141	3.811	0.138
12	2,048	57	235	4.123	0.115
13	4,096	87	389	4.471	0.095
14	8,192	134	643	4.799	0.078
15	16,384	205	1,053	5.137	0.064
16	32,768	315	1,723	5.470	0.053
17	65,536	483	2,803	5.803	0.043
18	131,072	741	4,549	6.139	0.035
19	262,144	1,137	7,359	6.472	0.028
20	524,288	1,744	11,872	6.807	0.023
21	1,048,576	2,676	19,110	7.141	0.018
22	2,097,152	4,105	30,688	7.476	0.015
23	4,194,304	6,298	49,188	7.810	0.012
24	8,388,608	9,662	78,691	8.144	0.009
25	16,777,216	14,823	125,681	8.479	0.007

Potential Disadvantages of RICs

RICs obviously is not loopless, a characteristic which would allow it to achieve theoretically minimal time complexity. Nor is it iterative, but rather, it is recursive, a quality some consider a disadvantage relative to iterative algorithms, mostly on the basis of ease of manipulation and/or understanding (see Strojmenovic, 2008). However, some believe the opposite in many cases, since some computers actually provide a speed premium to recursive algorithms. Also, given the binomial coefficient representation of Pascal's triangle that is the foundational structure of the algorithm, RICs should lend itself to an iterative implementation, with iteration based on the row numbers and column numbers of the Triangle. To avoid recursion, the row numbers and column numbers would have to be updated dynamically. Completing a non-recursive version of RICs is the topic of continuing research.

A potentially more valid criticism of RICs (depending on the reason for its usage) is that the order of the restricted integer compositions it generates is neither lexicographic nor antilexicographic. The implementation of RICs in Appendix B is lexicographic for each value of *k*, and it easily can be modified to be antilexicographic for each value of *k*, but it is neither across all values of *k*, that is, it is not (anti)lexicographic across all of the restricted integer compositions generated. This may not matter to the user of RICs depending on his or her objectives, but if one of these two common orderings is required, then an additional sort of the compositions generated by RICs is required.

Advantages of RICs

RICs is flexible, allowing any combination of ranges of values for a) the number of parts allowed, and b) the values of those parts, to be specified simultaneously. No other algorithm in the literature known to this author can make this claim. RICs also is reasonably fast, mainly because it uniquely identifies each valid restricted integer composition once and only once, and with minor exceptions, takes no unnecessary steps "looking for" these valid compositions. On modern computers, even for moderately large n, RICs runs in only seconds. For all practical purposes, its time complexity, based on empirical observation, appears to be approximately the product of the number of compositions generated, and the number of parts in these compositions. Per composition, this approximate time complexity is simply $\sim O(k)$. Finally, RICs is based on very fundamental and well understood mathematical constructs, namely, Pascal's triangle and its (Fibonacci series) off-diagonals, the fundamental properties of which lend a generality to the algorithm that allows it, with only minor modifications, to generate restricted integer partitions as well, as shown below.

From RICs to RIPs: Generating Restricted Integer Partitions

While several algorithms exist to count the number of restricted integer partitions under very general conditions (Beyer and Swinehart, 1973, Sanchis and Squire, 1996, Uppuluri and Carpenter, 2006, and White, 1970a), none known to this author actually generate restricted integer partitions under the two most common restrictions, applied concurrently. Yet starting with RICs, only two minor modifications are required to create the "RIPs" algorithm to efficiently generate restricted integer partitions under the same two restrictions: upper and lower bounds on the number of parts allowed, and upper and lower bounds on the values of those parts. The two additional rules are: a) for each right turn on a Level of the triangle, increment the value of "a," the minimum part value, by one; and b) in the output loop, decrease the high loop value by half the distance to the low value: high ← high – ceiling((high – low)/2). The former restriction prevents a different ordering of the same partition from being generated at the lower Levels, and the latter restriction prevents the same from occurring at the two highest Levels whose Cell[] values are assigned in the output loop. Pseudo-code that implements either RICs or RIPs (i.e. the "RICs_RIPs" algorithm), depending on a user-specified parameter, is provided below, with differences from RICs highlighted in red. SAS® code for RICs_RIPs is provided in Appendix C (RICs_RIPs will be made available on the author's website in C++ code as well). Its time complexity (see Graphs 3 and 4 below) and (dis)advantages are the same as those of RICs above, and Table 8 below can be compared to Table 3 above, as it presents the output from RIPs instead of RICs under the same specified parameter values of n, min$k$, max$k$, a and b.

While facially the similarities between integer partitions and integer compositions are obvious, algorithms presented in the literature to generate these two combinatorial objects, whether restricted or unrestricted, often are quite dissimilar. RICs_RIPs is a unifying exception to this. Basing this algorithm on such a fundamental mathematical construct as Pascal's triangle provides an important *algorithmic* link between these two combinatorial objects not previously identified in the literature, and one that warrants further study. This link is readily apparent when comparing (2), (3), and (4) above to the formulae presented below, (5), (6), and (7), for counting doubly-restricted integer partitions.

Table 8: All Restricted Integer Partitions of n = 11 with min$k$=2, max$k$=5, a=2, b=4

Partition #	Level1	Level2	Level3	Level4	Level5
1	3	4	4		
2	2	2	3	4	
3	2	3	3	3	
4	2	2	2	2	3

```
BEGIN RICs_RIPs(comp_part, n, mink, maxk, a, b)

Define Subroutine Node2Node(row, col, level, cum_sum_parts, a)
    if col ≠ 0 then do
        high ← min[b, (n – cum_sum_parts – a)]
        if comp_part = part then high ← high – floor((high – low + 1)/2)
        if col = 1 then for j = max[a, (n – cum_sum_parts – b)]  to  high  do
            Cell[i – 1] ← j
            Cell[i] ← (n – cum_sum_parts – j)
            Print Cell[1] through Cell[i]
        endo
        else do
            Cell[Level+1] ← a
            cum_sum_parts_temp ← cum_sum_parts + a
            if ( a≤ [(n – cum_sum_parts_temp)/(i – Level – 1)] ≤b  &  Cell[Level+1] ≤b ) then
                Node2Node (row ← row–a+1, col ← col–1, Level ← Level+1, cum_sum_parts ← cum_sum_parts_temp, a ← a)
            else for q=1 to min[(b – a), (row – a) – (col – 1)] do
                Cell[Level+1] ← Cell[Level+1] + 1
                if comp_part = part then a ← a + 1
                cum_sum_parts_temp ← cum_sum_parts_temp + 1
                if ( a≤ [(n – cum_sum_parts_temp)/(i – Level – 1)] ≤b  &  Cell[Level+1] ≤b ) then do
                    q2  ← q
                    q  ← min[(b – a), (row – a) – (col – 1)]
                    Node2Node (row←row–a+1–q2, col←col–1, Level←Level+1, cum_sum_parts←cum_sum_parts_temp, a ← a)
                endo
            endo
        endo
    endo
    else Print n
    if Level > 0 & row > 1 then  do
        Cell[Level] ← Cell[Level]+1
        if comp_part = part then a ← a + 1
        cum_sum_parts ← cum_sum_parts + 1
        if Cell[Level]<a then do
            cum_sum_parts ← cum_sum_parts + (a – Cell[Level])
            Cell[Level] ← a
            row ← row  –  (a – Cell[Level])
        endo
        toploop ← min[(b – Cell[Level]), (row – 1 – col)]
        if ( a≤ [(n – cum_sum_parts)/(i – Level)] ≤b  &  Cell[Level] ≤b ) then
            Node2Node (row←row – 1, col←col, Level← Level, cum_sum_parts ← cum_sum_parts, a ← a)
        else for p=1 to toploop do
            Cell[Level] ← Cell[Level]+1
            if comp_part = part then a ← a + 1
            cum_sum_parts ← cum_sum_parts + 1
            if ( a≤ [(n – cum_sum_parts)/(i – Level)] ≤b  &  Cell[Level] ≤b ) then do
                p2  ← p
                p  ← toploop
                Node2Node (row←row – p2, col←col, Level← Level, cum_sum_parts ← cum_sum_parts, a ← a)
            endo
        endo
    endo
End Node2Node

rowdec ← 0
for i = mink to maxk do
    if a≠1 then rowdec ← i
    if (a≤ (n/i) ≤b) then Node2Node (row ← n–1–rowdec, col ← i–1, level ← 0, cum_sum_parts ← 0, a ← a)
endo

END RICs_RIPs

CALL RICs_RIPs(comp_part←part, n←11, mink←2, maxk←5, a←2, b←4)
```

