A Generating Function that Counts the Combinatorial Full-Span Sub Array Structure of a Regular Array with Some Applications to APL

An Algorithm to Compute All Full-Span Sub Arrays of a Regular Array

A New "Dual-View" Diagram of Array Structure

Ronald I. Frank
Ronald I. Frank

Information Technology Researcher and Innovator, Senior R & D Project Manager, International Consultant, R & D Business Chairman & CEO, and Life-long Educator.

IBM
I. Research Division staff member, manager & corporate-board consultant
II. Principal IBM US spokesman/consultant for the engineering and scientific PC
III. Scientific Center staff member / project manager and staff member & Computer Center Manager
IV. District and Branch Offices Scientific Marketing Rep.
V. Service Bureau Development Programmer: commercial & scientific applications

IBM Projects
I. Corporate-board consultant [educational uses of the internet]
II. Research manager of parallel computer interfaces
III. Research manager of low-end APL
IV. Research staff member (APL language and Virtual Reality research
V. Team created the first Remote Operation of [370/3090 Main Frame] Computers
  - VM OS Programmable Operator component.
VI. Team created first modern LAN Network Management.
VII. Team developed the first WAN line quality monitoring technology.
VIII. APL Development team - Earliest "Home Use" Of Personal Computers
  - Developed Scientific and Engineering applications for "proto personal computers"
IX. Manager IBM Systems Research Institute Computing Center [Grad School & Research Center]
X. Scientific Marketing Consultant to ~10% of US Industrial Research. (Esso (sic), ATT, Bell Labs, General Dynamics Electric Boat, etc.)
  - Special Scientific rep to the University of Pennsylvania / Consultant to Provost on developing a computing facility for Academic/Scientific use
  - Created some of the earliest courses and taught IBM-ers and customers what computers were and how they could be used.

Entrepreneurial Business Experience
IMMERSIVE SYSTEMS INC. (A virtual reality R & D company) Chairman and CEO

EDUCATION

Pace University School of Computer Science & Information Systems DPS in Computing.

New York University Courant Institute of Mathematical Sciences Ph.D. Student In Applied Math as an IBM GRADUATE RESIDENT FELLOW. MS Applied Math & Ph.D. “ABD”.

Rutgers University BA, Math

ACADEMIC EXPERIENCE (Undergraduate and Graduate Level)
Massachusetts College System & Pace University All courses in CS & IS & Telecom.

HOBBIES
I like to hike the Westchester North County Trailway, snorkel, and read science fiction - not all at the same time.
Table of Contents

These papers were presented by Ronald I. Frank at the
International Conference on APL sponsored by the
ACM Special Interest Group on the APL Programming Language
held during the
ACM 2003 Federated Computing Research Conference
in San Diego, California from June 7 - June 14, 2003.

A Generating Function that Counts the Combinatorial ............. page 1
Full-Span Sub Array Structure of a Regular Array
with Some Applications to APL

Originally published on pages 63-69 in the
Proceedings of the 2003 Conference on APL: Stretching the Mind

An Algorithm to Compute All Full-Span Sub Arrays ............... page 9
of a Regular Array

Originally published on pages 48-58 in the
Proceedings of the 2003 Conference on APL: Stretching the Mind

A New "Dual-View" Diagram of Array Structure ..................... page 21

Originally published on pages 59-62 in the
Proceedings of the 2003 Conference on APL: Stretching the Mind

Appendix 1: class AlgorithmDriver (Java code) .................. page 25

Appendix 2: class Primitives (Java code) ......................... page 31
A Generating Function That Counts The Combinatorial Full-Span Sub Array Structure of a Regular Array with Some Applications to APL

Ronald I. Frank, DPS (Computing)
Associate Professor School of CS & IS
Pace University
325 Goldstein Hall
861 Bedford Road
Pleasantville, NY 10570
1-(914) 773-3444
Email rfrank @ pace.edu

ABSTRACT

Using a radically new way of representing arrays, we present a formalism that expands (or decomposes) a regular array into a weighted sum of null arrays. We show that this "polynomial" expansion (1.16) exhaustively represents the regular full-span array sub structure of the original array. Full-span means full length in the dimensions used. The polynomial is a generating function whose coefficients of which count and indicate the shape of the regular full-span sub arrays of the given regular array. These results are all structural. They do not use knowledge of the particular data contents of the arrays. We apply this new decomposition to catenation and lamination and uncover some new insights into array structure.

The decomposition and the algebraic results provide a unifying view and new formalism for regular multi-dimensional arrays. It has application wherever multi-dimensional arrays are used, particularly to generalized hyper cube architectures, OLAP hierarchiaclal data structures, and array oriented languages. It subsumes some previous results. Some of these applications are indicated with their bibliography.

There is a combinatorial argument on the shape vector that could generate the coefficients, but it does not give the structural insight of this approach.

Categories and Subject Descriptors

General Terms
Algorithms, Languages, Theory.

