# NUMERAL SYSTEM USING AN ARRAY OF COMBINATIONS

by: Ihor Jakowec
March 2007

Definitions:

Here, a combination listing is defined to be: a sequence of a constant (yet, finite), number of ordered positions (or, place-holders); each being optionally occupied by an element. The combination listing can be displayed by an algorithm as a horizontal n-tuple of elements (refer to the column labeled as (6,3)_pull in table 2).

This listing pertains to a n-tuple of symbols on a two dimensional display medium. Whose spatial arrangement is linear (usually: horizontal or vertical).

(Here after, a combination listing is simply referred to as a combination. Note: the standard definition of combination is given in the appendix.)

More specifically, a combination refers to a unique positioning of elements within the ordered sequence of positions. Extra combination(s) are not created by swapping places among elements; the elements are considered to be indistinguishable. A complete combinatorial array, refers to all possible distinct combinations, that can be formed from, a fixed number of place-holders, containing a fixed number of elements.

A complete combinitorial array can further be described by the ordering of the combinations within the array. A label can be assigned to a given permutation (of combinations), of the complete combinitorial array. Refer to table 2. Consider a complete combinitorial array where each combination has 3 elements that can be placed at 6 possible locations. One such permutation of combinations is labeled (6,3)_pull. Another, permutation of these combinations is labeled (6,3)_push. Table 2, also illustrates (7,5)_pull, a complete combinitorial array whose combinations all have 5 elements that can be placed at 7 possible locations.
In general, the two types of prescribed orderings are called: (q,r)_push, and (q,r)_pull.

Note 1: q = number of positions, a positive integer >= 1
        r = number of elements, a positive integer <= q

Properties:

An element can only occupy one position, (at a given time in a combination). Here, elements are considered to be identical or indistinguishable to each other. The number of elements does not change, it can be set to the number of positions, or less by some integer constant, but not fewer than zero element(s). Only, the position of the element(s) can change, (from one combination to the next).

Consider one end of the combination as having the most significant position. When moving to the opposite end, the locations become less significant. (The opposite end has the least significant position.) The algorithm starts with a combination that has all of it's elements contiguously least significantly positioned, (call this right-justified). A combination whose elements are all contiguously located in the most significant positions, is called left-justified. (The algorithm ends with this combination.)

The following algorithm lists a combinitorial array, beginning with a right-justified combination and ending with a left-justified combination.

The current combination refers to the combination being formed or listed during a given stage in the algorithm, from the right-justified combination to the left-justified combination.

Discussed (below), are two algorithms that generate the same type of complete combinitorial array, namely: (q,r)_pull. One algorthm uses a traditional procedural approach. The other a more 'functional' style employs a data driven approach.

## An algorithm for listing (q,r)_pull.
(procedural version)

i) Start with a right-justified array.
Initialize a counter, LIMIT=1

ii a) Append the current combination to the combination array.
Increment the counter LIMIT=LIMIT+1
Check is LIMIT < (q,r)
If so, continue to ii b). Otherwise, stop.

ii b) Refer to the least significant element of the current combination.

iii) If this element has a more significant element immediately preceding it, then refer to this preceding element.

iv) Repeat iii), until you refer to the element that has an empty place-holder immediately preceding it.

iv a) If an empty place-holder is encountered: Move this element to that place-holder. If an element is moved leftward, (to the next significant place-holder); right-justify any elements less significant than the element just moved leftward. (Moving an element leftward and right-justifying any less significant elements counts as one combination. Go to ii a).

iv b) If no empty place-holder is encountered: Check to see if all of the elements are left-justified. (This occurs when none of the elements has vacant position preceding it.) If all of the elements are left-justified, stop. Otherwise, go to ii b).

## An algorithm for listing (q,r)_pull.
(data driven version)

The algorithm in general:

Generate a (q x r) array, called a place value array, (abreviated as PVA). Whose generalised format is as follows:

TABLE 0:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| (q-1,r) | (q-2,r) | (q-3,r) | ... | (r,r) | 0 | -1 | ... | -1 |
| -1 | (q-2,r-1) | (q-3,r-1) | ... | (r-1,r-1) | 0 | -1 | ... | -1 |
| -1 | -1 | (q-3,r-2) | ... | (r-2,r-2) | 0 | -1 | ... | -1 |
| ... | | | | | | | | |
| ... | | | | | | | | |
| ... | | | | | | | | |
| -1 | -1 | -1 | ... | | (3,1) | (2,1) | (1,1) | 0 |

(Array entries marked -1, indicate not useable element positions.)