Counting the Number of Restricted Integer Partitions – the General Case

As previously mentioned, several algorithms exist to count the number of restricted integer partitions under very general conditions, but none present the specific formula for counting the number of doubly-restricted integer partitions with both upper and lower bounds. Andrews (1998) presents the general result from which (5), (6), and (7) below can be derived, but for these specific formulae, Ruskey (2003) comes closest when presenting a formula (4) for counting the number of doubly restricted integer partitions with only an upper bound on the part values (and exactly k parts):

$$p(n,k,b) = \sum_{i=\max\left(1,\left\lceil \frac{n-b}{k-1} \right\rceil\right)}^{\min(b,n-b-k+2)} p(n-b,k-1,i) \tag{4}$$

(4) is similar to the recursion shown below in (6) that places both upper and lower bounds on part values for a specific k, but first the case with no restrictions on the number of parts, k, is shown in (5):

$$p(n,a,b) = I(n \le b) + \sum_{i=\max\left[1,(n-b)\right]}^{(n-a)} p(i,n-i,b) \qquad \text{where } 1 \le a \le b \le n \ \ (b > n \text{ is ignored}). \tag{5}$$

Table 9 contains results based on (5). Note that when (5) is compared to (2), we can see that the term for the minimum part value ("a") in the summation is decremented for each value i of the n parameter, as is done explicitly in the RICs_RIPs generation algorithm. Otherwise, the two formulae are identical, which is an intriguing finding. For a specific number of parts, k, the formula for partitions below (6) shares the same similarity with the analogous formula for compositions, (3):

$$p(n,k,a,b) = \sum_{i=\max\left[1,(n-b)\right]}^{(n-a)} p(i,k-1,n-i,b) \tag{6}$$

where $p(i,0,a,b) = 1$, $1 \le a \le b \le n$ ($b > n$ is ignored), and $\lceil n/b \rceil \le k \le \lfloor n/a \rfloor$.

Tables 10a-10i below show results based on (6). Finally, (7) shows the doubly-restricted formula for partitions that is analogous to (4) for compositions:

$$p(n,\min k \le k \le \max k,a,b) = \sum_{k=\min k}^{\max k} \left[\sum_{i=\max\left[1,(n-b)\right]}^{(n-a)} p(i,k-1,n-i,b) \right] \tag{7}$$

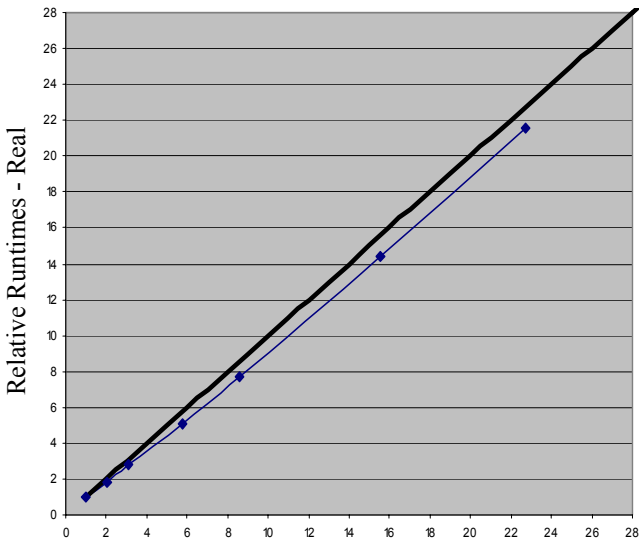
where $p(i,0,a,b) = 1$, $1 \le a \le b \le n$ ($b > n$ is ignored), and $\lceil n/b \rceil \le k \le \lfloor n/a \rfloor$.

Note that the columns of Tables 9 and 10a-10i will be recognized as the coefficients of the Gaussian polynomials, or the q-analogs of the binomial coefficient (see Koepf, 1998). This is not surprising given that the ordered binomial coefficients of Pascal's triangle form the basis of the algorithm, and this directly leads us to see (5), (6), and (7) as specific results of the general solution for counting restricted integer partitions presented in Andrews (1998). However, the specific, easily interpretable (and programmable) *form* of equations (5), (6), and (7) has not been presented previously, which is probably why their important link to the completely original, analogous solutions of (2), (3), and (4) for compositions have been missed until now. Consequently, they are worth presenting

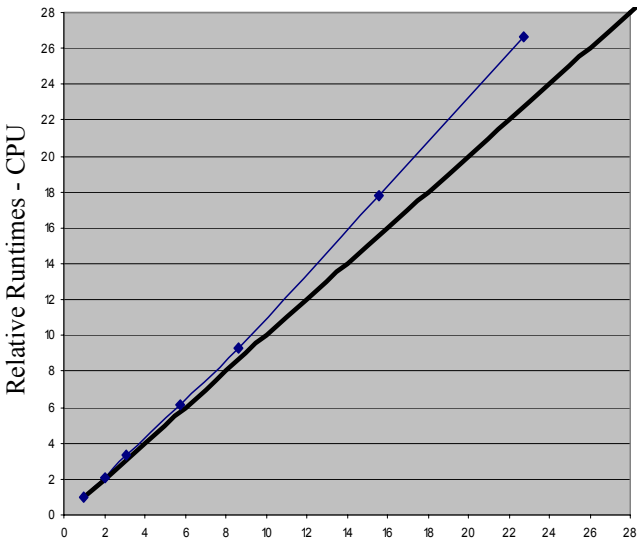
in this first paper to unify the approach to generating (un)restricted integer compositions and (un)restricted integer partitions.