Keywords
Array Structure, Null-Array Expansion, Sub Array, APL

1. INTRODUCTION

After reviewing standard terminology, we introduce a novel representation of null vectors and use it to derive a polynomial representing the structure of a regular array A. We show that the polynomial is a generating function for the numbers of full span sub arrays of the array A. The algorithmic generation of the sub arrays appears in another paper submitted to this conference [14].

The polynomial has independent interest in that it points to generalizations of the concept of regular array.

2. TERMINOLOGY

We take as known or intuitive the terminology:

- Regular Array (all dimensions filled to full span)
- Equilateral Regular Array (ERA) (all axes of equal length) also called a cube
- Non-equilateral Regular Array (NERA) (axes of possibly different length) also called a block or brick
- Shape Vector (list of dimension lengths of array) 0 in a position means 0 length axis
- Dimension or Rank (number of positions in the shape vector)
- Axis (position in the shape vector)
- Length of Axis (value of position in the shape vector)
- Null Array (at least one 0 in the shape vector)
- Index (vector of values indicating a cell in the array) for our purposes we use index origin 1
- Index Order: we use mathematical or FORTRAN order (left-to-right) for exposition. These results hold, with the proper transpositions, for APL-C-Java order (right-to-left).
- Index (a single position (axis) in the index defined above)
- Index Value (of the vector or of a single position (axis))
- Full Set of Index Values == indicates a cell
• Scalar (has no dimension, no length; also considered the element "in" a cell)
• Domain of the index mapping (the set of cells)
• Co-domain of the index mapping (the set of scalar "contents" of the cells)
  • Even though they may have intrinsic structure, that structure is not used here. This discussion is not concerned with these cell contents.
• The axes are independent of each other.
• Adjacency of cells is implied by the index ordering’s adjacency
• Empty Arrays: Null arrays are to be distinguished here from empty arrays. Empty arrays can be empty because cells map to a null element in the codomain (they therefore have no "contents"). Also, empty arrays can be empty because no cell mapping has been specified (they are not mapped). Both of these cases are not our null arrays, which have no cells.

3. NUMBER REPRESENTATION ↔ ARRAY NOTATION

3.1 Critical Observation on Notation
We observe that the mathematical shape indicator
$$\{ n_1 \times n_2 \times \cdots \times n_{N-1} \times n_N \}$$
which is an algebraic numerical product having a numerical value can be thought of as "representing" an N dimensional array containing that many index sets or cells. The factors are the count of cells in each dimension. If all of the factors are equal we use the usual notation
$$\begin{array}{c}
 (n) \end{array}^N \]

In this number representation, an $$\begin{array}{c}
 n = 0 \end{array}$$ factor represents a dimension (axis) that has no length. An exponent of $$\begin{array}{c}
 N = 0 \end{array}$$ represents an array with no dimensions (no rank, no axes).

3.2 Two Special Values
The values $$\begin{array}{c}
 (0)^0 = 1 \end{array}$$ and $$\begin{array}{c}
 (1)^0 = 0 \end{array}$$ are two important basic values. The first is the "Iverson Convention" [10], also analyzed in [9] and [5] (see the immediately next section)

3.3 Array Representation – Notation
We introduce a notation for an array that recognizes the corresponding number representation: The exponent is the array dimension, and the subscripts are the lengths in each dimension.
$$\begin{array}{c}
 T_n^N \rightarrow (n_1 \times n_2 \times \cdots \times n_{N-1} \times n_N) \end{array}^N \]
If all of the factors are equal we shorten the notation to:
$$\begin{array}{c}
 T_n^N \rightarrow (n) ^N \end{array}^N \]
The two special cases above become:
$$\begin{array}{c}
 T_0^N \rightarrow (0) ^N = 1 \end{array}$$
$$\begin{array}{c}
 T_0^1 \rightarrow (0) ^1 = 0 \end{array}$$

More generally,
$$\begin{array}{c}
 T_n^0 \rightarrow (n) ^0 = 1 \end{array}$$
for $$n \geq 0$$, and
$$\begin{array}{c}
 T_0^n \rightarrow (0) ^n = 0 \end{array}$$
for $$n \geq 1$$.

3.4 Binomial Expansions (These new representations are of independent and fundamental interest).
We notice the numerical representation implies the two following relationships:
$$\begin{array}{c}
 T_n^N \rightarrow n^N = (n-1)^N + 1 \end{array}$$
$$\begin{array}{c}
 \sum_{i=0}^{n} \binom{n}{i} (n-1)^{N-i} = \sum_{i=0}^{n} \binom{n}{i} T_{n-i}^N \end{array}$$
and
$$\begin{array}{c}
 T_n^N \rightarrow n^N = (n+1)^N - 1 \end{array}$$
$$\begin{array}{c}
 \sum_{i=0}^{n} \binom{n}{i} (n+1)^{N-i} (-1)^i = \sum_{i=0}^{n} \binom{n}{i} (-1)^i T_{n-i}^N \end{array}$$

These are expansions of the cube in terms of arrays of larger and smaller lengths and of equal and smaller dimension.

