A Combinatorial Operator in APL

Bob Smith
Sudley Place Software

Originally Written
24 Aug 2016
Latest Update
23 Jun 2017

Table of Contents

Introduction

Counting and generating items is fundamental in mathematics, but has been sorely lacking in APL (notwithstanding the counting functions `!R` and `L!R`); instead we have had to rely upon a patchwork of various library routines.

Moreover, most APL papers on the topic have focused on the implementation of the algorithms rather than their organization and syntax mostly because, at the time, there was no unifying concept nor common syntax.

The main purpose of this document is to present in APL a unified organizing principle to classify and access various Combinatorial Algorithms.

A secondary purpose is to shed light on the relationships between the various algorithms through a new perspective provided by Gian-Carlo Rota's clever way to fit them into a single organizational framework.

The goal of this document is to describe a single APL primitive to both **count** and **generate** various Combinatorial Arrays: permutations, combinations, compositions, partitions, etc. The unifying (and very APL-like) principle for such a primitive is Gian-Carlo Rota's **Twelvefold Way** as described in Richard Stanley's "Enumerative Combinatorics"[1], Knuth's TAoCP, Vol. 4A[3], and Wikipedia[2] among other references.

Twelvefold Way

This elegant notion consolidates twelve of these algorithms into a single `2×2×3` array based on the simple concept of placing balls into boxes (urns, to you old-timers). The three dimensions of the array can be described as follows:

- The balls may be labeled (or not) {2 ways},
- The boxes may be labeled (or not) {2 ways}, and
- The # balls allowed in a box may be one of (at most one | unrestricted | at least one) {3 ways}.

Amazingly, these twelve choices spanning three dimensions knit together within a single concept (balls in boxes) all of the following interesting, fundamental, and previously disparate and disorganized

Combinatorial Algorithms:

- Permutations
- Compositions
- Partitions of a set
- Tuples
- Combinations
- Multisets
- Partitions of a number
- Pigeon Holes

Function Selector

One way to design a primitive to encompass the Twelvefold Way is first to define a one- or two-element integer vector as a function selector in a manner similar to the left argument to dyadic Circle (`L○R`) and dyadic Pi (`LπR`)[8]. The items in the vector `V` are described as follows where the first element of `V` is split into three digits as in `W←0 10 10⊤V[1]`:

`W[1] ∊ 0 1`	for (unlabeled | labeled) **balls**
`W[2] ∊ 0 1`	for (unlabeled | labeled) **boxes**
`W[3] ∊ 0 1 2`	for the # **balls** allowed in each **box** corresponding to the three possibilities of (at most one | unrestricted | at least one), defined as `0 = ≤1` (at most one) ball in each box `1 =` unrestricted `2 = ≥1` (at least one) ball in each box The reasoning for the above choice is that `0` is clearly a maximum unambiguous integer representative of `≤1` and `2` is clearly a minimum unambiguous integer representative of `≥1`, and `1` is in the middle, so it represents both `≤1` and `≥1`, that is, unrestricted.
`V[2] ∊ 0 1`	(optional) where `0` (the default) means **count** the # solutions and `1` means **generate** them.

As a matrix, the algorithms are organized by function selector as follows (and is referred to elsewhere as the **FS Table**):

FS Table	**Balls Per Box**		
	At Most One xx0	**Unrestricted** xx1	**At Least One** xx2
L unlabeled balls 00x R unlabeled boxes	L pigeons into R holes 000	partitions of L into ≤R parts 001	partitions of L into R parts 002
L unlabeled balls 01x R labeled boxes	L-combinations of R items 010	L-multisets of R items 011	compositions of L into R parts 012
L labeled balls 10x R unlabeled boxes	L pigeons into R holes 100	partitions of S into ≤R parts 101	partitions of S into R parts 102
L labeled balls 11x R labeled boxes	L-permutations of R items 110	L-tuples of R items 111	partitions of S into R ordered parts 112

where S is the set `{⍳L}`. The above table was copied with slight changes from Knuth's TAoCP, Vol 4A, p. 390[4].

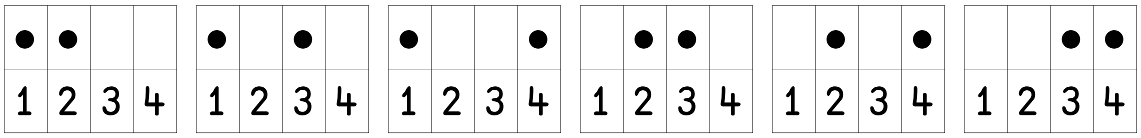
The colored cells in the above table are discussed in the sections here, here, and here.

Although the Function Selector is a simple integer, in this paper it is written as a three-digit number with leading zeros to emphasize that each digit has a separate meaning.

For example:

- A function selector of `010` means unlabeled balls, labeled boxes, and at most one ball in each box.

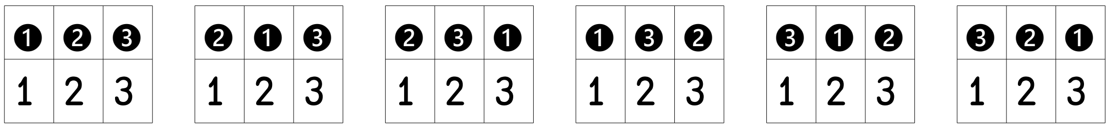
  If we have `2` unlabeled balls (●●) and `4` labeled boxes (`1234`) with at most one ball per box, there are `6` (↔ `2!4`) ways to meet these criteria:



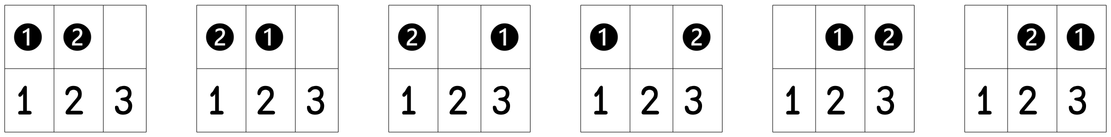
  from which it is easy to see that these criteria correspond to `L` combinations of `R` items (↔ `L!R`). See case **010** below.

- A function selector of `110` means labeled balls & boxes and at most one ball in each box.

  If we have `3` labeled balls (❶❷❸) and `3` labeled boxes (`123`) with at most one ball per box, there are `6` (↔ `(!⍠¯3)3 ↔ 3×2×1`) ways to meet these criteria:



  If we have `2` labeled balls (❶❷) and `3` labeled boxes (`123`) with at most one ball per box, there are `6` (↔ `(!⍠¯2)3 ↔ 3×2`) ways to meet these criteria:



  from which it is easy to see that these criteria correspond to `L` permutations of `R` items. When `L=R`, this is the # permutations of `⍳R`, (↔ `!R`), and when `L<R`, this is the # `L`-permutations, also

called the falling factorial (!⍠(-L))R. See case **110** below.