Relative Runtimes (Real and CPU) by Relative Approximation of $O(\text{RIPs})$

Graph 3



Graph 4



Relative Values of Calculated Estimate of $O(\text{RIPs})$

Table 9: Counts of Restricted Integer Partitions for Specified Values of *n*, *a*, and *b*:

	a=	10	9	8	7	6	5	4	3	2
n	*b*=	10	10	10	10	10	10	10	10	10
1										
2										**1**
3									**1**	**1**
4								**1**	**1**	**2**
5							**1**	**1**	**1**	**2**
6						**1**	**1**	**1**	**2**	**4**
7					**1**	**1**	**1**	**1**	**2**	**4**
8				**1**	**1**	**1**	**1**	**2**	**3**	**7**
9			**1**	**1**	**1**	**1**	**1**	**2**	**4**	**8**
10		**1**	**1**	**1**	**1**	**1**	**2**	**3**	**5**	**12**
11							1	2	5	13
12						1	2	4	8	20
13						1	2	4	9	22
14					1	2	3	6	11	31
15					1	2	4	6	14	36
16				1	2	3	4	8	17	48
17				1	2	2	4	8	19	55
18			1	2	2	3	5	11	25	73
19			1	1	1	2	5	11	28	83
20		1	1	1	1	3	7	15	34	107
21					1	3	7	15	40	123
22					1	4	8	19	47	154
23					2	4	9	20	54	177
24				1	3	6	11	26	66	220
25				1	3	5	12	27	74	**251**

Tables 10a-10i: Counts of Restricted Integer Partitions for Specified Values of n, k, a, and b:

10a: ($a = 2$)

	a=	2	2	2	2	2	2	2	2	2	2	2	2
	b=	10	10	10	10	10	10	10	10	10	10	10	10
n	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2		**1**											
3		**1**											
4		**1**	1										
5		**1**	1										
6		**1**	2	1									
7		**1**	2	1									
8		**1**	3	2	1								
9		**1**	3	3	1								
10		**1**	4	4	2	1							
11			4	5	3	1							
12			5	7	5	2	1						
13			4	8	6	3	1						
14			4	10	9	5	2	1					
15			3	11	11	7	3	1					
16			3	12	15	10	5	2	1				
17			2	12	17	13	7	3	1				
18			2	13	21	18	11	5	2	1			
19			1	12	23	22	14	7	3	1			
20				13	27	28	20	11	5	2	1		
21				11	28	33	25	15	7	3	1		
22				10	31	40	33	21	11	5	2	1	
23				8	31	45	40	27	15	7	3	1	
24				7	33	52	51	36	22	11	5	2	1
25				5	**31**	57	59	45	28	15	7	3	1

Table 10b: $a = 3$

	a=	3	3	3	3	3	3	3	3	3	3	3	3
	b=	10	10	10	10	10	10	10	10	10	10	10	10
n	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3		**1**											
4		**1**											
5		**1**											
6		**1**	1										
7		**1**	1										
8		**1**	2										
9		**1**	2	1									
10		**1**	3	1									
11			3	2									
12			4	3	1								
13			4	4	1								
14			4	5	2								
15			3	7	3	1							
16			3	8	5	1							
17			2	9	6	2							
18			2	10	9	3	1						
19			1	10	11	5	1						
20			1	10	14	7	2						
21				10	16	10	3	1					
22				9	19	13	5	1					
23				8	20	17	7	2					
24				7	23	21	11	3	1				
25				5	23	26	14	5	1				

Table 10c: $a = 4$

	a=	4	4	4	4	4	4	4	4	4	4	4	4
	b=	10	10	10	10	10	10	10	10	10	10	10	10
n	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													
4		**1**											
5		**1**											
6		**1**											
7		**1**											
8		**1**	1										
9		**1**	1										
10		**1**	2										
11			2										
12			3	1									
13			3	1									
14			4	2									
15			3	3									
16			3	4	1								
17			2	5	1								
18			2	7	2								
19			1	7	3								
20			1	8	5	1							
21				8	6	1							
22				8	9	2							
23				7	10	3							
24				7	13	5	1						
25				5	14	7	1						

Table 10d: $a = 5$

n	a=	5	5	5	5	5	5	5	5	5	5	5	5
	b=	10	10	10	10	10	10	10	10	10	10	10	10
	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													
4													
5		**1**											
6		**1**											
7		**1**											
8		**1**											
9		**1**											
10		**1**	1										
11			1										
12			2										
13			2										
14			3										
15			3	1									
16			3	1									
17			2	2									
18			2	3									
19			1	4									
20			1	5	1								
21				6	1								
22				6	2								
23				6	3								
24				6	5								
25				5	6	1							

Table 10e: $a = 6$

n	a=	6	6	6	6	6	6	6	6	6	6	6	6
	b=	10	10	10	10	10	10	10	10	10	10	10	10
	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													
4													
5													
6		**1**											
7		**1**											
8		**1**											
9		**1**											
10		**1**											
11													
12			1										
13			1										
14			2										
15			2										
16			3										
17			2										
18			2	1									
19			1	1									
20				3									
21				3									
22				4									
23				4									
24				5	1								
25				4	1								

Table 10f: $a = 7$

n	a=	7	7	7	7	7	7	7	7	7	7	7	7
	b=	10	10	10	10	10	10	10	10	10	10	10	10
	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													
4													
5													
6													
7		**1**											
8		**1**											
9		**1**											
10		**1**											
11													
12													
13													
14			1										
15			1										
16			2										
17			2										
18			2										
19			1										
20			1										
21				1									
22				1									
23				2									
24				3									
25				3									

Table 10g: $a = 8$

n	a=	8	8	8	8	8	8	8	8	8	8	8	8
	b=	10	10	10	10	10	10	10	10	10	10	10	10
	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													
4													
5													
6													
7													
8		**1**											
9		**1**											
10		**1**											
11													
12													
13													
14													
15													
16			1										
17			1										
18			2										
19			1										
20			1										
21													
22													
23													
24				1									
25				1									

Table 10h: $a = 9$

n	a=	9	9	9	9	9	9	9	9	9	9	9	9
	b=	10	10	10	10	10	10	10	10	10	10	10	10
	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													
4													
5													
6													
7													
8													
9		**1**											
10		**1**											
11													
12													
13													
14													
15													
16													
17													
18			1										
19			1										
20			1										
21													
22													
23													
24													
25													

Table 10i: $a = 10$

n	a=	10	10	10	10	10	10	10	10	10	10	10	10
	b=	10	10	10	10	10	10	10	10	10	10	10	10
	k=	1	2	3	4	5	6	7	8	9	10	11	12
1													
2													
3													
4													
5													
6													
7													
8													
9													
10		**1**											
11													
12													
13													
14													
15													
16													
17													
18													
19													
20			1										
21													
22													
23													
24													
25													

Conclusions

This paper presents a unified algorithm ("RICs_RIPs") that generates both restricted integer compositions and restricted integer partitions under the two most commonly imposed restrictions simultaneously – upper and lower bounds on the number of parts allowed, and concurrently, upper and lower bounds on the values of those parts (the algorithm can implement each constraint individually, or no constraints for the unrestricted case). These two fundamental combinatorial objects are important for mathematical, statistical, and scientific applications, yet no other algorithms exist to generate either, let alone both, under these two common restrictions when applied simultaneously. And while the basic connection between integer compositions and integer partitions superficially is obvious, algorithms that generate them, whether restricted or unrestricted, often are quite dissimilar. The RICs_RIPs algorithm is a unifying exception to this – it is the first to provide an important *algorithmic* link between these two combinatorial objects due to its direct foundation on a very fundamental mathematical construct, namely, Pascal's triangle and its (Fibonacci series) off-diagonals. RICs_RIPs is recursive, and given its generality, it is reasonably fast with good time complexity. Actual code implementing it in the most widely available statistical software programming language is included herein. Finally, this paper also proposes a general, closed-form solution to the previously open problem of counting the number of doubly-restricted integer compositions; its formulaic link to an analogous solution for counting doubly-restricted integer partitions is shown to mirror the previously unidentified algorithmic link between these two combinatorial objects.

References

Andrews, G. (1998), *The Theory of Partitions*, Cambridge University Press, Cambridge, UK.

Beyer, T. and Swinehart, D. (1973), Algorithm 448: Number of Multiply-Restricted Partitions, *Communications of the ACM*, Vol.16 (6), 379.

Chinn, P. and Heubach, S. (2003), Compositions of *n* with No Occurrence of *k*, *Congressus Numerantium*, 164, 33-51.