3.5 Application of the Binomial Expansions:
Derivation of the Master Equation (1.16)
We apply (1.3) to the first of the special cases.
$$\begin{array}{c}
 T_0^n \rightarrow n^0 = 1 = (n-1)^0 + 1 \end{array}$$
$$\begin{array}{c}
 \sum_{i=0}^{n} \binom{n}{i} (n-1)^{0-i} = \binom{n}{0} - n^0 = 1 = T_0^n \end{array}$$
yielding,
$$\begin{array}{c}
 T_0^n \rightarrow T_0^n \forall n \geq 0 \end{array}$$
This is consistent with the purely numerical statement in the discussion of (1.2). Also:
$$\begin{array}{c}
 T_{-n}^0 \rightarrow (-n)^0 = (1-n)^0 = \sum_{i=0}^{n} \binom{n}{i} (1-n)^{0-i} = \binom{n}{0} T_{-n}^0 \end{array}$$
yielding, recursively,
$$\begin{array}{c}
 T_{-n}^0 \rightarrow T_{-n}^0 \forall n \geq 0 \end{array}$$
The further meaning of this negative length is left for another paper. We apply (1.3) to the other special case.
3.6 Vector Notation  
(the prototypical polynomial expansion as a weighted sum of null arrays)

We use the notation for a simple 1 D array, a vector of length \( n \):

\[
T_n^1 \rightarrow (n)T_0^0 + T_0^1
\]

This says nothing about the contents of the vector. It does say that the vector has cell count \( n \) and a shape defined by the 1 D null vector (i.e., it is a 1 Dimensional object).

3.7 Outer Product Notation

We can define higher dimensional regular array shapes by taking outer products of vectors (polynomial multiplication). E.g.,

\[
T_{n,m}^2 = T_n^1 \otimes T_m^1
\]

\[
(nT_0^0 + T_0^1) \otimes (mT_0^0 + T_0^1) =
(nmT_0^0 + (n+m)T_0^1 + T_0^2)
\]

This depends upon the following three results:

\[
(T_0^0) \otimes (T_0^0) = T_0^0 \rightarrow 1 \times 1 = 1
\]

\[
(T_0^1) \otimes (T_0^1) = T_0^1 \rightarrow 1 \times 0 = 0
\]

\[
(T_0^1) \otimes (T_0^1) = T_0^2 \rightarrow 0 \times 0 = 0
\]

The shape holder, which is the null array of a given dimension, acts something like 0 acts in a positional notation system.

3.8 Outer Product Structural Definition of the General Non-Equilateral Array

\[
T_{n_1,n_2,\ldots,n_{N-1},n_N}^N \rightarrow
(n_1T_0^0 + T_0^1) \otimes (n_2T_0^0 + T_0^1) \otimes \ldots
\]

\[
\otimes (n_{N-1}T_0^0 + T_0^1) \otimes (n_NT_0^0 + T_0^1)
\]

To simplify this presentation, we will use index origin 1 and take the minimum axis values to be 1. In general, a non-equilateral regular array \((n_1 \times n_2 \times \ldots \times n_{N-1} \times n_N)\) can be represented by an outer product. This should be evident from geometric considerations.

3.9 Symmetric Functions, Roots of Equations, and the Master Equation

We observe the formal equivalence of this outer product to the polynomial factorization into linear factors

\[
P(X) = (n_1 + X) \times (n_2 + X) \times \ldots
\]

\[
\times (n_{N-1} + X) \times (n_N + X)
\]

This polynomial has coefficients which are the symmetric functions of its roots by the Vieta root theorem [6] and [13]. Using a generalization of (1.12), for \( k \geq 1 \):

\[
(T_0^0) \otimes (T_0^0) = T_0^0 \rightarrow 1 \times 1 = 1
\]

\[
(T_0^1) \otimes (T_0^k) = T_0^k \rightarrow 1 \times 0 = 0
\]

\[
(T_0^1) \otimes (T_0^1) = T_0^2 \rightarrow 0 \times 0 = 0
\]

This allows us to state the **MASTER EQUATION** for NERAs as:

3.10 The Master Equation

\[
(n_1 \times n_2 \times \ldots \times n_{N-1} \times n_N) \rightarrow
\]

\[
T^N_{(n_1,n_2,\ldots,n_{N-1},n_N)} =
\]

\[
\sum_{j=0}^{N} T_{n_j}^j \left[ \sum_{(j)} \left[ (n_{i_1}) \times (n_{i_2}) \times \ldots \times (n_{i_{N-j}}) \times (n_{i_{N-j+1}}) \times \ldots \times (n_{i_N}) \right] \right]
\]

This is a purely algebraic-combinatorial result. We observe that the first two lines can be thought of as representing an array. The equivalent algebraic expansion can be interpreted similarly as having either numerical coefficients or array components. We use the numerical coefficient interpretation in this discussion. By construction the expansion is invariant under arbitrary permutations of the axes.