As a side note, the above examples reveal one of the many insights the Twelvefold Way provides into Combinatorial Algorithms. Previously, you might not have seen any connection between the algorithms for Combinations and Permutations, but, as the above examples show, they are closely related in that they differ only in the use of **labeled** vs. **unlabeled** balls, both in labeled boxes with at most one ball per box.

Syntax

One way of defining the syntax for such a Combinatorial Primitive is as a monadic operator DoubleShriek (‼ – U+203C) deriving a monadic function whose operand is the function selector (V) and whose right argument is the # balls (L) & boxes (R) as in: V‼L R. This symbol can be entered from the default keyboard layout with Alt-’k’ or Ctrl-’k’, depending upon your choice of keyboard layouts.

The derived function from the operator is monadic only and mixed (i.e., not scalar). When generating results, some are sensitive to ⎕IO, and some may be nested as not all items need be of the same rank and shape.

Operand and Argument

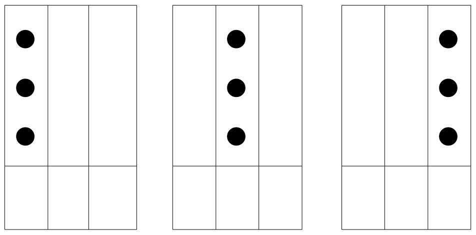
The (left) operand is limited to non-negative integer scalars, and the (right) argument is a non-negative (possibly multi-precision) integer scalar or one- or two-element vector. If there is only one element in the argument, it is treated as if it were duplicated as in (L R)←2⍴R. Various Combinatorial Algorithms have been extended to other datatypes (e.g., Hypercomplex numbers[9]), but I’ve chosen not to do that for the initial design.

Labeled vs. Unlabeled

Boxes

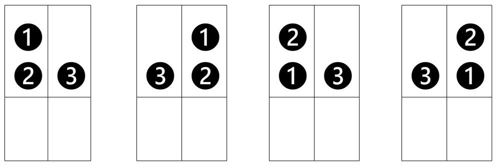
For most cases, the boxes are the columns of the result. Two or more **labeled** boxes may hold identical content, but because the boxes are labeled, they are considered distinct. On the other hand, **unlabeled** boxes with identical content are indistinguishable.

For example, the following (partial) configurations of 3 unlabeled balls (●●●) in 3 unlabeled boxes



are all considered equivalent and are counted only once because the boxes are unlabeled.

Similarly, the following (partial) configurations of 3 labeled balls (❶❷❸) in 2 unlabeled boxes



are also all considered equivalent and are counted only once, again because the boxes are unlabeled.

Note that the order of the (labeled) balls within a box is ignored which means that even if the boxes were labeled, the first and third configurations above are equivalent, as are the second and fourth.

Correspondingly, all other things being equal, a result for unlabeled boxes can be expanded to a result for labeled boxes by labeling the unlabeled boxes in all possible permutations. In particular, in cases **002** and **012** as well as **102** and **112** both pairs share this property. Moreover, because cases **102** and **112** involve **labeled** balls their counts differ exactly by a factor of `!R`; cases **002** and **012** use **unlabeled** balls so their counts differ in a more complicated way.

Balls

In a similar manner, the counts and generations for combinations (**010**) and permutations (**110**) differ by a factor of `!L`, this time because of the **balls**: one is **unlabeled** and the other **labeled**. That is, the count for `L` combinations of `R` items is

```
L!R ↔ (!R)÷(!R-L)×!L
```

and the count for `L` permutations of `R` items is

```
(!⍨(-L)) R ↔ (!R)÷!R-L
```

Of course, when `L=R`, the permutation count is the familiar `!R`.

See the examples above on the differences between labeled and unlabeled balls.

Finally, the generations of permutations and combinations are related as follows:

```
a←010  1‼L  R      L combinations of R items
b←110  1‼L         L permutations of L items (L=R)
c←110  1‼L  R      L permutations of R items (L<R)
       c≡,[⎕IO+0 1] a[;b]
```

Counting Partitions of a Set

The counts in three of the cases below (**101**, **102**, and **112**) are dependent on the Stirling numbers of the 2nd kind. They satisfy the following recurrence relation defined on integer `n≥0` and `k≥0`:

```
S(0,0) = 1
S(0,n) = S(n,0) = 0                      for n>0
S(n,k) = k × S(n-1,k) + S(n-1,k-1)
```

and, for the purposes of this paper, is implemented in the dyadic APL function `SN2`. These numbers are used in the field of combinatorics and the study of partitions of a set[5].

As a side note, as the count for case **102** (the # of partitions of the set `{⍳L}` into exactly `R` parts) is `L SN2 R`, the Stirling numbers of the 2nd kind may be obtained from `102‼L R` in case they are needed in another context.

For example:

```
      ∘.SN2⍨ 0..10 ⍝ ↔ 102‼¨∘.,⍨0..10
   │ 0 1   2    3     4     5     6    7   8  9 10
───┼──────────────────────────────────────────────
 0 │ 1 0   0    0     0     0     0    0   0  0  0
 1 │ 0 1   0    0     0     0     0    0   0  0  0
 2 │ 0 1   1    0     0     0     0    0   0  0  0
 3 │ 0 1   3    1     0     0     0    0   0  0  0
 4 │ 0 1   7    6     1     0     0    0   0  0  0
 5 │ 0 1  15   25    10     1     0    0   0  0  0
 6 │ 0 1  31   90    65    15     1    0   0  0  0
 7 │ 0 1  63  301   350   140    21    1   0  0  0
 8 │ 0 1 127  966  1701  1050   266   28   1  0  0
 9 │ 0 1 255 3025  7770  6951  2646  462  36  1  0
10 │ 0 1 511 9330 34105 42525 22827 5880 750 45  1
```

Counting Partitions of a Number

The counts in two of the cases below (**001** and **002**) are dependent on the following recurrence relation for partitions defined on integer `n` and `k`:

```
P(0,0) = 1
P(n,k) = 0                          for n≤0 or k≤0
P(n,k) = P(n-k,k) + P(n-1,k-1)
```

and, for the purposes of this paper, is implemented in the dyadic APL function `PN`. These numbers are used in the field of combinatorics and the study of partitions of a number[7].