Ehrlich, G. (1973), Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations, *Journal of the ACM*, Vol.20, 500-513.

Grimaldi, R. (2001), Compositions Without the Summand 1, *Congressus Numerantium*, 152, 33-43.

Heubach, S. and Mansour, T. (2004), Compositions of *n* with Parts in a Set, *Congressus Numerantium*, 168, 127-143.

Kimberling, C. (2008), email correspondence with author J.D. Opdyke, July 12, 2008 and July 24, 2008.

Kimberling, C. (2002), Path-Counting and Fibonacci Numbers, *The Fibonacci Quarterly*, Vol. 40(4).

Kimberling, C. (2001), Enumeration of Paths, Compositions of Integers, and Fibonacci Numbers, *The Fibonacci Quarterly*, Vol. 39(5).

Klingsberg, P. (1982), A Gray Code for Compositions, *Journal of Algorithms*, Vol.3, 41-44.

Koepf, W. (1998), *Hypergeometric Summation: An Algorithmic Approach to Summation and Special Function Identities*, Braunschweig, Germany: Vieweg.

Knuth, D. and Szwarcfiter, J. (1974), A Structured Program to Generate All Topological Sorting Arrangements, *Information Processing Letters*, 2, 153-157.

Knuth, D. (1994), *The Stanford Graphbase*, ACM Press, New York, New York, and Addison-Wesley, Reading, MA.

Knuth, D. (1997), *The Art of Computer Programming*, Addison-Wesley, Reading, MA.

Ruskey, F. (2003), *Combinatorial Generation*, Working Version (1j-CSC 425/520).

Sanchis, L. and Squire, M. (1996), Parallel Algorithms for Counting and Randomly Generating Integer Partitions, *Journal of Parallel and Distributed Computing*, Vol.34 (1), 29-35.

Sloan, N.J.A. (2008), editor, *The Online Encyclopedia of Integer Sequences*, http://www.research.att.com/~njas/sequences

Stojmenovic, I. (2008), Generating All and Random Instances of a Combinatorial Object, in *Handbook of Applied Algorithms: Solving Scientific, Engineering and Practical Problems*, Nayak, A. and Stojmenovic, I., eds., John Wiley & Sons, Inc.

Uppuluri, V. and Carpenter, J. (2006), A Problem of Restricted Partitions, *Naval Research Logistics Quarterly*, Vol.21 (1), 201-205.

Walsh, T. (2000), Loop-Free Sequencing of Bounded Integer Compositions, *Journal of Combinatorial Mathematics and Combinatorial Computing*, Vol.33, 323-345.

White, J. (1970), Algorithm 373: Number of Doubly Restricted Partitions[A1], *Communications of the ACM*, Vol.13(2), 120.

White, J. (1970), Algorithm 374: Restricted Partition Generator[A1], *Communications of the ACM*, Vol.13(2), 120.

Yamanaka, K., et al. (2007), Constant Time Generation of Integer Partitions, *IEICE Transaction Fundamentals*, Vol.E90–A, No.5, May.

Appendix A

SAS® Code Implementing the RICs_Base Algorithm

RICs_Base Algorithm written in SAS® (requires on Base SAS® module):

```
/* code above macro */
/* code above macro */
/* code above macro */


%macro RICs_Base(comp_n=, file_dir=, logfile=);


*** Valid values for user-specified macro variable comp_n of the macro RICs_Base (Restricted Integer Compositions - Base
Case):   comp_n   - integers greater than 1
***;

*** save pre-macro SAS options to reinstitute after the macro run is completed.;

  proc optsave;
  run;

  options  nosource
           pagesize=max
           MSYMTABMAX=max
```

```
          ls=256
          nocenter
          nodate
          nonumber
          nonotes
          nomprint
          nomlogic
          ;

*** verify that a valid file directory is specified for the output file.;

   libname dir_chk "&file_dir.";
   %if %sysfunc(libref(dir_chk)) ~= 0 %then %do;
      skip 2;
      %put The directory for the output file specified by the user in the RICs macro does not exist.  Please create it.;
      skip 2;
      %GOTO end_RICs;
   %end;


*** redirect the SAS log file for use as the user-specified output file.;

   filename logprint "&file_dir.\&logfile.";
   proc printto log=logprint new;
   run;


*** check to make sure the macro variable values passed by the user are valid.;

   %let badval_not_integer=0;
   %if &comp_n.= %then %let badval_not_integer=1;
   %else %if %verify(&comp_n.,0123456789)>0 %then %let badval_not_integer=1;
   %else %if %substr(&comp_n.,1,1)=0 %then %let badval_not_integer=1;
   %if &badval_not_integer.=1 %then %do;
      %put;
      %put The user-specified value for the macro variable comp_n must be a positive integer.;
      %put;
      %GOTO end_RICs;
   %end;

   %if &comp_n.< 2 %then %do;
      %put;
      %put The user-specified value for the macro variable comp_n must be equal to or greater than 2.;
      %put;
      %GOTO end_RICs;
   %end;


   %macro celllabels;
      %do nm=1 %to &num_off_diags;
         level&nm.
      %end;
   %mend celllabels;
   %macro cellinitialize;
      %global comp_num;
      %let comp_num = 0;
      %do qp=1 %to &num_off_diags;
         %global cell&qp.;
         %let cell&qp. = ;
      %end;
   %mend cellinitialize;
   %macro cellvals(currlevel=);
      %do po=1 %to &currlevel.;
         %cmpres(&&cell&po.)
      %end;
   %mend cellvals;

  %macro Node2Node(nrow=, ncol=, level=);
    %if &ncol.=0 %then %do;
      %let binet_fibn =
%sysfunc(round(%sysevalf(%sysevalf(1/%sysfunc(sqrt(5)))*%sysevalf(%sysevalf(%sysevalf(%sysevalf(1+%sysfunc(sqrt(5)))/2)**
%eval(&comp_n.-1))-%sysevalf(%sysevalf(%sysevalf(1-%sysfunc(sqrt(5)))/2)**%eval(&comp_n.-1))))));
       %put There are %cmpres(&binet_fibn.) integer compositions of %cmpres(&comp_n.) with all part values greater than 1.
These are listed below.;
       %put;
       %put Comp.#   %cmpres(%celllabels);
         %cellinitialize;
       %let cell1 = %cmpres(&comp_n.);
       %let comp_num = %eval(&comp_num.+1);
       %put %cmpres(&comp_num.)        %cmpres(%cellvals(currlevel=&ih.));
    %end;
    %else %do;
      %let nxt_2_last = %eval(&ih.-1);
      %if &ncol. = 1 %then %do ji=1 %to &nrow.;
        %let cell&nxt_2_last. = %eval(&ji.+1);
        %let cell&ih. = %eval(&nrow. + 2 - &ji.);
          %let comp_num = %eval(&comp_num.+1);
```

```
          %put %cmpres(&comp_num.)        %cmpres(%cellvals(currlevel=&ih.));
       %end;
       %else %do;
         %let level1 = %eval(&level.+1);
         %let cell&level1. = 2;
         %Node2Node(nrow=%eval(&nrow.-1),ncol=%eval(&ncol.-1),level=%eval(&level.+1));
       %end;
      %end;
     %if &nrow.>1 & &level.>0 %then %do;
       %let cell&level. = %eval(&&cell&level.+1);
       %Node2Node(nrow=%eval(&nrow.-1),ncol=&ncol.,level=&level.);
     %end;
   %mend Node2Node;

   %let num_off_diags = %sysfunc(floor(%sysevalf(&comp_n./2)));
   %do ih=1 %to &num_off_diags.;
     %Node2Node(nrow=%eval(&comp_n.-1-&ih.),ncol=%eval(&ih.-1),level=0);
   %end;


   %end_RICs:

*** redirect the SAS log file back to its default.;
    proc printto;
    run;

*** reinstitute the SAS options that were in place before the macro was called.;
   proc optload;
   run;

%mend RICs_Base;



%RICs_Base(comp_n=11,
           file_dir=C:\RICs_root,
           logfile=RICs_Base_n11.txt
           );



/* code below macro */
/* code below macro */
/* code below macro */
```