3.11 Interpretation of this Notation

This states that the general non-equilateral array can be expanded into the weighted sum of null arrays of smaller or equal dimension as in 1.3 to 1.10 above. The coefficients are symmetric functions of the roots of the polynomial (axis lengths). They are products of the roots taken \([N - j]\) at a time from the \( N \) factors making up the array shape vector. There are
The null array factor \( T_0^j \) stands for a \( j \) dimensional array all of whose axes are of 0 length. Interestingly, the one non-zero term (in the numerical interpretation), \( T_0^0 \), has a coefficient that counts the number of 0-D sub-arrays. These are the cells of the original array.

### 3.12 Special Case of the Equilateral Array

When all of the \( N \) dimensional lengths are equal the master equation \( (1.16) \) becomes

\[
T_n^N = \sum_{j=0}^{N} T_0^j \binom{N}{j} n^{N-j}.
\]

### 3.13 The Meaning of the Master Equation

#### Summation By Examples

Since these are purely structural equations, the sum might be taken as a set union. It will appear later that there is a set difference interpretation for coefficients with minus signs.

#### 3.13.1 Example: the 2x2

\[
T_2^2 = (2 \times 2) T_0^0 + (2 + 2) T_0^1 + T_0^2 = 4 T_0^0 + 4 T_0^1 + T_0^2
\]

This says that a \((2 \times 2)\) array comprises all at the same time 4 cells, 4 vectors (the horizontal and vertical rows and columns) and one 2-dimensional object.

#### 3.13.2 Example: the binary hypercube \( (n = 2) \)

\[
T_2^N = \sum_{j=0}^{N} T_0^j \binom{N}{j} 2^{N-j}
\]

This says that the binary hypercube comprises objects of lesser dimension that are themselves binary hyper cubes. This corresponds to the formulation used in [3] for finding sub-cubes of processors for allocation in a hypercube machine.

#### 3.13.3 Example: the vector of length \( n \) (which we started with)

derived from Error! Reference source not found.

\[
T_n^1 = \sum_{j=0}^{1} T_0^j \binom{1}{j} n^{1-j} = \binom{1}{0} T_0^0 n + \binom{1}{1} T_0^1 = n T_0^0 + T_0^1
\]

The interpretation of this in terms of the Master Equation is that a vector comprises its length, here \( n \) cells, and a dimension indicator, here one dimension.

#### 3.13.4 Example: The 2x3x4 regular array (left-to-right)

Figure 1. A \((2 \times 3 \times 4)\) Array

\[
T^3_{(2,3,4)} = T_0^0 (24) + T_0^1 (26) + T_0^2 (9) + T_0^3 (1)
\]

This comes from

\[
T^3_{(2,3,4)} = \sum_{j=0}^{3} T_0^j \left( \sum_{i=0}^{3} [(n_i) x (n_j) x \ldots x (n_{j,0}) x (n_{j,0})] \right)
\]

(1 product of 3 factors)

\[
T_0^0 ([(2) x (3) x (4)]) +
\]

(sum of 3 products of 2 factors)

\[
T_0^1 ([(2) x (3)] + [(2) x (4)] + [(3) x (4)]) +
\]

(sum of 3 "products" of 1 factor)

\[
T_0^2 ([(2)] + [(3)] + [(4)]) +
\]
\[ T_0^3 \left( \sum_{i=3}^{3} [(n_i) \times (n_i) \times \ldots \times (n_{i+1}) \times (n_{i+2})] \right) \]

(sum of 1 product of 3 factors)

\[ T_0^3 (1) \]

3.13.4.1 The Final Result Is

\[
\begin{align*}
T_0^0 & ([(2) \times (3) \times (4)]) + \\
T_0^1 & ([(2) \times (3)] + [(2) \times (4)] + [(3) \times (4)]) + \\
T_0^2 & ( (2) + (3) + (4) ) + \\
T_0^3 & (1) =
\end{align*}
\]

which yields

\[
\begin{align*}
T_0^0 & (24) + \\
T_0^1 & (6) + (8) + (12) = 26 + \\
T_0^2 & (9) + \\
T_0^3 & (1)
\end{align*}
\]

or

\[
T_0^0 (24) + T_0^1 (26) + T_0^2 (9) + T_0^3 (1) \ QED.
\]

This means that the (2x3x4) contains 24 cells, 26 vectors, 9 planes and 1, 3-dimensional entity. The vectors are the 6 front to back vectors, 12 top to bottom vectors, and 8 left to right vectors. The 9 planes are the 2 horizontal planes, 3 vertical front to back planes, and 4 left to right vertical planes.