Create a counter running from 0 to (q,r)-1. Partition a given instance of this this count, by using the largest positive values from each row of the (q x r) array. The columnar indices of these values mark the locations of the elements in the combination. In this way a positive integer ranging from zero to (q,r) can be associated with each combination obtained from q items chosen from a set of r items.

Each element will have some value depending not only on it's position, but also depending on it's significance: That is, (whether it is the leading element (most significant element); some element between other elements; or the element that trails the other elements (least significant element).

In general, the top row of the PVA determines the place values for the most significant element. The bottom row of the PVA, determines the place values for the least significant element. The n'th row, from the top, holds the possible place values for the n'th significant element.

More specifically, (for example), the second most significant element could have the values of: (q-2,r-1), (q-3,r-1), ..., 1, 0; during the instances, when it occupies the places indexed by: q-1, q-2, ..., 2, respectively. (Refer to the second row from the top in the generalized format for the PVA (table 0).)

For values for other elements, refer to the row labeled by that element.

A, -1 in a given row of the PVA, signifies a location that can not be held by the element associated with that row.

An example of the algorithm:

Using, q=6, r=3. (refer to Note 1, and (6,3)_pull of table 2)

A (6 x 3) place value array for (6,3)_pull is derived from the (q x r) generalized format (shown in table 0):

Table 1:

```
      10 4 1 0 . .   <--  (5,3) (4,3) (3,3)   0     -1   -1
       . 6 3 1 0 .   <--    -1  (4,2) (3,2) (2,2)   0    -1
       . . 3 2 1 0   <--    -1    -1  (3,1) (2,1) (1,1)   0

index: 6 5 4 3 2 1   <-- columnar location indices, of the combination
```

Refer to the second row from the top in the place value array shown in table 1. Since, any possible position indices for one element are determined by only one given row of the place value array. The second most significant element could have the values of: 6 3 1 0; at the places indexed by: 5 4 3 2, respectively.

Since, (6,3) = 20 combinations, (form the complete combination array); create a counter running from 0 to (q,r)-1 = (6,3)-1 = 19.

However, this example shows how to list or generate one combination. The other combinations, can be obtained in the similar manner to what is described below:

To generate the combination associated with, say the integer 18, proceed as follows:

Scan the top row of the place value array (table 1), find the largest value that is less than 18, this being 10. The columnar index of 10 is 6. Thus, 6 gives the position of the most significant element in the combination.

Scan for the next lower row of the place value array, find the largest value that is less than (18 - 10), this being 6. The columnar index of 6 is 5. Thus 5 gives the position of the next significant element in the combination.

Scan the bottom row of the place value array, find the largest value that is less than (18 - 10 - 6), this being 2. The columnar index of 2 is 3. Thus, 3 gives the position of the least significant element in the combination.

The combination associated with the integer 18, has the positions of 6, 5, 3; for the indices of the elements. Placing elements at the place-holders dictated by the indices 6, 5, 3; forms the result:

1 1 _ 1 _ _ .

In table 2, the combination in the (q,r)_pull column, that shares the same row with the integer 18; is identical to the result above, just achieved by using the data driven version.

The complete combinitorial array as listed in table 2, under the column labeled (6,3)_pull was generated by the algorithm using the procedural version. The algorithm using the data driven version can be used to generate all of (6,3)_pull, (producing identical output to the algorithm using the procedural version).
More generally, other integers associated with combinations of (q,r)_pull can be generated by using this data driven version.
(By using a (q x r) place value array for the data.)

## NUMERAL SYSTEM USING AN ARRAY OF PERMUTATIONS

Definitions:

Here, a permutation listing is defined to be: a sequence of a constant (yet, finite), number of ordered positions (or, place-holders); each being occupied by a disparate element.
The permutation listing can be displayed by an algorithm as a horizontal n-tuple of elements (refer to the column labeled as 4!_cascade in table 2).

This listing pertains to a n-tuple of symbols on a two dimensional display medium. Whose spatial arrangement is linear (usually: horizontal or vertical).

(Here after, a permutation listing is simply referred to as a permutation. Note: the standard definition of permutation is given in the appendix.)