Appendix B

SAS® Code Implementing the RICs Algorithm

RICs Algorithm written in SAS® (requires only Base SAS® module):

```
/* code above macro */
/* code above macro */
/* code above macro */


%macro RICs(comp_n=, mink=, maxk=, minpart=, maxpart=, file_dir=, logfile=);


*** Valid values for user-specified macro variables of the macro RICs (Restricted Integer Compositions):
   comp_n   - 0 < integers (also referred to as 'n')
   maxpart  - 0 < integers <= comp_n (also referred to as 'b')
   minpart  - 0 < integers <= maxpart (also referred to as 'a')
   maxk     - 0 < integers <= floor(comp_n/minpart)
   mink     - ceiling(comp_n/maxpart) <= integers <= maxk
   file_dir - an existing file directory, such as c:\homedir\user, where the output file is placed
   logfile  - a filename, such as RICs_run2.txt, which names the output file
***;


*** save SAS options in effect before calling RICs to reinstitute after RICs is completed.;

  proc optsave;
  run;

  options  nosource
           pagesize=max
           MSYMTABMAX=max
           ls=256
```

```
          nocenter
          nodate
          nonumber
          nonotes
          nomprint
          nomlogic
          ;

*** verify that a valid file directory is specified for the output file.;

   libname dir_chk "&file_dir.";
   %if %sysfunc(libref(dir_chk)) ~= 0 %then %do;
      skip 2;
      %put The directory for the output file specified by the user in the RICs macro does not exist.  Please create it.;
      skip 2;
      %GOTO end_RICs;
   %end;


*** redirect the SAS log file for use as the user-specified output file.;

   filename logprint "&file_dir.\&logfile.";
   proc printto log=logprint new;
   run;


*** check to make sure the macro variable values passed by the user are valid.;

  %let stopprogram = 0;

  %macro isBlank(param);
     %sysevalf(%superq(param)=,boolean)
  %mend isBlank;

  %let badval_not_integer=0;
  %if %isBlank(&comp_n.) %then %let badval_not_integer=1;
  %else %if %verify(&comp_n.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&comp_n.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable comp_n must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %let badval_not_integer=0;
  %if %isBlank(&mink.) %then %let badval_not_integer=1;
  %else %if %verify(&mink.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&mink.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable mink must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %let badval_not_integer=0;
  %if %isBlank(&maxk.) %then %let badval_not_integer=1;
  %else %if %verify(&maxk.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&maxk.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable maxk must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %let badval_not_integer=0;
  %if %isBlank(&minpart.) %then %let badval_not_integer=1;
  %else %if %verify(&minpart.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&minpart.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %let badval_not_integer=0;
  %if %isBlank(&maxpart.) %then %let badval_not_integer=1;
  %else %if %verify(&maxpart.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&maxpart.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable maxpart must be a positive integer.;
    %put;
    %let stopprogram = 1;
```

```
  %end;

  %if &stopprogram. ~= 1 %then %do;
    %if &minpart.<1 %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be an integer greater than zero.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &maxpart.< &minpart. %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be equal to or smaller than that specified for
maxpart.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &maxpart.<1 %then %do;
    %put;
    %put The user-specified value for the macro variable maxpart must be an integer greater than zero.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &comp_n.< &minpart. %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be equal to or smaller than that specified for
comp_n.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &maxk.< &mink. %then %do;
    %put;
    %put The user-specified value for the macro variable mink must be equal to or smaller than that specified for maxk.;
    %put;
    %let stopprogram = 1;
    %end;

    %if %sysevalf(%sysfunc(floor(%sysevalf(&comp_n./2)))< &mink., boolean) %then %do;
    %put;
    %put The user-specified value for the macro variable mink cannot be larger than that specified for comp_n divided by
2, rounded down.;
    %put;
    %let stopprogram = 1;
    %end;

  %end;


  %if &stopprogram.=1 %then %goto end_RICs;


*** find the actual maximum # of parts that result after the user-specified composition restrictions are imposed,
    in case it is not what the user specified in maxk.;

  %let maxk_hold = 0;
  %do zy=&maxk. %to &mink. %by -1;
    %if %sysevalf((&comp_n./&zy.)>=&minpart. & (&comp_n./&zy.)<=&maxpart., boolean) %then %do;
       %let maxk_hold = &zy.;
       %let zy = &mink.;
    %end;
  %end;

  %if &maxk_hold.=0 %then %do;
     %put;
     %put There are no restricted integer compositions of &comp_n. that satisfy the user-specified restrictions listed
below:;
     %put;
     %put List all restricted integer compositions of integer n = %cmpres(&comp_n.);
      %put with at least %cmpres(&mink.) parts and no more than %cmpres(&maxk.) parts;
      %put and only part values within the range from %cmpres(&minpart.) through %cmpres(&maxpart.).;
     %put;
      %goto end_RICs;
  %end;

  %if &maxk_hold. ~= &maxk. %then %do;
  %put;
  %put The maximum value possible for maxk is floor(comp_n / minpart).  Because this is smaller than the user-specified
value (%cmpres(&maxk.)),;
   %put the value of maxk has reassigned to equal floor(comp_n / minpart) = %cmpres(&maxk_hold.).;
  %put;
  %let maxk = &maxk_hold.;
  %end;

  %let maxpart_hold = %sysfunc(min(&maxpart.,&comp_n.));
```

```
  %if &maxpart_hold. ~= &maxpart. %then %do;
  %put;
  %put The user-specified value for the macro variable maxpart cannot be larger than that specified for comp_n, so
maxpart has been set equal to comp_n.;
  %put;
  %let maxpart = &maxpart_hold.;
  %end;


*** define initialization and output/printing macros;

   %macro celllabels;
      %do nm=1 %to &maxk.;
         level&nm.
      %end;
   %mend celllabels;

   %macro cellinitialize;
    %global comp_num;
    %let comp_num = 0;
      %do qp=1 %to &maxk.;
       %global cell&qp.;
          %let cell&qp. = ;
      %end;
   %mend cellinitialize;

   %macro cellvals(currlevel=);
      %do po=1 %to &currlevel.;
         %cmpres(&&cell&po.)
      %end;
   %mend cellvals;

  %put;
  %put List all restricted integer compositions of integer n = %cmpres(&comp_n.);
   %put with at least %cmpres(&mink.) part(s) and no more than %cmpres(&maxk.) part(s);
   %put and only part values within the range from %cmpres(&minpart.) through %cmpres(&maxpart.).;
  %put;

  %put Comp.#   %cmpres(%celllabels);
   %cellinitialize;


*** define the main Node2Node macro that starts at each of the "Fibonacci" off-diagonals (unless minpart=1) and traces
    composition paths through Pascals triangle;

  %macro Node2Node(nrow=, ncol=, level=, cum_sum_parts=);
    %if &ih.~=1 %then %do;
         %if &ncol. = 1 %then %do;
         %let nxt_2_last = %eval(&ih.-1);
        %let low = %sysfunc(max(&minpart.,%eval(&comp_n. - &cum_sum_parts. - &maxpart.)));
        %let high = %sysfunc(min(&maxpart.,%eval(&comp_n. - &cum_sum_parts. - &minpart.)));
          %do ji=&low. %to &high.;
            %let cell&nxt_2_last. = &ji.;
            %let comp_num = %eval(&comp_num.+1);
            %let cell&ih. = %eval(&comp_n. - &cum_sum_parts. - &ji.);
            %put %cmpres(&comp_num.)        %cmpres(%cellvals(currlevel=&ih.));
        %end;
      %end;
      %else %do;
        %let level1 = %eval(&level.+1);
         %let cell&level1. = &minpart.;
          %let cum_sum_parts1 = %eval(&cum_sum_parts.+ &minpart.);
        %let toploop = %sysfunc(min(%eval(&maxpart.-&minpart.),%eval(&nrow.-&minpart.-&ncol.+1)));
        %let avg_amt_left = %sysevalf((&comp_n.-&cum_sum_parts1.)/(&ih.-&level1.));
        %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart. & &&cell&level1. <= &maxpart., boolean)
            %then %Node2Node(nrow=%eval(&nrow.- &minpart. + 1),ncol=%eval(&ncol.-
1),level=%eval(&level.+1),cum_sum_parts=&cum_sum_parts1.);
        %else %do sr=1 %to &toploop.;
            %let cell&level1. = %eval(&&cell&level1.+1);
            %let cum_sum_parts1 = %eval(&cum_sum_parts1.+1);
           %let avg_amt_left = %sysevalf((&comp_n.-&cum_sum_parts1.)/(&ih.-&level1.));
           %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart. & &&cell&level1. <= &maxpart.,
boolean) %then %do;
             %let sr_use = &sr.;
             %let sr = &toploop.;
               %Node2Node(nrow=%eval(&nrow.-&minpart.+1-&sr_use.),ncol=%eval(&ncol.-
1),level=%eval(&level.+1),cum_sum_parts=&cum_sum_parts1.);
          %end;
        %end;
      %end;
     %end;
    %else %do;
      %let cell1 = %cmpres(&comp_n.);
      %let comp_num = %eval(&comp_num.+1);
      %put %cmpres(&comp_num.)        %cmpres(%cellvals(currlevel=1));
    %end;
```

```
     %if &nrow.>1 & &level.>0 %then %do;
      %let cell&level. = %eval(&&cell&level.+1);
      %let cum_sum_parts = %eval(&cum_sum_parts.+1);
       %if &&cell&level.<&minpart. %then %do;
         %let cum_sum_parts = %eval(&cum_sum_parts.+ &minpart. - &&cell&level.);
         %let nrow = %eval(&nrow.- &minpart. + &&cell&level.);
          %let cell&level. = &minpart.;
       %end;

      %let avg_amt_left = %sysevalf((&comp_n.-&cum_sum_parts.)/(&ih.-&level.));
      %let lowloop = %eval(&&cell&level.+1);
      %let toploop = %sysfunc(min(%eval(&maxpart.-&&cell&level.),%eval(&nrow.-1-&ncol.)));
      %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart. & &&cell&level. <= &maxpart., boolean)
      %then %Node2Node(nrow=%eval(&nrow.-1),ncol=&ncol.,level=&level.,cum_sum_parts=&cum_sum_parts.);
      %else %do vu=&lowloop. %to &toploop.;
         %let cell&level. = %eval(&&cell&level.+1);
         %let cum_sum_parts = %eval(&cum_sum_parts.+1);
         %let avg_amt_left = %sysevalf((&comp_n.-&cum_sum_parts.)/(&ih.-&level.));
         %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart. & &&cell&level. <= &maxpart., boolean)
%then %do;
           %let vu_use = &vu.;
           %let vu = &toploop.;
             %Node2Node(nrow=%eval(&nrow.-1-&vu_use.),ncol=&ncol.,level=&level.,cum_sum_parts=&cum_sum_parts.);
        %end;
      %end;
    %end;
  %mend Node2Node;


*** call Node2Node on each of the "Fibonacci" off-diagonals of Pascals triangle (or its horizontal row if minpart=1),
    tracking the "path" of the composition in the triangle via its combinatoric representation: "rows" on top and
"columns"
   on the bottom of the "n-choose-k" representation of the triangle.;

  %do ih=&mink. %to &maxk.;
    %let rowdec=&ih.;
     %if &minpart.=1 %then %let rowdec=0;
    %let avg_amt_left = %sysevalf(&comp_n./&ih.);
    %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart., boolean)
    %then %Node2Node(nrow=%eval(&comp_n.-1-&rowdec.),ncol=%eval(&ih.-1),level=0,cum_sum_parts=0);
  %end;
  %put;

  %end_RICs:


*** redirect the SAS log file back to its default.;

   proc printto;
   run;

*** reinstitute the SAS options that were in place before RICs was called.;

  proc optload;
  run;


%mend RICs;


%RICs(comp_n=11,
      mink=1,
      maxk=5,
      minpart=2,
      maxpart=4,
      file_dir=C:\RICs_Dir,
      logfile=RICs_n11_1k5_a2_b4.txt
      );

/* code below RICs macro */
/* code below RICs macro */
/* code below RICs macro */
```