For example:

```
      ∘.PN⍨ 0..10 ⍝ ←→ 2‼¨∘.,⍨0..10
   │ 0 1 2 3 4 5 6 7 8 9 10
───┼────────────────────────
 0 │ 1 0 0 0 0 0 0 0 0 0  0
 1 │ 0 1 0 0 0 0 0 0 0 0  0
 2 │ 0 1 1 0 0 0 0 0 0 0  0
 3 │ 0 1 1 1 0 0 0 0 0 0  0
 4 │ 0 1 2 1 1 0 0 0 0 0  0
 5 │ 0 1 2 2 1 1 0 0 0 0  0
 6 │ 0 1 3 3 2 1 1 0 0 0  0
 7 │ 0 1 3 4 3 2 1 1 0 0  0
 8 │ 0 1 4 5 5 3 2 1 1 0  0
 9 │ 0 1 4 7 6 5 3 2 1 1  0
10 │ 0 1 5 8 9 7 5 3 2 1  1
```

If you have seen the movie “The Man Who Knew Infinity” (2015) (about the life and academic career of the brilliant Indian mathematician Srinivasa Ramanujan), you may recall that at one point it focuses on the problem of calculating `p(200)` – the number of Partitions of the number `200` into at most `200` parts. This number can be calculated by

```
      1‼200
3972999029388
```

in a few hundred-thousandths of a second.

The Twelve Algorithms

The algorithms are presented in lexicographic order of the function selector (**000** through **112**). The examples use the abbreviation #bpb = “# balls per box”. All ⎕IO-sensitive results are in origin 1. The order of the rows in matrix results and the items in nested array results is unspecified.

Case 000: (Back to FS Table)

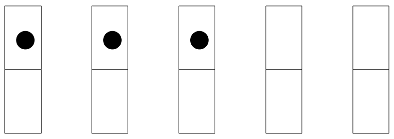
- L unlabeled balls into R unlabeled boxes, at most one per box
- Not ⎕IO-sensitive
- Result is a Boolean matrix.

This case is trivial: If L>R, then there is no answer, or more accurately, the result is an empty matrix of shape 0 R. If L≤R, then the result is a one-row matrix with L leading 1s and the rest 0s. Combining these two cases yields a result of ((L≤R) R)ρR↑Lρ1. It is identical to case **100**.

The count for this function is L≤R.

For example:

If we have 3 unlabeled balls (●●●) and 5 unlabeled boxes with at most one ball per box, there is only 1 (↔ 3≤5) way to meet these criteria:



The diagram above corresponds to

```
      000 1‼3 5
1 1 1 0 0

      ⍝ L pigeons into R holes
      ⍝ Unlabeled balls & boxes, ≤1 #bpb
      000 1‼4 5
1 1 1 1 0
```

Case 001: (Back to FS Table)

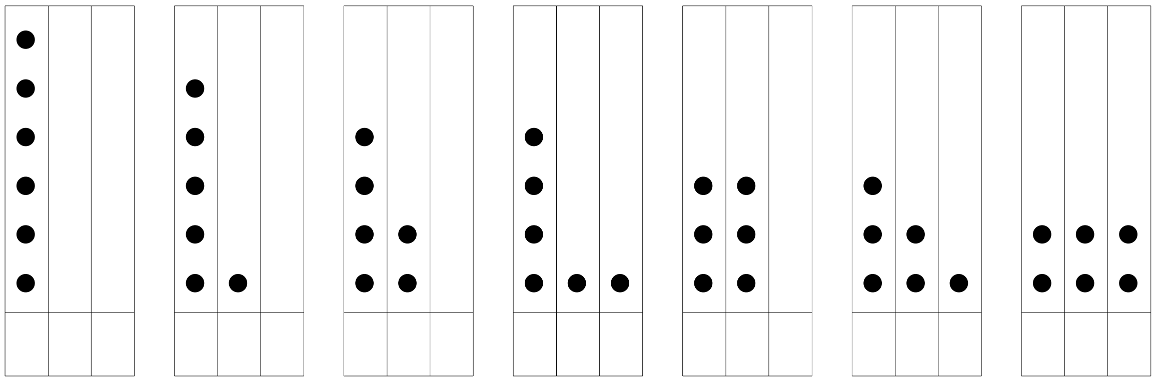
- L unlabeled balls, R unlabeled boxes, any # of balls per box
- Not ⎕IO-sensitive
- Result is a nested vector of integer vectors.

This case produces the partitions of the number L into at most R parts.

The count for this function is (L+R)PN R where PN is described above. As shown in Wikipedia[11], (L+R)PN R ↔ +/L PN¨0..R.

For example:

If we have 6 unlabeled balls (●●●●●●) and 3 unlabeled boxes with any # of balls per box, there are 7 (↔ (6+3)PN 3) ways to meet these criteria:



The diagram above corresponds to the nested array

```
      ⍪001 1‼6 3
 6
 5 1
 4 2
 4 1 1
 3 3
 3 2 1
 2 2 2
```

```
      ⍝ Partitions of L into at most R parts
      ⍝ Unlabeled balls & boxes, any #bpb
      ⍪001 1‼5 5
 5
 4 1
 3 2
 3 1 1
 2 2 1
 2 1 1 1
 1 1 1 1 1
      ⍪001 1‼5 4
 5
 4 1
 3 2
 3 1 1
 2 2 1
 2 1 1 1
      ⍪001 1‼5 3
 5
 4 1
 3 2
 3 1 1
 2 2 1
      ⍪001 1‼5 2
 5
 4 1
 3 2
      ⍪001 1‼5 1
 5
```

Because partitions of `L` into `R` non-negative parts (**001**) is the same as partitions of `L+R` into `R` positive parts (**002**), these cases are related by the following identity:

```
001 1‼L R ↔ (⊂[⎕IO+1] ¯1+002 1‼(L+R) R)~¨0
```

Case 002: (Back to FS Table)

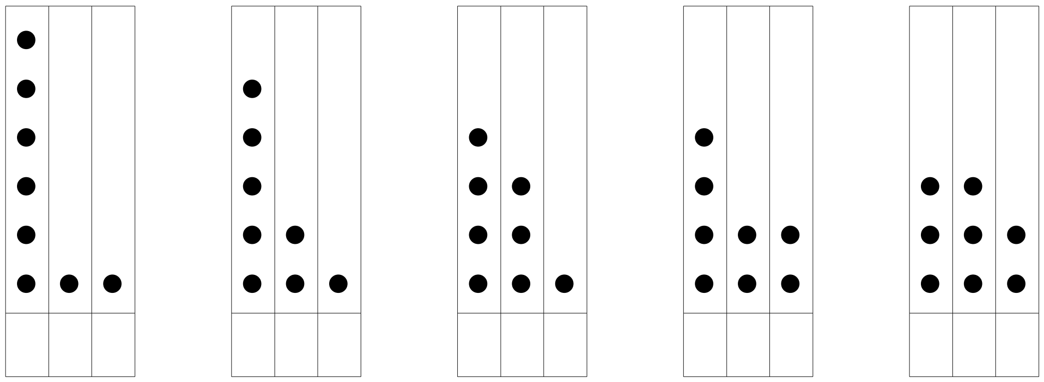
- `L` unlabeled balls, `R` unlabeled boxes, at least one ball per box
- Not `⎕IO`-sensitive
- Result is an integer matrix.