More specifically, a permutation refers to a unique positioning of elements within the ordered sequence of positions. A complete permutation array refers to all possible distinct permutations, that can be formed from, a given fixed number of number of elements. A label can be assigned to a given permutation (of permutations), that is a complete permutation array. Consider a complete permutation array where each permutation has 4 elements. Such permutation of permutations is labeled as: 4!_cascade. (Refer to table 2.)

In general, the algorithm described here, generates an array of permutations called n!_cascade.

Note 1: n refers to the number of elements each permutation has.

A minres class is a partitioning of the set of whole numbers.

Let W = { 0, 1, 2, 3, ...}.
Let m be an element of M, where m is a multiple of n, such that M = { n, 2n, 3n, ...}.

Let R be the absolute value of the mimimum of the representatives of the residue classes of modulo n, R = { 0, 1, 2, 3, ..., (n-1) }, where r is an element of R.

A given minres class is formed by adding a particular m to each element of R. Such that, a given minres class is:

{ m+0, m+1, m+2, m+3, ..., m+n-1 }

The set of whole numbers is partitioned as follows:

{{0,1,2,...,n-1}, {n,n+1,n+2,...,2n-1}, {2n,2n+1,2n+2,...3n-1},...}

For example, the minres classes of modulo 4:

{{0,1,2,3}, {4,5,6,7}, {8,9,10,11}, ...}

Properties:

An element can only occupy one position, (at a given time in a permutation). The number of elements (length of the permutation), does not change, for each permutation, of a complete permutation array. The same set of disparate elements are used for each permutation. Only, the position of the elements can change, from one permutation to the next.

Consider one end of the permutation as having the most significant position. When moving to the opposite end, the locations become less significant. The opposite end has the least significant position.

A unique rank is associated with each element. Thus, elements can be sorted and listed in a permutation. The lowest ranking element can be placed in in the most significant position. The next lowest element can be placed in the next significant position. This is repeated until the highest element is put in the least significant position. (This is the ordering of the initial permutation that the algorithm generates.)  The last permutation has the ordering of it's elements reversed.

The current permutation refers to the permutation being formed or listed during a given stage in the algorithm

An algorithm for listing an n!_cascade.

In general:

Create a counter running from 0 to (n!-1), that designates the numeral being described.  So that, a positive integer ranging from zero to (n!-1) can be associated with each permutation (of size n), of a complete permutation array.

The current permutation is derived from a given instance of this this count (called, the permutation count).  The current permutation is labeled CPRN, the current permutation count is labeled CPC.

Let USED refer to a used element array (the array containing the elements so far derived in forming the current permutation). Initialize the used element array (initially, as being empty).

Designate REF as the reference list, an (ordered n-tuple), of all of the elements of the permutation, (sorted by rank).  The indices of ref range from 0 to n-1.  Where, the most significant element has an index of 0, the next significant element has an index of 1.  The index of successively lesser significant elements have indices correspondingly incremented by one.  Up to until, the least significant element, whose index is n-1.  REF is a constant.

Assign integer i, to designate the i'th significant element. Where, i=1, refers to the most significant element. Continue to i=n,  this refers to the least significant element.

Let V be the value of the element in the current permutation at position i.

Let Q be the 'minres class'element, that refers to a particular element of the permutation.  Alternatively stated, Q the index pointer, (points to the next element to use), from the sorted list of available elements.  However, using such a sorted list of availble elements requires pruning any used element, and moving any elements (that followed), forward --to fill the place left behind by the used element.

Less work can be done by using the following approach:
Use the reference array REF (a constant), and the used element array USED, and a counter called OFFSET.  As elements are used they are appended to USED.  Here, ( Q + OFFSET ) points to the next element to use, from REF.

The OFFSET is calculated as follows:
Every time a new element is obtained; successively compare the new element with elements in the used element array. Each time an element of a equal or higher rank is encountered, increment an offset counter.  Append the new element to the used element array.  Add the offset count to the quotient of the integer division.  The result is used to index the reference array in

obtaining the next element. (The offset count is just referred to as the OFFSET.)

```
Note 2:   DIV refers to integer division
          MOD refers to the remainder from integer division
```

K is the quotient of DIV and MOD. Whose dividend determines the minres class that forms a bijection with an element of the reference list REF. The remainder of K MOD (n-i)! is used to calculate the next K.