Appendix C

SAS® Code Implementing the RICs_RIPs Algorithm

RICs_RIPs Algorithm written in SAS® (requires only Base SAS® module):

```
/* code above macro */
/* code above macro */
/* code above macro */


%macro RICs_RIPs(comp_part=, part_n=, mink=, maxk=, minpart=, maxpart=, file_dir=, logfile=);


*** save SAS options in effect before calling RICs_RIPs to reinstitute after RICs_RIPs is completed.;

  proc optsave;
  run;

   options nosource
           pagesize=max
         MSYMTABMAX=max
           ls=256
           nocenter
           nodate
         nonumber
         nonotes
         nomprint
         nomlogic
         ;

*** Valid values for user-specified macro variables of the macro RICs_RIPs (Restricted Integer Compositions/Partitions):
  comp_part - comp or part (for composition or partition)
   part_n    - 0 < integers (also referred to as 'n')
   maxpart   - 0 < integers <= part_n (also referred to as 'b')
   minpart   - 0 < integers <= maxpart (also referred to as 'a')
   maxk      - 0 < integers <= floor(part_n/minpart)
   mink      - ceiling(part_n/maxpart) <= positive integers <= maxk
   file_dir  - an existing file directory, such as c:\homedir\user, where the output file is placed
   logfile   - a filename, such as RICs_RIPs_run2.txt, which names the output file
***;


*** verify that a valid file directory is specified for the output file.;

   libname dir_chk "&file_dir.";
   %if %sysfunc(libref(dir_chk)) ~= 0 %then %do;
      skip 2;
      %put The directory for the output file specified by the user in the RICs_RIPs macro does not exist.  Please create
it.;
      skip 2;
      %GOTO end_RICs_RIPs;
   %end;


*** redirect the SAS log file for use as the user-specified output file.;

   filename logprint "&file_dir.\&logfile.";
   proc printto log=logprint new;
   run;


*** check to make sure the macro variable values passed by the user are valid.;

  %let stopprogram = 0;

  %if %upcase(&comp_part.)~=COMP & %upcase(&comp_part.)~=PART %then %do;
    %put;
    %put The user-specified value for the macro variable comp_part must be 'COMP' (composition) or 'PART' (partition).;
    %put;
    %let stopprogram = 1;
  %end;

  %macro isBlank(param);
     %sysevalf(%superq(param)=,boolean)
  %mend isBlank;

  %if %upcase(&comp_part.)=COMP %then %let comb_obj=composition;
  %else %let comb_obj=partition;

  %let badval_not_integer=0;
  %if %isBlank(&part_n.) %then %let badval_not_integer=1;
  %else %if %verify(&part_n.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&part_n.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable part_n must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;
```

```
  %let badval_not_integer=0;
  %if %isBlank(&mink.) %then %let badval_not_integer=1;
  %else %if %verify(&mink.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&mink.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable mink must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %let badval_not_integer=0;
  %if %isBlank(&maxk.) %then %let badval_not_integer=1;
  %else %if %verify(&maxk.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&maxk.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable maxk must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %let badval_not_integer=0;
  %if %isBlank(&minpart.) %then %let badval_not_integer=1;
  %else %if %verify(&minpart.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&minpart.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %let badval_not_integer=0;
  %if %isBlank(&maxpart.) %then %let badval_not_integer=1;
  %else %if %verify(&maxpart.,0123456789)>0 %then %let badval_not_integer=1;
  %else %if %substr(&maxpart.,1,1)=0 %then %let badval_not_integer=1;
  %if &badval_not_integer.=1 %then %do;
    %put;
    %put The user-specified value for the macro variable maxpart must be a positive integer.;
    %put;
    %let stopprogram = 1;
  %end;

  %if &stopprogram. ~= 1 %then %do;
    %if &minpart.<1 %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be an integer greater than zero.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &maxpart.< &minpart. %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be equal to or smaller than that specified for
maxpart.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &maxpart.<1 %then %do;
    %put;
    %put The user-specified value for the macro variable maxpart must be an integer greater than zero.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &part_n.< &minpart. %then %do;
    %put;
    %put The user-specified value for the macro variable minpart must be equal to or smaller than that specified for
part_n.;
    %put;
    %let stopprogram = 1;
    %end;

    %if &maxk.< &mink. %then %do;
    %put;
    %put The user-specified value for the macro variable mink must be equal to or smaller than that specified for maxk.;
    %put;
    %let stopprogram = 1;
    %end;

    %if %sysevalf(%sysfunc(floor(%sysevalf(&part_n./2)))< &mink., boolean) %then %do;
    %put;
    %put The user-specified value for the macro variable mink cannot be larger than that specified for part_n divided by
2, rounded down.;
    %put;
```

```
    %let stopprogram = 1;
    %end;

  %end;

  %if &stopprogram.=1 %then %goto end_RICs_RIPs;


*** find the actual maximum # of parts that result after the user-specified composition/partition restrictions are
imposed,
    in case it is not what the user specified in maxk.;

  %let maxk_hold = 0;
  %do zy=&maxk. %to &mink. %by -1;
    %if %sysevalf((&part_n./&zy.)>=&minpart. & (&part_n./&zy.)<=&maxpart., boolean) %then %do;
       %let maxk_hold = &zy.;
       %let zy = &mink.;
    %end;
  %end;

  %if &maxk_hold.=0 %then %do;
     %put;
     %put There are no integer %cmpres(&comb_obj.) of &part_n. that satisfy the user-specified restrictions listed
below:;
     %put;
     %put List all integer %cmpres(&comb_obj.)s of integer n = %cmpres(&part_n.);
      %put with at least %cmpres(&mink.) parts and no more than %cmpres(&maxk.) parts;
      %put and only part values within the range from %cmpres(&minpart.) through %cmpres(&maxpart.).;
     %put;
      %goto end_RICs_RIPs;
  %end;

  %if &maxk_hold. ~= &maxk. %then %do;
  %put;
  %put The maximum value possible for maxk is floor(part_n / minpart).  Because this is smaller than the user-specified
value (%cmpres(&maxk.)),;
   %put the value of maxk has reassigned to equal floor(part_n / minpart) = %cmpres(&maxk_hold.).;
  %put;
  %let maxk = &maxk_hold.;
  %end;