4. GENERAL SHAPE CARRIERS:

The single 0 dimension case.

\[ T^N (n_1, n_2, n_3, \ldots, n_{N-1}, n_N) = \sum_{i=0}^{N} T_0^i \left( \sum_{j=0}^{N} \cdots \left( [n_i] \times [n_j] \times \cdots \right) \times \left( n_{N-i} \right) \times \left( n_{N-j} \right) \right) \]

The modified interior summation (*) means that because one polynomial factor has a 0 root, the polynomial coefficient sums are missing terms which have that 0 factor. Consider

4.1 (2x3x4) Example

\[
\begin{align*}
T_0^3 & = T_0^0 (2x3x4) + T_0^1 (2x3 + 2x4 + 3x4) + \\
& + T_0^2 (2 + 3 + 4) + T_0^3 (1) \\
T_0^3 & = T_0^0 (24) + T_0^1 (26) + T_0^2 (9) + T_0^3 (1)
\end{align*}
\]

where as

\[
\begin{align*}
T_0^3 & = T_0^0 (2x0x4) + T_0^1 (2x0 + 2x4 + 0x4) + \\
& + T_0^2 (2 + 0 + 4) + T_0^3 (1) \\
T_0^3 & = T_0^0 (0) + T_0^1 (0) + T_0^2 (6) + T_0^3 (1)
\end{align*}
\]

Notice that this null array has NO cells (its first term). Another way of looking at this expansion is that the coefficients are the same as for an array of dimension one less but they appear in the higher dimensional expansion with no cells. To see this consider first:

\[
\begin{align*}
T_2^2 & = T_0^0 (2x4) + T_0^1 (2 + 4) + T_0^2 + \\
& + T_0^3 (6) + T_0^2 \cdot T_0^3
\end{align*}
\]

Now consider the outer product which yields our result:

\[
\begin{align*}
T_0^3 & \otimes T_2^2 = T_0^0 (2x4) \otimes T_0^1 (2 + 4) \otimes T_0^2 + \\
& + T_0^3 (6) \otimes T_0^2 \cdot T_0^3
\end{align*}
\]

5. CATENATION OF ARRAYS [2, 7, 8]

We can combine two conformal arrays by catenation. Conformal means they have the same rank and size in all axes except the catenation axis. However, we discover a very interesting new fact. Let us catenate a (2x3x4) along the third axis with a (2x3x4) to get a (2x3x8)

5.1 (2x3x8) Expansion

\[
T_2^3 (2,3,8) = \sum_{i=0}^{8} T_0^i \left( \sum_{j=0}^{3} [(2) \times (3) \times (8)]_{3-i} \right) = \\
T_0^0 [2] (3) (8) + \\
T_0^0 [(2)(3)+(2)(8)+(3)(8)] + \\
T_0^2 [(2) + (3) + (8)] + T_0^3 = \\
48T_0^0 + 46T_0^1 + 13T_0^2 + T_0^3
\]

5.2 (2x3x8) Expansion

However, two (2x3x4) have the expansion:

\[
\begin{align*}
2T_0^3 & = 2T_0^0 (24) + 2T_0^1 (26) + 2T_0^2 (9) + 2T_0^3 (1) \\
& = T_0^0 (48) + T_0^0 (52) + T_0^0 (8) + T_0^3 (1)
\end{align*}
\]

Subtracting we get.
27^{3.4} - T_{3.3}^{3} = [48T_{0}^{0} + 67T_{0}^{1} + 13T_{0}^{2} + T_{0}^{3}] 
= [0T_{0}^{0} + 67T_{0}^{1} + 5T_{0}^{2} + T_{0}^{3}] = T_{3.3}^{3}

or

\[ T_{3.3}^{3} = 27^{3.4} - T_{3.3}^{3} \]

That is, the expansion of the catenation of two arrays equals the sum of the expansions minus the shape of the catenation interface.

The general proof of this appears in [5]. This also shows a use for set difference as the interpretation of negative coefficients. We subtract the sub arrays by taking them away from the set. This result has many implications and generalizations for the construction of higher dimensional arrays from lower dimensional arrays [5].

6. DE-CATENATION OF ARRAYS

From the arithmetic of the catenation expansions mentioned above, it is clear that taking an array apart (de-catenating) generates an interface shape holder. There is a conservation of these shape holders. The de-aggregating of hierarchical data in OLAP [12] raises the question of the shape of the interface of the two resulting arrays. It is the product of the other dimensions. The combinatorics of the sub arrays of the parts are captured in their expansions.