FLG is an array of flags. FLG is the same size as REF. When a flag of FLG is set the corresponding element in REF is marked as used.

```
The algorithm begins as follows:
    REF = [a,b,c, ... , n]    where, n is the size of the permutation
    append REF to CPRN

repeat while CPC <= (n! - 1):
    K = CPC,  USED = [],  OFFSET = 0,  i = 1,  V = '', Q = 0
    FLG = [0,0,0, ... , 0]    the same size as REF

    repeat while i <= (n-1):
        Q =  K DIV (n-i)!
        P =  Q + OFFSET
        V =  REF(P)
        FLG(P) = 1
        calculate next K,
        K =  K MOD (n-(i+1))!
        calculate next offset:
           compare REF(Q) with elements of USED
           the OFFSET is the number of elements
           of USED being <= REF(Q)

        append V to USED
        i = i+1
end of loop, invarients: i=0 to (n-1), USED[] to USED[V0,...,V(n-1)].

when i = n:
Scan flag array FLG for the location of the remaining flag,
that is still set as unused.  (The corresponding position in
REF holds the remaining element.)

repeat for j=0 to j==(n-1):
    if FLG[j] <> 0 then append REF[j] to USED
    j = j+1
end of loop, invarients: j=0 to (n-1), USED[0,...,n-1] to USED[0,...,n].


CPC = CPC - 1
append USED to CPRN
end of loop, invarients: CPC=0 to (n-1), CPRN[0] to CPRN[(n-1)!].
end of algorithm.
```

Example: Refer to the column labeled 4!_cascade of table 2.

```
  Here, n=4                 the number of elements in a permutation
  then, n! = 4! = 24        the number of permutations in 4!_cascade
```

To generate the permution associated with, say the integer 17.
Proceed as follows:

```
Initially:
    K = 17
    USED = []
    OFFSET = 0
    i = 1
    REF  =  [a,b c,d]      The indices of REF range from 0 to n-1 = 3.
    indices: 0,1,2,3       Where, the most significant element has an
                           index of 0, the least significant element
                           has an index of 3.  REF is a constant.
                           (REF[0] = a, REF[1] = b, etc.)


Obtaining most significant element:
  Q = K DIV (n-i)!         obtain 'minres class' element
    = 17 DIV (4-1)!
    = 2
  V = REF( Q + OFFSET )
    = REF( 2 + 0 )
    = REF(2)               index of most significant element
    = 'c'

  append V to USED,        USED = [c]
  FLG(2) = 1               toggle corresponding flag as used
  calculate next K,
  K = K MOD (n-i)!
    = 17 MOD (4-1)!            (remainder, of second minres class)
    = 5


Obtaining next most significant element:
  i = i+1
    = 2
  Q = K DIV (n-i)!
    = 5 DIV (4-2)!
    = 2
calculate next offset,
  REF(Q) = REF(2) = 'c'
  comparing 'c' with elements of USED =[c]
  the OFFSET is the number of elements of USED that are <= 'c'
  OFFSET = 1

  V = REF( Q + OFFSET )
    = REF( 2 + 1 )
    = 'd'

append V to USED,          USED = [c,d]
FLG[3] = REF[3]
calculate next K,
  K =  K MOD (n-i)!
    =  5 MOD 2!            remainder
    =  1


Obtaining next most significant element:
  i = i+1
    = 3
  Q = K DIV (n-i)!
    = 1 DIV (4-3)!
    = 1
calculate next offset,
  REF(Q) = REF(1) = 'b'
  comparing 'b' with elements of USED =[c,d]
  the OFFSET is the number of elements of USED that are <= 'b'
```

```
  OFFSET = 0

  V = REF( Q + OFFSET )
    = REF( 1 + 0 )
    = 'b'

append V to USED,          USED = [c,d,b]
FLG[1] = 1
calculate next K,
  K =  K MOD (4-3)!
    =  1 MOD 1!
    =  1                   remainder
```

Obtaining least significant element ~O(n):

```
  i = i+1
    = 4
```

Since, i=n, only one element remains.

Scan flag array FLG for the location of the remaining flag, that is still set as unused. (The corresponding position in REF holds the remaining element.)

```
repeat for j=0 to j==(n-1):
    if FLG[j] <> 0 then append REF[j] to USED
    j = j+1
end of loop, invarients: j=0 to (n-1), USED[0,...,n-1] to USED[0,...,n].
```

Here, FLG[0] <> 0, then REF[0] being 'a' is appended to USED.