  %let maxpart_hold = %sysfunc(min(&maxpart.,&part_n.));
  %if &maxpart_hold. ~= &maxpart. %then %do;
  %put;
  %put The user-specified value for the macro variable maxpart cannot be larger than that specified for part_n, so
maxpart has been set equal to part_n.;
  %put;
  %let maxpart = &maxpart_hold.;
  %end;


*** define initialization and output/printing macros;

   %macro celllabels;
      %do nm=1 %to &maxk.;
         level&nm.
      %end;
   %mend celllabels;

   %macro cellinitialize;
    %global part_num;
    %let part_num = 0;
      %do qp=1 %to &maxk.;
       %global cell&qp.;
          %let cell&qp. = ;
      %end;
   %mend cellinitialize;

   %macro cellvals(currlevel=);
      %do po=1 %to &currlevel.;
         %cmpres(&&cell&po.)
      %end;
   %mend cellvals;

  %put;
  %put List all integer %cmpres(&comb_obj.)s of integer n = %cmpres(&part_n.);
   %put with at least %cmpres(&mink.) part(s) and no more than %cmpres(&maxk.) part(s);
   %put and only part values within the range from %cmpres(&minpart.) through %cmpres(&maxpart.).;
  %put;

   %if %upcase(&comp_part.)=COMP %then %put Comp.#   %cmpres(%celllabels);
   %else %put Part.#   %cmpres(%celllabels);
   %cellinitialize;


*** define the main Node2Node macro that starts at each of the "Fibonacci" off-diagonals (unless minpart=1) and traces
    composition/partition paths through Pascals triangle;
```

```
  %macro Node2Node(nrow=, ncol=, level=, cum_sum_parts=, minpart=);
    %if &ih.~=1 %then %do;
         %if &ncol. = 1 %then %do;
          %let nxt_2_last = %eval(&ih.-1);
         %let low = %sysfunc(max(&minpart.,%eval(&part_n. - &cum_sum_parts. - &maxpart.)));
         %let high = %sysfunc(min(&maxpart.,%eval(&part_n. - &cum_sum_parts. - &minpart.)));
         %if %upcase(&comp_part.)=PART %then %let high = %eval(&high.-%sysfunc(floor(%sysevalf((&high.-&low.+1)/2))));
        %do ji=&low. %to &high.;
            %let cell&nxt_2_last. = &ji.;
            %let part_num = %eval(&part_num.+1);
            %let cell&ih. = %eval(&part_n. - &cum_sum_parts. - &ji.);
            %put %cmpres(&part_num.)        %cmpres(%cellvals(currlevel=&ih.));
         %end;
       %end;
       %else %do;
         %let level1 = %eval(&level.+1);
          %let cell&level1. = &minpart.;
           %let cum_sum_parts1 = %eval(&cum_sum_parts.+ &minpart.);
             %let minpart1 = &minpart.;
         %let toploop = %sysfunc(min(%eval(&maxpart.-&minpart.),%eval(&nrow.-&minpart.-&ncol.+1)));
         %let avg_amt_left = %sysevalf((&part_n.-&cum_sum_parts1.)/(&ih.-&level1.));
         %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart. & &&cell&level1. <= &maxpart., boolean)
            %then %Node2Node(nrow=%eval(&nrow.- &minpart. + 1),ncol=%eval(&ncol.-
1),level=%eval(&level.+1),cum_sum_parts=&cum_sum_parts1., minpart=&minpart.);
         %else %do sr=1 %to &toploop.;
            %let cell&level1. = %eval(&&cell&level1.+1);
           %if %upcase(&comp_part.)=PART %then %let minpart1 = %eval(&minpart1.+1);
            %let cum_sum_parts1 = %eval(&cum_sum_parts1.+1);
           %let avg_amt_left = %sysevalf((&part_n.-&cum_sum_parts1.)/(&ih.-&level1.));
           %if %sysevalf(&avg_amt_left. >= &minpart1. & &avg_amt_left. <= &maxpart. & &&cell&level1. <= &maxpart.,
boolean) %then %do;
             %let sr_use = &sr.;
             %let sr = &toploop.;
               %Node2Node(nrow=%eval(&nrow.-&minpart1.+1-&sr_use.),ncol=%eval(&ncol.-
1),level=%eval(&level.+1),cum_sum_parts=&cum_sum_parts1.,minpart=&minpart1.);
          %end;
         %end;
       %end;
     %end;
    %else %do;
      %let cell1 = %cmpres(&part_n.);
      %let part_num = %eval(&part_num.+1);
      %put %cmpres(&part_num.)        %cmpres(%cellvals(currlevel=1));
    %end;

     %if &nrow.>1 & &level.>0 %then %do;
      %let cell&level. = %eval(&&cell&level.+1);
       %if %upcase(&comp_part.)=PART %then %let minpart = %eval(&minpart.+1);
      %let cum_sum_parts = %eval(&cum_sum_parts.+1);
       %if &&cell&level.<&minpart. %then %do;
         %let cum_sum_parts = %eval(&cum_sum_parts.+ &minpart. - &&cell&level.);
         %let nrow = %eval(&nrow.- &minpart. + &&cell&level.);
          %let cell&level. = &minpart.;
       %end;

      %let avg_amt_left = %sysevalf((&part_n.-&cum_sum_parts.)/(&ih.-&level.));
      %let lowloop = %eval(&&cell&level.+1);
      %let toploop = %sysfunc(min(%eval(&maxpart.-&&cell&level.),%eval(&nrow.-1-&ncol.)));
      %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart. & &&cell&level. <= &maxpart., boolean)
      %then %Node2Node(nrow=%eval(&nrow.-1),ncol=&ncol.,level=&level.,cum_sum_parts=&cum_sum_parts.,minpart=&minpart.);
      %else %do vu=&lowloop. %to &toploop.;
         %let cell&level. = %eval(&&cell&level.+1);
         %if %upcase(&comp_part.)=PART %then %let minpart = %eval(&minpart.+1);
         %let cum_sum_parts = %eval(&cum_sum_parts.+1);
         %let avg_amt_left = %sysevalf((&part_n.-&cum_sum_parts.)/(&ih.-&level.));
         %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart. & &&cell&level. <= &maxpart., boolean)
%then %do;
           %let vu_use = &vu.;
           %let vu = &toploop.;
             %Node2Node(nrow=%eval(&nrow.-1-
&vu_use.),ncol=&ncol.,level=&level.,cum_sum_parts=&cum_sum_parts.,minpart=&minpart.);
        %end;
      %end;
    %end;
  %mend Node2Node;


*** call Node2Node on each of the "Fibonacci" off-diagonals of Pascals triangle (or its horizontal row if minpart=1),
    tracking the "path" of the composition/partition in the triangle via its combinatoric representation: "rows" on top
and "columns"
   on the bottom of the "n-choose-k" representation of the triangle.;

  %do ih=&mink. %to &maxk.;
    %let rowdec=&ih.;
     %if &minpart.=1 %then %let rowdec=0;
    %let avg_amt_left = %sysevalf(&part_n./&ih.);
```

```
    %if %sysevalf(&avg_amt_left. >= &minpart. & &avg_amt_left. <= &maxpart., boolean)
    %then %Node2Node(nrow=%eval(&part_n.-1-&rowdec.),ncol=%eval(&ih.-1),level=0,cum_sum_parts=0,minpart=&minpart.);
  %end;
  %put;

  %end_RICs_RIPs:


*** redirect the SAS log file back to its default.;

   proc printto;
   run;

*** reinstitute the SAS options that were in place before RICs_RIPs was called.;

  proc optload;
  run;


%mend RICs_RIPs;

%RICs_RIPs(comp_part=part,
           part_n=11,
           mink=2,
           maxk=5,
           minpart=2,
           maxpart=4,
           file_dir=C:\RICs_Dir,
           logfile=RIPs_n11_lk5_a2_b4.txt
           );
```