This case produces the partitions of the number `L` into exactly `R` parts.

The count for this function is `L PN R` where `PN` is described above.

For example:

If we have `8` unlabeled balls (●●●●●●●●) and `3` unlabeled boxes with at least one ball per box, there are `5` (`↔ 8 PN 3`) ways to meet these criteria:



The diagram above corresponds to

```
      002 1‼8 3
6 1 1
5 2 1
4 3 1
4 2 2
3 3 2
```

```
      ⍝ Partitions of L into R parts
      ⍝ Unlabeled balls & boxes, ≥1 #bpb
      002 1‼5 5
1 1 1 1 1
      002 1‼5 4
2 1 1 1
      002 1‼5 3
3 1 1
2 2 1
      002 1‼5 2
4 1
3 2
      002 1‼5 1
5
```

Because partitions of L into R non-negative parts (**001**) is the same as partitions of L+R into R positive parts (**002**), these cases are related by the following identity (after sorting the rows):

```
002 1‼L R ↔ ⊃1+R↑¨001 1‼(0⌈L-R) R
```

Case 010: (Back to FS Table)

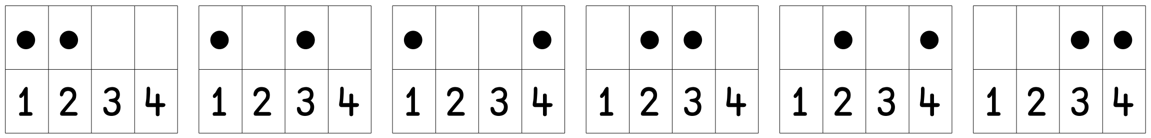
- L unlabeled balls, R labeled boxes, at most one ball per box
- Sensitive to ⎕IO
- Result is an integer matrix.

This case produces the L combinations of R items.

The count for this function is L!R.

For example:

If we have 2 unlabeled balls (●●) and 4 labeled boxes (1234) with at most one ball per box, there are 6 (↔ 2!4) ways to meet these criteria:



and, in general, it's easy to see that this case solves the familiar problem of L combinations of R items.

The diagram above corresponds to

```
      010 1‼2 4
1 2
1 3
1 4
2 3
2 4
3 4
      ⍝ Combinations
      ⍝ Unlabeled balls, labeled boxes, ≤1 #bpb
      3!5
10
```

```
      010‼3 5
10
      010 0‼3 5
10
      ⍴010 1‼3 5
10 3
      010 1‼3 5
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

In general, this case is related to that of Multisets (**011**) and Compositions (**012**) via the following identities:

```
010 1‼L R ↔ (011 1‼L,R-L-1)+[⎕IO+1] 0..L-1
011 1‼L R ↔ (010 1‼L,L+R-1)-[⎕IO+1] 0..L-1

010 1‼L R ↔ +\0 ¯1↓012 1‼⍠1 R L+1
012 1‼L R ↔ ¯2-\(010 1‼⍠1 R L-1),L
```

where `‼⍠1` uses the Variant operator `⍠` to evaluate `‼` in origin `1`.

Case 011: (Back to FS Table)

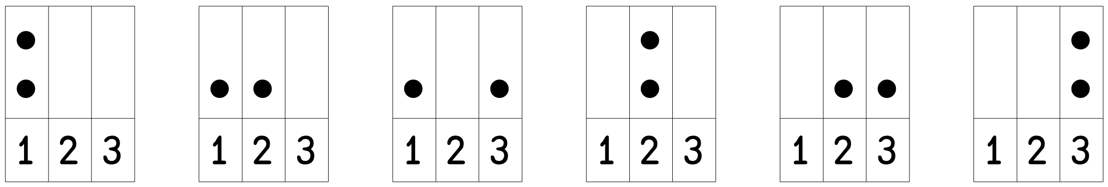
- L unlabeled balls, R labeled boxes, any # balls per box
- Sensitive to ⎕IO
- Result is an integer matrix.

This case produces L multicombinations of R items. A multicombination is a collection of multisets[10] (sets which may contain repeated elements) according to certain criteria. In particular, it produces a matrix whose rows are multisets of length L, from the values ⍳R.

The count for this function is L!L+R-1.

For example:

If we have 2 unlabeled balls (●●) and 3 labeled boxes (123) with any # of balls per box, there are 6 (↔ 2!2+3-1) ways to meet these criteria:



The diagram above corresponds to:

```
      011 1‼2 3
1 1
1 2
1 3
2 2
2 3
3 3
```

```
      ⍝ L Multicombinations of R items
      ⍝ Unlabeled balls, labeled boxes, any #bpb
      011 1‼3 3
1 1 1
1 1 2
1 1 3
1 2 2
1 2 3
1 3 3
2 2 2
2 2 3
2 3 3
3 3 3
      011 1‼3 2
1 1 1
1 1 2
1 2 2
2 2 2
      011 1‼3 1
1 1 1
```

In general, this case is related to that of Combinations (**010**) via the following identities:

```
010 1‼L R ↔ (011 1‼L,R-L-1)+[⎕IO+1] 0..L-1
011 1‼L R ↔ (010 1‼L,L+R-1)-[⎕IO+1] 0..L-1
```

Case 012: (Back to FS Table)

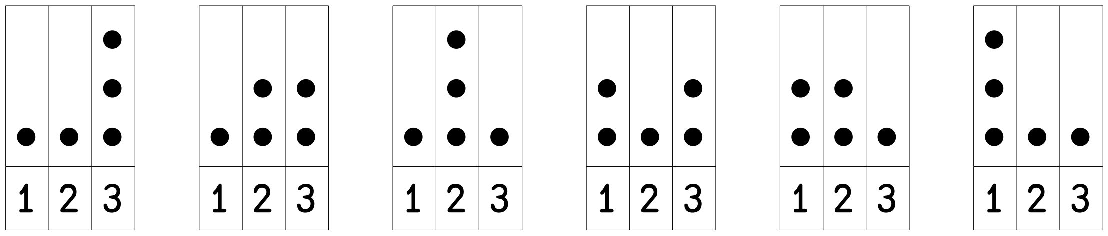
- L unlabeled balls, R labeled boxes, at least one ball per box
- Not ⎕IO-sensitive
- Result is an integer matrix.

This case produces compositions of the number L into R parts. A composition is a way of representing a number as the sum of all positive integers, in this case it's a way of representing L as the sum of R positive integers. It can also be thought of as a partition of L into R ordered parts.

The count for this function is `(L-R)!L-1`.

For example:

If we have 5 unlabeled balls (●●●●●) and 3 labeled boxes (123) with at least one ball per box, there are 6 (↔ `(5-3)!5-1`) ways to meet these criteria:



The diagram above corresponds to