Alternate approach ~O(n^2);

Comparing: USED=[c,d,b] with REF=[a,b,c,d], shows that 'a' is the remaining element. Append 'a' to USED thus, USED = [c,d,b,a].

Listing each element in the order it was obtained from USED gives the permutation:

cdba

This checks with the permution listed in table 2, under the column labeled permutation cascade, in the same row as the integer 17.

In table 2, the permutation in the n!_cascade column, that shares the same row with the integer 17; is identical to the result above, just achieved by using the data driven version.

The complete permutation array as listed in table 2, under the column labeled 4!_cascade was generated by the algorithm using the procedural version. The above algorithm (using the data driven version), can be used to generate all of 4!_cascade, (producing identical output as the algorithm using the procedural version).

Similarly, other integers associated with permutations of n!_cascade can be generated by using this data driven version.

0 1 1 0 1 0

A permutation (can be written as a sequence of integers), for example:

0 3 4 1 2

The permutation compliment would be written by correspondingly writing down the n's compliment of each element. (Where, n is the value of the largest element in the permutation.) The permutation compliment, of the previous example, would be written down by obtaining the 4's compliment of each corresponding element, namely:

4 1 0 3 2

Note: the existence of a bijection between (n,q)_push and (n,q)_pull.

Note: the existence of a bijection between n-bit binary numerals and (n,p)_pull, (or (n,p)_push), where p={0,1,2,...,n}. Moreover, a numeral system can be formed from a composite of combination cascades of a given length.

For example, using combination cascades in "composition", say, (n,p)_pull where p={0,1,2,3,...,5}.

```
binary   (5,0)_push   binary   (5,1)_push   binary   (5,2)_push   binary   (5,3)_push   binary   (5,4)_push
binary   (5,5)_push
00000    - - - - -    00001    - - - - |    00110    - - - | |    10000    - - | | |    11010    - | | | |
11111    | | | | |
                      00010    - - - | -    00111    - - | - |    10001    - | - | |    11011    | - | | |
                      00011    - - | - -    01000    - | - - |    10010    | - - | |    11100    | | - | |
                      00100    - | - - -    01001    | - - - |    10011    - | | - |    11101    | | | - |
                      00101    | - - - -    01010    - - | | -    10100    | - | - |    11110    | | | | -
                                            01011    - | - | -    10101    | | - - |
                                            01100    | - - | -    10110    - | | | -
                                            01101    - | | - -    10111    | - | | -
                                            01110    | - | - -    11000    | | - | -
                                            01111    | | - - -    11001    | | - - -
```

Observe that by substituting the symbols: - |, for 0 1; and then re-ordering the (5,p)_push combinations to monotonically increase according to binary numeral convention, you obtain the full 5 bit consecutive binary integer sequence (with no repeating or missing integers): 00000, 00001, ..., 11111.

From this you can derive: (2**n) -1 = sigma (c(n,p)) where p={0,1,...,n}

Each numeral system has it's own advantages and disadvantages. It is taken for granted that the Hindu-Arabic system is more convenient for many uses from: multiplication to algebraic operations. However, other systems may have advantages for limited applications. Roman numerals have an advantage for problems involving partitioning a number in to it's components, (that add up to the number being partitioned). Especially, for components involving multiples of tens, fives, and ones. For example, partitioning xxviii into multiples of tens, fives, and ones; results in: two tens, one five, and three ones. Combinatorial, numeral systems can be incorporated in writing more compact algorithms that create combinatorial lists. The same applies to the permutational number system.

Note: For permutations of length 2 to 21 a base ten representation is shorter in length than the permutation representing an integer. For permutations of length 22 to 24 a base ten representation is the same length as the permutation representing the integer. However, for permutations whose length is greater than 25, a base ten representation is longer in length.

Also Note: A base ten digit uses up a certain amount of bits depending on the hardware or encoding (ASCII, EBDIC, unicode, etc.). So does the symbol used in the permutation. These must be taken into account when determining overall usage space.

For combinations: say, (900,450), the length of a combination n-tuple is 900, and the base ten representation is 270 digits in length. (This being near the upper limits of machine calculation, using recursive algorithms.) Without recursion (10000,5000), a 10000 element long combination n-tuple, can represent up to a 3009 (base ten) digit number. These combinatorial numeral systems are not as compact in their notation as base ten notation. However, these combinatorial numeral systems approach the compactness of base two notation.