Appendix D

Mathematica® Code Implementing Counting Formulae for Doubly-Restricted Integer Compositions [formulae (2) and (4)] and Doubly-Restricted Integer Partitions [formulae (5) and (7)]

(* Formula (2) *)

$$\text{ncomp}[n\_, a\_, b\_] := \text{If}\left[a \le n \le b, 1 + \sum_{i=\text{Max}[1, n-b]}^{n-a} \text{ncomp}[i, a, b], \sum_{i=\text{Max}[1, n-b]}^{n-a} \text{ncomp}[i, a, b]\right]$$

(* Formula (4) *)

$$\text{ncompk}[n\_, k\_, a\_, b\_] := \text{If}\left[k == 1, \text{If}[a \le n \le b, 1, 0], \sum_{i=\text{Max}[1, n-b]}^{n-a} \text{ncompk}[i, k-1, a, b]\right]$$

(* Formula (5) *)

$$\text{npart}[n\_, a\_, b\_] := \text{If}\left[a \le n \le b, 1 + \sum_{i=\text{Max}[1, n-b]}^{n-a} \text{npart}[i, n-i, b], \sum_{i=\text{Max}[1, n-b]}^{n-a} \text{npart}[i, n-i, b]\right]$$

(* Formula (7) *)

$$\text{npartk}[n\_, k\_, a\_, b\_] := \text{If}\left[k == 1, \text{If}[a \le n \le b, 1, 0], \sum_{i=\text{Max}[1, n-b]}^{n-a} \text{npartk}[i, k-1, n-i, b]\right]$$

abinputs = {{2, 3}, {3, 4}, {4, 5}, {2, 4}, {2, 5}, {4, 5}, {4, 6}, {4, 7}, {4, 8}, {4, 9}, {4, 10}, {4, 11}, {4, 12}};

(* Table 4 *)

```
TableForm[ Table[sol = ncomp[ n, abinputs[[col,1]], abinputs[[col,2]]];  If[sol>0, sol, ""], {n, 1, 25}, col, 1, Length[abinputs]}]]]

(* Table 5 *)

TableForm[ Table[sol = ncompk[ n, k, 2, 5]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 6 *)

TableForm[ Table[sol = ncompk[ n, k, 3, 7]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 6}]]

(* Table 9 *)

TableForm[ Table[sol = npart[n, a, 10]; If[sol>0, sol, ""], {n, 1, 25}, {a, 10, 2, -1}]]

(* Table 10a *)

TableForm[ Table[sol = npartk[n, k, 2, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10b *)

TableForm[ Table[sol = npartk[n, k, 3, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10c *)

TableForm[ Table[sol = npartk[n, k, 4, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10d *)

TableForm[ Table[sol = npartk[n, k, 5, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10e *)

TableForm[ Table[sol = npartk[n, k, 6, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10f *)

TableForm[ Table[sol = npartk[n, k, 7, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10g *)

TableForm[ Table[sol = npartk[n, k, 8, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10h *)

TableForm[ Table[sol = npartk[n, k, 9, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]

(* Table 10i *)

TableForm[ Table[sol = npartk[n, k, 10, 10]; If[sol>0, sol, ""], {n, 1, 25}, {k, 1, 12}]]
```


View publication stats