```
      012 1‼5 3
1 1 3
1 2 2
1 3 1
2 1 2
2 2 1
3 1 1
```

```
      ⍝ Compositions of L into R parts
      ⍝ Unlabeled balls, labeled boxes, ≥1 #bpb
      012 1‼5 5
1 1 1 1 1
      012 1‼5 4
1 1 1 2
1 1 2 1
1 2 1 1
2 1 1 1
      012 1‼5 3
1 1 3
1 2 2
1 3 1
2 1 2
2 2 1
3 1 1
      012 1‼5 2
1 4
2 3
3 2
4 1
      012 1‼5 1
5
```

In general, because the counts of both compositions (**012**) and combinations (**010**) is a binomial coefficient, there might be a mapping between the two, and indeed there is, as seen by the following identities:

```
010 1‼L R ↔ +\0 ¯1↓012 1‼⍠1 R L+1
012 1‼L R ↔ ¯2-\(010 1‼⍠1 R L-1),L
```

where `‼⍠1` uses the Variant operator `⍠` to evaluate `‼` in origin `1`.

Case 100: (Back to FS Table)

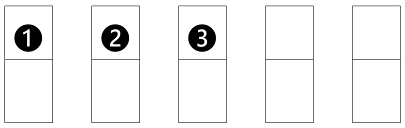
- L labeled balls, R unlabeled boxes, at most one ball per box
- Not ⎕IO-sensitive
- Result is a Boolean matrix.

This case is trivial: If L>R, then there is no answer, or more accurately, the result is an empty matrix of shape 0 R. If L≤R, then the result is a one-row matrix with L leading 1s and the rest 0s. Combining these two cases yields a result of ((L≤R) R)⍴R↑L⍴1. It is identical to case **000**.

The count for this function is L≤R.

For example:

If we have 3 labeled balls (❶❷❸) and 5 unlabeled boxes with at most one ball per box, there is only 1 (↔ 3≤5) way to meet these criteria:



The diagram above corresponds to

```
      100 1‼3 5
1 1 1 0 0
      ⍝ L pigeons into R holes
      ⍝ Labeled balls, unlabeled boxes, ≤1 #bpb
      100 1‼4 5
1 1 1 1 0
```

Case 101: (Back to FS Table)

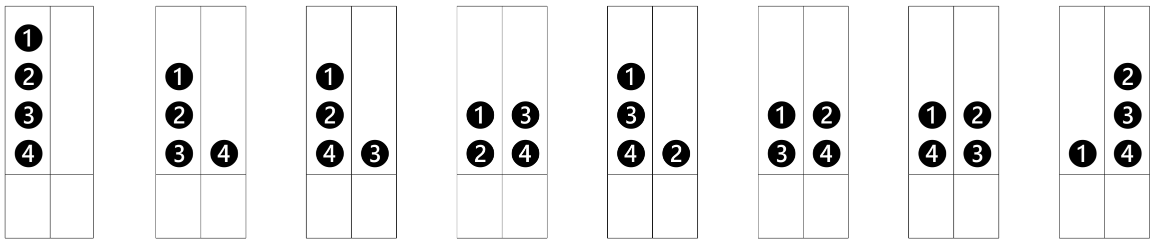
- `L` labeled balls, `R` unlabeled boxes, any # balls per box
- Sensitive to `⎕IO`
- Result is a nested vector of nested integer vectors.

This case produces partitions of the set `{⍳L}` into at most `R` parts. Partitioning a set is different from partitioning a number in that the former subdivides the set into non-empty disjoint subsets.

The count for this function is `+/L SN2¨0..R` where `SN2` is described above.

For example:

If we have `4` labeled balls (❶❷❸❹) and `2` unlabeled boxes with any # of balls per box, there are `8` (`↔ +/4 SN2¨0..2 ↔ +/0 1 7`) ways to meet these criteria:



The diagram above corresponds to the nested array

```
      ⍪101 1‼4 2
  1 2 3 4
  1 2 3  4
  1 2 4  3
  1 2  3 4
  1 3 4  2
  1 3  2 4
  1 4  2 3
  1  2 3 4
```

```
      ⍝ Partitions of {⍳L} into at most R parts
      ⍝ Labeled balls, unlabeled boxes, any #bpb
      101 0‼4 4
15
      101 0‼4 3
14
      101 0‼4 2
8
      101 0‼4 1
1
      101 0‼4 0
0
```

The first column in the following table serves to number the rows and is referenced in **102**. The second column illustrates the result of `101 1‼4 4` as a nested array with `15` elements. The third column includes visual separators to make it clearer where one subset stops and another begins:

1	1 2 3 4	1 2 3 4
2	1 2 3 4	1 2 3|4
3	1 2 4 3	1 2 4|3
4	1 2 3 4	1 2|3 4
5	1 2 3 4	1 2|3|4
6	1 3 4 2	1 3 4|2
7	1 3 2 4	1 3|2 4
8	1 3 2 4	1 3|2|4
9	1 4 2 3	1 4|2 3
10	1 2 3 4	1|2 3 4
11	1 2 3 4	1|2 3|4
12	1 4 2 3	1 4|2|3
13	1 2 4 3	1|2 4|3
14	1 2 3 4	1|2|3 4
15	1 2 3 4	1|2|3|4

See the next case for identities that relate **101** and **102**.

Case 102: (Back to FS Table)

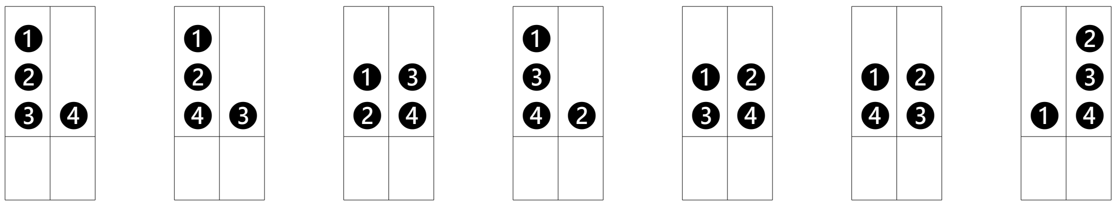
- L labeled balls, R unlabeled boxes, at least one ball per box
- Sensitive to ⎕IO
- Result is a nested vector of nested integer vectors.

This case produces partitions of the set {⍳L} into exactly R parts. As such, it produces a subset of **101**, limiting the result to just those rows with L subsets.

The count for this function is L SN2 R where SN2 is described above.

For example:

If we have 4 labeled balls (❶❷❸❹) and 2 unlabeled boxes with at least one ball per box, there are 7 (↔ 4 SN2 2) ways to meet these criteria:



The diagram above corresponds to the nested array