7. LAMINATION OF ARRAYS

Lamination is an operation which is introduced in APL [2, 7, 8]. It too catenates two conformal arrays but along a new non-existing axis creating an array of length 2 in that new dimension. We use "[1.5]" to mean laminate along a new axis lying between the first and second, creating a third axis.

\[ T_{3.4}^{3} = T_{3.4}^{3}[1.5]T_{3.4}^{2} \]

In terms of our polynomial master equation (1.16) this is simply an outer product with a two vector (using invariance under axis permutation):

\[ T_{3.4}^{3} = (24T_{0}^{0} + 26T_{e}^{0} + 9T_{e}^{1} + T_{e}^{2}) \]

\[ T_{3.4}^{3}[1.5]T_{3.4}^{2} = (24T_{0}^{0} + 26T_{e}^{0} + 9T_{e}^{1} + T_{e}^{2}) \]

\[ (2T_{0}^{0} + T_{e}^{1}) \otimes (12T_{0}^{0} + 7T_{e}^{0} + T_{e}^{2}) =
\]

\[ (24T_{0}^{0} + 14T_{e}^{0} + 27T_{e}^{0} + T_{e}^{2}) = (12T_{0}^{0} + 7T_{e}^{0} + T_{e}^{2}) \]

\[ (24T_{0}^{0} + 26T_{e}^{0} + 9T_{e}^{1} + T_{e}^{2}) \]

QED.

8. GENERALIZED HYPERCUBE MACHINES [1, 3, 4, 11]

The number of sub arrays of a given dimension is determined by the master equation (1.16). If specific indexes are required, they can be computed from the coefficients used in the weighting, by invoking an assumed index order of the coefficient factors.

The different generalizations of hypercubes are regular equilateral arrays with cells being processors and various proposed connectivities. The sub array allocation problem is amenable to analysis using the master equation. The single processor is considered a 0 D array. Two processors or more in a sequential connection are an n element vector, and so on.

9. ACKNOWLEDGMENTS

My thanks to Adin Falkoff for many useful discussions and the loan of an APL system.

10. REFERENCES


[7]. IBM APL LANGUAGE. GC26-3847-5 IBM, San Jose, CA, 1983.


An Algorithm to Compute All Full-Span Sub Arrays of a Regular Array

Ronald I. Frank, DPS (Computing)
Associate Professor School of CS & IS
Pace University
325 Goldstein Hall
861 Bedford Road
Pleasantville, NY 10570
1-(914) 773-3444
Email rfrank@pace.edu

ABSTRACT

We present an algorithm that explicitly generates all full-span regular sub arrays of a given N dimensional regular array. By explicit, we mean that we generate the N-dimensional indexes of the j-dimensional sub array cells that make up each j-dimensional sub array. The algorithm can be viewed as generating all j-dimensional full span sub arrays for fixed j, for j = 0 to j = N. We first list some lemmas and definitions, and then give the algorithm that is based upon the knowledge gained from a new expansion [1] and [2]. There is an alternate combinatorial argument yielding the same algorithm.

First we give an heuristic summary and then a more detailed example use. In the algorithm, we use APL-C-Java order in the indexes, i.e., right-to-left (RTL). However, essentially the same algorithm (except for transpositions) holds for left-to-right (LTR) order as used in FORTRAN. In the exposition, we use mathematical notation, which is LTR.

There is a Java 1.4 code and an APL WS available. The Java code requires an interactive input of the dimension of the original array and its shape vector. The APL WS contains FNS that work the same way, and others that can be used as callable sub FNS. An appendix exhibits the APL code that computes this algorithm for a given j.

Categories and Subject Descriptors

General Terms
Algorithms, Theory.

Keywords
Algorithm, Array Structure, Full Span Sub Array, APL, Regular Array, Array Decomposition.

1. INTRODUCTION

To the Heuristic Summary of the Algorithm

This problem is two-parts;

• To generate a combination of the elements of an N dimensional array's shape vector. S is the N-vector shape vector of array A.

• To generate the sub array N-index sets defined by that combination. The index sets are sets of N-index cell designators naming cells of a sub array of A.

2. DEFAULT BACKGROUND

We assume an index ordering to the shape vector. Either it is left to right (LTR) ordered (FORTRAN ordering) or it is right to left (RTL) ordered (APL-C-Java ordering).

• LTR is interpreted geometrically as row, column, plane, hyper plane etc. and leads to COLUMN MAJOR ORDER.

• RTL is interpreted geometrically as higher dimensional objects, hyper planes, planes, columns, and rows. This leads to ROW MAJOR ORDER.

Whatever choice we make as a default, we use it to interpret all vectors including the shape vector its sub vectors.

The number of numBits = j vectors in BM in II below is \[ \binom{N}{j} \]. The number of j dimensional sub arrays generated for a given j depends on the values of the specific j-element sub set (s) of the shape vector S.

P is the product of the elements of S. p is the product of the elements of s. The number of sub arrays generated for this.
3. GENERATING COMBINATIONS
Two Tricks:

3.1 Binary Representation if Integers
The first trick is to notice that the binary representation of the integers, 0 to \(2^N-1\) are \(N\) bit strings that cover all combinations of \(N\) bits. Therefore, to generate a combination of the shape vector \(S\), we first get a bit combination and then select, by masking or compression, the elements of \(S\) corresponding to the 1 bits.