Aside: Another way of looking at a combination n-tuple, is by conceptualizing the way that empty place-holders (symbolized by _) move, as shown by successive combination n-tuples (associated with numbers from zero to 19 in the table above). The _ place-holders start by all being left justified. They then move rightward (in-between the letters. The place-holders continue moving past the letters; until, all of the place-holders are right justified.

## SYMMETRY ASSOCIATED WITHIN ARRAYS:
## (q,r)pull, (q,r)push AND n!_cascade

To observe symmetry in these arrays: list the n-tuples in these arrays in a top to bottom arrangement; list the elements of each n-tuple in a left to right direction.

Consider the order of the n-tuples, and the order of each element in the top half of the array. To obtain the bottom half of the array: Reverse the order of the n-tuples of top half of the array. Then, reverse the order of the elements in each n-tuple.

In some of these arrays; the inverted bottom half of the array 'mirrors' a given type of compliment of the top half. When q=2r, the one's compliment is expressed in the inverted bottom half of (q,r)_pull, and (q,r)_push.  When n MOD 2 = 0, the n's compliment is expressed in the reversed order of the bottom half of n!_cascade.  This property can be used to reduce the number of iterations in algorithms that generate these types of arrays.

```
example: (6,3)_pull    (writing the combination using 0's and 1's )

        top half of      bottom half of
        (6,3)_pull        (6,3)_pull
   0   0 0 0 1 1 1   10   1 0 0 0 1 1   19   1 1 1 0 0 0     0 0 0 1 1 1
   1   0 0 1 0 1 1   11   1 0 0 1 0 1   18   1 1 0 1 0 0     0 0 1 0 1 1
   2   0 0 1 1 0 1   12   1 0 0 1 1 0   17   1 1 0 0 1 0     0 0 1 1 0 1
   3   0 0 1 1 1 0   13   1 0 1 0 0 1   16   1 1 0 0 0 1     0 0 1 1 1 0
   4   0 1 0 0 1 1   14   1 0 1 0 1 0   15   1 0 1 1 0 0     0 1 0 0 1 1
   5   0 1 0 1 0 1   15   1 0 1 1 0 0   14   1 0 1 0 1 0     0 1 0 1 0 1
   6   0 1 0 1 1 0   16   1 1 0 0 0 1   13   1 0 1 0 0 1     0 1 0 1 1 0
   7   0 1 1 0 0 1   17   1 1 0 0 1 0   12   1 0 0 1 1 0     0 1 1 0 0 1
   8   0 1 1 0 1 0   18   1 1 0 1 0 0   11   1 0 0 1 0 1     0 1 1 0 1 0
   9   0 1 1 1 0 0   19   1 1 1 0 0 0   10   1 0 0 0 1 1     0 1 1 1 0 0
                     Compare top half   Flip or reverse    Reverse the order
                     with the bottom    the bottom half's  of each element
                     half.              order.             in each n-tuple,
                                                           (of the reversed
                                                           bottom half).
                                                           Or, since  q=2r;
                                                           1's compliment
                                                           these elements.
                                                           The result is
                                                           again the top half.
```

```
example: (6,3)_push       (using 0's as the placeholders)

        top half of      bottom half of
        (6,3)_push        (6,3)_push
   0    0 0 0 1 1 1  10   0 0 1 1 1 0   19   1 1 1 0 0 0    0 0 0 1 1 1
   1    0 0 1 0 1 1  11   0 1 0 1 1 0   18   1 1 0 1 0 0    0 0 1 0 1 1
   2    0 1 0 0 1 1  12   1 0 0 1 1 0   17   1 0 1 1 0 0    0 1 0 0 1 1
   3    1 0 0 0 1 1  13   0 1 1 0 1 0   16   0 1 1 1 0 0    1 0 0 0 1 1
   4    0 0 1 1 0 1  14   1 0 1 0 1 0   15   1 1 0 0 1 0    0 0 1 1 0 1
   5    0 1 0 1 0 1  15   1 1 0 0 1 0   14   1 0 1 0 1 0    0 1 0 1 0 1
   6    1 0 0 1 0 1  16   0 1 1 1 0 0   13   0 1 1 0 1 0    1 0 0 1 0 1
   7    0 1 1 0 0 1  17   1 0 1 1 0 0   12   1 0 0 1 1 0    0 1 1 0 0 1
   8    1 0 1 0 0 1  18   1 1 0 1 0 0   11   0 1 0 1 1 0    1 0 1 0 0 1
   9    1 1 0 0 0 1  19   1 1 1 0 0 0   10   0 0 1 1 1 0    1 1 0 0 0 1
                       Compare top half   Flip or reverse     Reverse the order
                          with the bottom    the bottom half's   of each element
                          half.              order.              in each n-tuple,
                                                                 (of the reversed
                                                                 bottom half, only
                                                              works for some values).
                                                                 Or, since  q=2r;
                                                                 1's compliment
                                                                 these elements.
                                                                 The result is
                                                                 again the top half.
```