```
    ⍪102 1‼4 2
1 2 3  4
1 2 4  3
1 2  3 4
1 3 4  2
1 3  2 4
1 4  2 3
1  2 3 4
```

```
⍝ Partitions of {⍳L} into R parts
⍝ Labeled balls, unlabeled boxes, ≥1 #bpb
⍝ The number to the right in parens
⍝    represent the corresponding row from
⍝    the table in case 101.
```

```
    ⍪102 1‼4 4
1  2  3  4      (15)
    ⍪102 1‼4 3
1 2  3  4       (5)
1 3  2  4       (8)
1  2 3  4       (11)
1 4  2  3       (12)
1  2 4  3       (13)
1  2  3 4       (14)
```

```
    ⍪102 1‼4 2
1 2 3  4        (2)
1 2 4  3        (3)
1 2  3 4        (4)
1 3 4  2        (6)
1 3  2 4        (7)
1 4  2 3        (9)
1  2 3 4        (10)
    ⍪102 1‼4 1
1 2 3 4         (1)
    ⍪102 1‼4 0
```

In general, this case is related to **101** through the following identities (after sorting the items):

```
101 1‼L R ↔ ⊃,/102 1‼¨L,¨0..R
102 1‼L R ↔ R {(⍺=≢¨⍵)/⍵} 101 1‼L R
```

and is related to **112** through the following identities:

```
102 1‼L R ↔ {(2≢/¯1,(⊂¨⍋¨⍵)⌷¨⍵)/⍵} 112 1‼L R
a←⊃102 1‼L R
b← 110 1‼R R
112 1‼L R ↔ ,⊂[⎕IO+2] a[;b]
```

Case 110: (Back to FS Table)

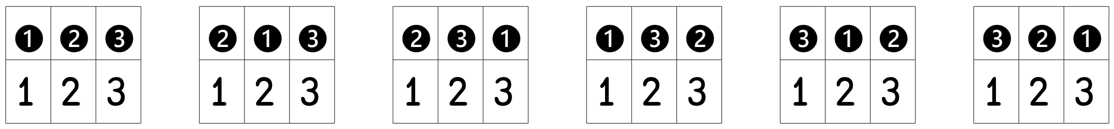
- `L` labeled balls, `R` labeled boxes, at most one ball per box
- Sensitive to `⎕IO`
- Result is an integer matrix.

This case produces `L`-permutations (also called Partial Permutations or Sequences Without Repetition) of `R` items, where when `L=R` produces the familiar permutations `!R`. The length of each permutation returned is always `L`.

The count for this function is `(!⍠(-L))R` where `(!⍠L)R` calculates the rising or falling factorial.

For example:

If we have `3` labeled balls (❶❷❸) and `3` labeled boxes (`123`) with at most one ball per box, there are `6` (`↔ (!⍠¯3)3 ↔ 3×2×1`) ways to meet these criteria:



The diagram above corresponds to

```
      110 1‼3
1  2  3
2  1  3
2  3  1
1  3  2
3  1  2
3  2  1
```

```
      ⍝ Permutations of length L of R items
      ⍝ Labeled balls & boxes, ≤1 #bpb
      !3
6
      110‼3
6
      110 0‼3
6
      ⍴110 1‼3
6 3
      110 1‼3
1 2 3
2 1 3
2 3 1
1 3 2
3 1 2
3 2 1
      110 1‼2 3
1 2
2 1
1 3
3 1
2 3
3 2
      110 1‼1 3
1
2
3
```

A function to calculate the permutations of `R` items could be defined as

```
      perm←{110 1‼⍵}
```

Case 111: (Back to FS Table)

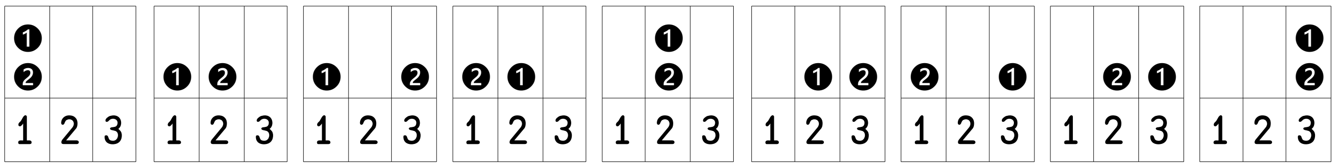
- L labeled balls, R labeled boxes, any # balls per box
- Sensitive to ⎕IO
- Result is an integer matrix.

This case produces L-tuples of R items. That is, all length L vectors with all possibilities of R items in each position, R*L rows all together.

The count for this function is R^L (↔ R*L).

For example:

If we have 2 labeled balls (❶❷) and 3 labeled boxes (123) with any # of balls per box, there are 9 (↔ 3*2) ways to meet these criteria:



The diagram above corresponds to

```
      110 1‼2 3
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

```
      ⍝ L-tuples of R items
      ⍝ Labeled balls & boxes, any #bpb
      111 0‼3 2
8
      111 1‼3 2
1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2
```

Case 112: (Back to FS Table)

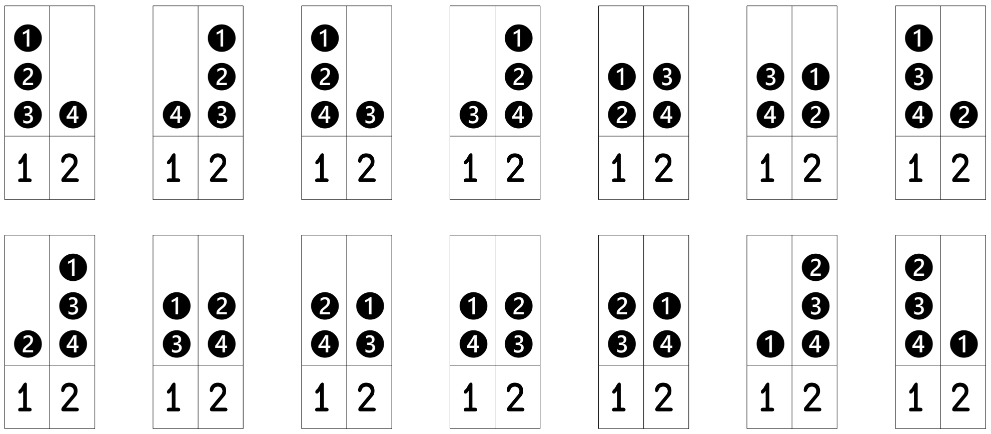
- L labeled balls, R labeled boxes, at least one ball per box
- Sensitive to ⎕IO
- Result is a nested vector of nested integer vectors.