3.2 Odometering
The second trick is to use only one counting routine (odometering routine) to advance or generate the rows of \(V\) regardless of the default ordering we have chosen.

4. The Algorithm

4.1 Input \(N\) and \(S\), Get \(P\), Choose an Ordering
\((N\) is the rank or dimension of \(A\), and number of elements of \(S\), \(A\)’s shape vector\).
- \(P = \) Get product of the elements of \(S\).
- Pick a default ordering for the elements of the shape vector \(S\) (RTL or LTR).
- If LTR, reverse it (we’ve implemented only the RTL version).

4.2 Create BMN (\(2^N\times N + 1\) column matrix)
BMN is BM with a first column of numBits.
- Create BM whose \(2^N\) rows are the \(N\)-bit binary form of the integers \(0\) to \(2^N-1\).
- Append a new first column containing the row’s numBits to make BMN. numBits is the number of 1 bits in the row.
- Sort BMN by the number of 1 bits (numBits) in the rows.

4.3 To generate a set of \(j\)-dimensional sub arrays of \(A\) [using 0 origin indexing]:

4.3.1 Drop the first and last rows of BMN, set \(j = 1\) at start.
- \(j = 0\) (b all 0s. See C below.) yields (odometered \(S\)) = \(V = W\) is \(P\), 0-D arrays; \(A\)’s cells. If LTR reverse \(W\) here.
- \(j = 2^N\) (b all 1s. See C below.) yields (odometered \(S\)) = \(V = W\) as \(1\), N-D array \(A\). If LTR, reverse \(W\) here.

4.3.2 Choose \(B\), the first unused row \(k\) in \((1\ to\ 2^N-2)\) which has numBits\(=j\)

4.3.3 Drop the first element (numBits) to get \(b\).
- \(b = \) an \(N\)-bit vector = a BM row with numBits = \(j\).

4.3.4 \(s\) = a \(j\)-vector derived using \(b\) as a mask choosing elements of \(S\).
- \(p = \) Get product of the \(j\) elements of \(s\) (the number of cells in the sub array).

4.3.5 \(v = \) an \((N-j)\)-vector from \(~b\) as a mask choosing elements of \(S\).

4.3.6 \(vs = v; s\) [\(v\) catenated with \(s\)] is a permutation of \(S\), and has length \(N\).

4.3.7 \(V = \) Odometer of \(vs\) [creates a \(P\times N\) array of indexes; a row indexes a cell].

4.3.8 \(W = \) \(V\) reordered back to the original ordering of \(S\) using \(b\) and \(~b\).
- If LTR, reverse the columns of \(W\) here.
- \(P/p (p \neq 0)\) j-dimensional arrays: take \(p\) rows of \(W\) at a time. Get \(P/p\) arrays.

4.3.9 Output these \(P/p\ j\)-dimensional arrays of indexes.
- For \(p = 0\) \([j = 0]\), output the \(P\) indexes of \(A\) as \(P\) separate arrays.
- For \(p = P\) \([j = N]\), output the \(P\) indexes of \(A\) as \(1\) array. The \(P\) indexes of \(A\) are generated by odometering \(S\) itself. See 4.3.1 above.

4.4 To generate all \(j\)-dimensional sub array index sets of \(A\):
- Repeat from 4.3.2 with a new numBits = \(j\) row of BM if there are any left.
- This uses \(k\), the row index in the algorithm below.

4.5 To generate all \(j\) sub array index sets of \(A\):
- Repeat from 4.3.2 with a new value of \(j\) for all \(j\) in \(0\ to \(N\) [see comment in 4.3.1 above].

5. Special Use Of The Algorithm
If you want to generate sub arrays using a specific bit pattern, it must be input in APL (RTL) order. E.g., to generate the matrices using planes and hyperplanes in LTR \((0\ 0\ 1\ 1)\), it must be input as \((1\ 1\ 0\ 0)\). Another example: LTR Hyperplanes, rows, & columns \((1\ 0\ 1\ 1)\) must be input as \((1\ 0\ 1\ 1)\).

6. Variables used
- \(A\) The original N-D array.
- \(N\) The rank of \(A\).
S The shape N-vector of A
P Product over S = # cells in A.
BM Bit Matrix, 2^N x N. Each row is the binary of that
row’s row index in origin 0, [0 to 2^N - 1].
numBits # of 1 bits in a BM row.
BMN BM with numBits appended as a first column.
  j Number of bits = dimension of generated sub array
  & = # bits in row of BM.
k Row index of BM or BMN currently being worked
  on (which has numBits=j)
b N element row of BM.
s b mask S.
p Product over s = # of cells in a sub array.
v ~b mask S.
vs v catenate s, a permutation of S.
V Odometered vs
W Columns of V returned to original order of S.
P/p # of j-D subarrays for b.