example: 4!_cascade (using 0 1 2 3 as the elements of the permutation)

| top half of 4!_cascade | | bottom half of 4!_cascade | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 1 2 3 | 12 | 2 0 1 3 | 23 | 3 2 1 0 | 0 1 2 3 |
| 1 | 0 1 3 2 | 13 | 2 0 3 1 | 22 | 3 2 0 1 | 0 1 3 2 |
| 2 | 0 2 1 3 | 14 | 2 1 0 3 | 21 | 3 1 2 0 | 0 2 1 3 |
| 3 | 0 2 3 1 | 15 | 2 1 3 0 | 20 | 3 1 0 2 | 0 2 3 1 |
| 4 | 0 3 1 2 | 16 | 2 3 0 1 | 19 | 3 0 2 1 | 0 3 1 2 |
| 5 | 0 3 2 1 | 17 | 2 3 1 0 | 18 | 3 0 1 2 | 0 3 2 1 |
| 6 | 1 0 2 3 | 18 | 3 0 1 2 | 17 | 2 3 1 0 | 1 0 2 3 |
| 7 | 1 0 3 2 | 19 | 3 0 2 1 | 16 | 2 3 0 1 | 1 0 3 2 |
| 8 | 1 2 0 3 | 20 | 3 1 0 2 | 15 | 2 1 3 0 | 1 2 0 3 |
| 9 | 1 2 3 0 | 21 | 3 1 2 0 | 14 | 2 1 0 3 | 1 2 3 0 |
| 10 | 1 3 0 2 | 22 | 3 2 0 1 | 13 | 2 0 3 1 | 1 3 0 2 |
| 11 | 1 3 2 0 | 23 | 3 2 1 0 | 12 | 2 0 1 3 | 1 3 2 0 |
| | | Compare top half to bottom half. | | Flip or reverse the bottom half's order. | | Reverse the order of each element in each n-tuple, (of the reversed bottom half). Or, since 4 MOD 2 =0; 4's compliment these elements. The result is again the top half. |

## RECURSIVE SELF-SYMMETRY
### (in n!_cascade)

The elements after the most significant element of an n!_cascade, constitute an array of (n-1)!_cascades. There are n of these (n-1)!_cascades. Recursively, the elements after the most significant element of an (n-1)!_cascade, constitute an array of (n-1)!_cascades. There are n(n-1) of these (n-1)!_cascades. Recursion, can be repeated until 2!_cascades are reached.

example:

```
4!_cascade                 3!_cascades                  2!_cascades
   abcd       a bcd            bcd            b cd          cd
   abdc       a bdc            bdc            b dc          dc
   acbd       a cbd            cbd
   acdb       a cdb            cdb            c bd          bd
   adbc       a dbc            dbc            c db          db
   adcb       a dcb            dcb
   bacd                                       d bc          bc
   badc       b acd            acd            d cb          cb
   bcad       b adc            adc
   bcda       b cad            cad            a cd          cd
   bdac       b cda            cda            a dc          dc
   bdca       b dac            dac
   cabd       b dca            dca            c ad          ad
   cadb                                       c da          da
   cbad       c abd            abd
   cbda       c adb            adb            d ac          ac
   cdab       c bad            bad            d ca          ca
   cdba       c bda            bda
   dabc       c dab            dab            a bd          bd
   dacb       c dba            dba            a db          db
   dbac
   dbca       d abc            abc            b ad          ad
   dcab       d acb            acb            b da          da
   dcba       d bac            bac
              d bca            bca            d ab          ab
              d cab            cab            d ba          ba
              d cba            cba
                    partitioning              a bc          bc
                    off 3!_cascades           a cb          cb
                    from 4!_cascade
                                              b ac          ac
                                              b ca          ca

                                              c ab          ab
                                              c ba          ba
                                                  partitioning
                                                  off 2!_cascades
                                                  from 3!_cascades
```