This case produces partitions of the set {⍳L} into R **ordered** parts. Essentially, this case is the same as **102**, except that the order of the elements is important so that there are more results by a factor of !R. For example, the 3-subset result of 1 2|3|4 for **102** is expanded to !4 (↔ 24) 3-subsets by permuting the values 1 2 3 4 in 24 ways.

The count for this function is (!R)×L SN2 R where SN2 is described above.

For example:

If we have 4 labeled balls (❶❷❸❹) and 2 labeled boxes (12) with at least one ball per box, there are 14 (↔ (!2)×4 SN2 2 ↔ 2×7) ways to meet these criteria:



The diagram above corresponds to the nested array

```
      ⍎112 1‼4 2
1 2 3  4
4  1 2 3
1 2 4  3
3  1 2 4
1 2  3 4
3 4  1 2
1 3 4  2
2  1 3 4
1 3  2 4
2 4  1 3
1 4  2 3
2 3  1 4
1  2 3 4
2 3 4  1
      ⍝ Partitions of the set {⍳L} into
      ⍝   R ordered parts
      ⍝ Labeled balls & boxes, any #bpb
      ⍎112 1‼3 3
1  2  3
2  1  3
2  3  1
1  3  2
3  1  2
3  2  1
      ⍎112 1‼3 2
1 2  3
3  1 2
1 3  2
2  1 3
1  2 3
2 3  1
      ⍎112 1‼3 1
1 2 3
```

In general, this case is equivalent to calculating the unlabeled boxes (**102**) and then permuting the items from that result as in

```
a←⊃102 1‼L R
b← 110 1‼R
112 1‼L R ↔ ,⊂[⎕IO+2] a[;b]
```

or vice-versa

```
102 1‼L R ↔ {(2≢/¯1,(⊂⍤⍋⍤ω)⌷⍤ω)/ω} 112 1‼R R
```

Summary of Related Algorithms

While the Twelvefold way consists of twelve Combinatorial Algorithms, half of these algorithms can be defined in terms of the other half as summarized in the following identities.

Note that some of the identities might not “match” identically from side to side (as in `a≡b`) because their items are in a different order. However, they do match when the order of the results is ignored, as in `a[⍋a]≡b[⍋b]`, or in the case of two matrices whose rows are in a different order, `a[⍋a;]≡b[⍋b;]`.

000 ↔ **100**: Pigeons in holes

```
000 1‼L R ↔ 100 1‼L R
```

001 ↔ **002**: Partitions of the number L into ≤R parts vs. into R parts

```
001 1‼L R ↔ (⊂[⎕IO+1] ¯1+002 1‼(L+R) R)~¨0
002 1‼L R ↔ ⊃1+R↑¨001 1‼(0⌈L-R) R
```

010 ↔ 011: Combinations vs. Multisets

```
010 1‼L R ↔ (011 1‼L,R-L-1)+[⎕IO+1] 0..L-1
011 1‼L R ↔ (010 1‼L,L+R-1)-[⎕IO+1] 0..L-1
```

010 ↔ 012: Combinations vs. Compositions

```
010 1‼L R ↔ +\0 ¯1↓012 1‼⍠1 R L+1    (‼⍠1 ↔ ‼ in origin 1)
012 1‼L R ↔ ¯2-\(010 1‼⍠1 R L-1),L
```

101 ↔ 102: Partitions of {⍳L} into ≤R parts vs. into R parts

```
101 1‼L R ↔ ⊃,/102 1‼L,¨0..R
102 1‼L R ↔ R {(⍺=≢¨⍵)/⍵} 101 1‼L R
```

102 ↔ 112: Partitions of {⍳L} into R parts vs. into R ordered parts

```
102 1‼L R ↔ {(2≢/¯1,(⊂¨⍋¨⍵)⌷¨⍵)/⍵} 112 1‼L R
a←⊃102 1‼L R
b← 110 1‼R
112 1‼L R ↔ ,⊂[⎕IO+2] a[;b]
```

110 ↔ 110: Permutations (L<R case in terms of the L=R case)

```
110 1‼L R ↔ ,[⎕IO+0 1](010 1‼L R)[;110 1‼L]
```

This means that out of the original twelve algorithms there are only six independent ones of which a representative sample is

000	**L pigeons into R holes**
002	**Partitions of the number L into R parts**
010	**L Combinations of R items**
102	**Partitions of the set {⍳L} into R parts**
110	**L Permutations of R items for L=R only**, and
111	**L Tuples of R items**.

Similarities in The FS Table

There is a similarity in the FS Table between the two meanings in this paper of the word partitions: of a number and of a set. There are five cells about partitions plus one for compositions which are really partitions of L into R ordered parts, so that's six partitions; three of the number L, and three of the set {⍳L}. As the table below shows, the two meanings differ only in that three partitions of the number L all use **unlabeled** balls and the three partitions of the set {⍳L} all use **labeled** balls:

Partitions of the ...			
number L into ...		**set {⍳L} into ...**	
at most R parts	(**001**)	at most R parts	(**101**)
R parts	(**002**)	R parts	(**102**)
R ordered parts	(**012**)	R ordered parts	(**112**)

Implementing the Algorithms

The algorithms described here would normally be written in a lower level language (e.g., C) and then be accessible from APL through this primitive without the need for a separate (and less efficient) library of APL routines.

There are many good sources for these algorithms in the literature as well as on the Internet. Here's a first cut as to the algorithms I plan to use – please feel free to suggest others:

000: L Pigeons into R holes

Trivial – implement the following APL statement:

```
        Z←((L≤R) R)ρR↑Lρ1
```

001: Partitions of the number L into at most R parts

Knuth's TAoCP, Vol 4A, p. 392, Algorithm P.

002: **Partitions of the number `L` into `R` parts**

Knuth’s TAoCP, Vol 4A, p. 392, Algorithm H.

010: **`L` Combinations of `R` items**

Knuth's TAoCP, Vol 4A, p. 359, Algorithm T.

011: **`L` Multisets of `R` items**

Implement the following APL statement:
```
        Z←(L comb L+R-1)-[1+⎕IO] 0..L-1
```
where `L comb R` generates all combinations of length `L` of `R` items (`comb ↔ {010 1‼α ω}`.

012: **Compositions of `L` into `R` parts**

Implement the following APL statements:
```
        ⎕IO←1
        Z←¯2-\((R-1) comb L-1),L
```
where `L comb R` generates all combinations of length `L` of `R` items (`comb ↔ {010 1‼α ω}`.

100: **`L` Pigeons into `R` holes**

Trivial – implement the following APL statement:
```
        Z←((L≤R) R)ρR↑Lρ1
```

101: **Partitions of the set `{⍳L}` into at most `R` parts**

Knuth's TAoCP, Vol 4A, p. 416, Algorithm H as `enum101sub` modified to include a specific test #1 as to which results are accepted, along with an implementation of the following APL statements:

```
        b←⊂[1+⎕IO] L enum101sub R
        Z←(1+(⊂¨⍋¨b)⌷¨b)⊂¨⍋¨b
```

102: Partitions of the set {⍳L} into R parts

Knuth's TAoCP, Vol 4A, p. 416, Algorithm H as `enum102sub` modified to include a specific test #2 as to which results are accepted, along with an implementation of the following APL statements:

```
        b←⊂[1+⎕IO] L enum102sub R
        Z←(1+(⊂¨⍋¨b)⌷¨b)⊂¨⍋¨b
```

110: L Permutations of R items

Knuth's TAoCP, Vol 4A, p. 322, Algorithm P for the `L=R` case of permutations. The `L<R` case is covered by implementing the following APL statement:

```
        Z←,[⎕IO+0 1](L comb R)[;perm L]
```

where `L comb R` generates all combinations of length `L` of `R` items (`comb ↔ {010 1‼⍺ ⍵}`, and `perm L` is from the above mentioned Algorithm P.

111: L Tuples of R items

Trivial – implement the following APL statement:

```
        Z←((R*L)L)⍴∊∘.,/L⍴⊂⍪⍳R
```

112: Partitions of the set {⍳L} into R ordered parts

Implement the following APL statements:

```
        a←⊃102 1‼L R
        b← 110 1‼R
        Z←,⊂[⎕IO+2] a[;b]
```

Open Questions

- The choice of the "double shriek" (‼ – U+203C) for the symbol is arbitrary – suggestions? Chi (χ) (U+03C7) for "chombinatorics"? Get out your copy of Unicode and start searching. The font CODE2003[6] is a good start for viewing the entire the Basic Multilingual Plane.
- One reviewer suggested that we could change the syntax to a single function (call it γ) and invoke it as in `(L V)γR`. Any thoughts?
- Another reviewer suggested that we could change the syntax to a single function (call it δ) and invoke it as in `VδL R`., where `L R` is the usual # balls & # boxes which can be extended to multiple arguments as both a two-column integer array as well as a nested array of two-element integer vectors. Any thoughts?

Future Work

At a later time, we might add a feature to `V[2]` to specify not only count vs. generation but, when generating, it can also specify the order of the items in the result as unspecified, lexicographic, gray code, etc., assuming the requested order applies to the selected generation function. For example, `V[2]=0` for the count, `V[2]=1` for generate in unspecified order, `V[2]=2` for lexicographic order, `V[2]=3` for gray code order, etc. For the moment, when generating, the order is unspecified.

Also, the function selector could specify in a new element `V[3]` that the result is to be returned one-by-one starting with a specified item so as to avoid returning the entire result at once. That is, with `V[2]≥1`, `V[3]` would range from `0` to the count of that particular function selector and its arguments, where `V[3]=0` means return the full result, and otherwise, `V[3]=N` means return the `N`th corresponding generated value in the order (if any) as specified by `V[2]`. I realize that some next-generation algorithms require as input the previous generation, so this design would need to be modified to accommodate that need.

And finally, in case Twelvefold Ways isn't enough, there is also a Thirtyfold Way paper[4]!

Conclusions

- Rota's amazing Twelvefold Way of consolidating numerous Combinatorial Algorithms through the unifying concept of balls in boxes into a single organizational framework is presented and each algorithm is discussed in detail with examples.
- This organizational framework is ideally suited for implementation in APL for both counting and generation by referencing the individual algorithms using a function selector as the (left) operand to a new monadic primitive operator.
- Insight into these Combinatorial Algorithms is gained when viewed from the perspective of the Twelvefold Way. To wit:
  - The relationships among the algorithms is made clearer when comparing their APL versions, especially through identities.
  - The algorithms are shown to have considerable dependence amongst themselves as shown through APL identities.
  - Interesting similarities within the function selector table are identified and are worthy of further investigation.
- Thanks to the work of D. E. Knuth in his TAoCP Vol 4A, each of the twelve ways has a high quality algorithm behind it.
- Finally, APL programmers need no longer search for the fastest APL program to generate any of several Combinatorial Counts or Generations as the fastest way is now available primitively.

Acknowledgments

No paper is written in isolation, and this paper is no exception. I'd like to thank David Liebtag, Roy Sykes, Norman Thomson, Jim Brown, Roger Hui, and Michael Turniansky for their helpful advice and suggestions.

Online Version

This paper is an ongoing effort and can be out-of-date the next day. To find the most recent version, go to http://sudleyplace.com/APL/ and look for the title of this paper on that page. There is also a workspace online (http://www.nars2000.org/download/workspaces/) that models this primitive with the user-defined operator `dem` used in place of the double shriek ‼. In your browser, right click on the workspace name and choose "Save Link As..." or "Save target as" to download the workspace to your local hard drive and then `)LOAD` it from there from within NARS2000.

References

1. Stanley, Richard P., "Enumerative Combinatorics", Volume 1, 2nd edition, Cambridge University Press, p. 71, ISBN 0-521-66351-2
2. Wikipedia, "Twelvefold Way", https://en.wikipedia.org/wiki/Twelvefold_way
3. Knuth, Donald E., "The Art of Computer Programming", Addison Wesley, Volume 4A, Combinatorial Algorithms, p. 390, ISBN 0-201-89685-0
4. Proctor, Robert A., "Let's Expand Rota's Twelvefold Way For Counting Partitions", http://www.math.unc.edu/Faculty/rap/30FoldWay.pdf
5. Wikipedia, "Stirling numbers of the second kind", https://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind

6. CODE2003, http://www.fontspace.com/st-gigafont-typefaces/code2003
7. Wikipedia, “Partitions: Restricted part size or number of parts”, https://en.wikipedia.org/wiki/Partition_(number_theory)#Restricted_part_size_or_number_of_parts
8. NARS2000 Wiki, “Primes”, http://wiki.nars2000.org/index.php/Primes
9. “Hypercomplex Numbers in APL”, http://www.sudleyplace.com/APL/Hypercomplex%20Numbers%20in%20APL.pdf
10. NARS2000 Wiki, “Multisets”, http://wiki.nars2000.org/index.php/Multisets
11. Wikipedia, “TwelveFold Way”, https://en.wikipedia.org/wiki/Twelvefold_way#case_fnx