7. A Partial Example (in RTL order)
N = 3, S = (2, 3, 4), P = 24, Ordering is LTR so this is an
S = (4, 2, 3) in RTL (2 rows, 3 columns, 4 planes).

7.1 The index matrix (NOT BM or BMN)
The index matrix is not used directly in the algorithm. It is shown
for clarity. It has 4x2x3 = 24 indices (rows) and 3 columns, one
for each dimension. The matrix of odometered indices (with row
number added in index origin 1) is:

<table>
<thead>
<tr>
<th>Row Index</th>
<th>3 Column Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1 1 1)</td>
</tr>
<tr>
<td>2</td>
<td>(1 1 2)</td>
</tr>
<tr>
<td>3</td>
<td>(1 1 3)</td>
</tr>
<tr>
<td>4</td>
<td>(1 2 1)</td>
</tr>
<tr>
<td>5</td>
<td>(1 2 2)</td>
</tr>
<tr>
<td>6</td>
<td>(1 2 3)</td>
</tr>
<tr>
<td>7</td>
<td>(2 1 1)</td>
</tr>
<tr>
<td>8</td>
<td>(2 1 2)</td>
</tr>
<tr>
<td>9</td>
<td>(2 1 3)</td>
</tr>
<tr>
<td>10</td>
<td>(2 2 1)</td>
</tr>
<tr>
<td>11</td>
<td>(2 2 2)</td>
</tr>
<tr>
<td>12</td>
<td>(2 2 3)</td>
</tr>
<tr>
<td>13</td>
<td>(3 1 1)</td>
</tr>
<tr>
<td>14</td>
<td>(3 1 2)</td>
</tr>
<tr>
<td>15</td>
<td>(3 1 3)</td>
</tr>
</tbody>
</table>

16 (3 2 1)
17 (3 2 2)
18 (3 2 3)
19 (4 1 1)
20 (4 1 2)
21 (4 1 3)
22 (4 2 1)
23 (4 2 2)
24 (4 2 3)

Figure 1. Index Matrix (Not BM or BMN)

This is the odometering of S. It stands for the single N
dimensional (N=3) array since there are 24 cell indexes. It also
can stand for the 24, 0 dimensional arrays that are the cells as
individual arrays. See 4.3.1 above.

7.2 BMN (Sorted)
The first column is the numBits (number of bits) in the bit matrix
BM row. The N following columns are the bits representing the
consecutive integers (0 to 2^2-1) = (0 to 7). BMN is the first
column cated on onto BM.

<table>
<thead>
<tr>
<th>numBits = j</th>
<th>Bit Matrix for N=3 in number order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>1 0 1</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

Figure 2. Unsorted BMN

<table>
<thead>
<tr>
<th>numBits = j</th>
<th>Sorted Bit Matrix by numBits = j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 1</td>
</tr>
<tr>
<td>2</td>
<td>1 0 1</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

Figure 3. Sorted BMN
7.3 The Number Of j-dimensional Rows
It can easily be shown that the count of the number of rows for a given j is $\binom{N}{j}$, since there are that many combinations of j things (bits = 1) from N bit positions. These will be the j-dimensional masking vectors b, determining j-dimensional sub arrays.

Figure 2. The Number of j-Dimensional Bit Vectors

<table>
<thead>
<tr>
<th>numBits = j</th>
<th>$\binom{3}{j}$</th>
<th>$#(\text{numBits} = j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

7.4 The Number Of j-dimensional Sub Arrays
A is a 3-dimensional array: (2x3x4) (LTR) or a (4x2x3) (RTL).

$T_0^0 \cdot (24) + T_0^1 \cdot (26) + T_0^2 \cdot (9) + T_0^3 \cdot (1)$ (any notation).

The expansion is taken from [1] and [2]. The expansion indicates 24, 0-dimensional cells, 26, 1-dimensional vectors, 9, 2-dimensional matrices, and one, 3-dimensional array (the original one).

The question is how do we explicitly generate the sub arrays counted here. The answer is the algorithm. Don't confuse the number of j-dimensional bit vectors and the number of j-dimensional arrays. We shall soon see they are related but not equal.

7.5 A 1-Dimensional Bit Vector (Row 1)
We now work in 0 origin indexing. We can neglect row 0 and the last row as they have been taken care of above in 7.1.

numBits = 1

b = (0 0 1) from row 1.

s = (0 0 3). This is extracted from S using b.

v = (4, 2), which is the complementary elements of S using b complement.

vs = (4, 2, 3), by catenating v and s. This is an N vector

V is odometered vs in index origin 1. (24 index triplets) V is folded here into two (12x3) sets to conserve space. Each triple is a 3-dimensional index of a cell in a sub array. The aggregation into sub arrays is done by aggregating the complimentary indexes.

Here these are the left most two dimensions. When the right most (corresponding to the bit in b) cycles it completes a sub array.

Figure 3. Odometered (4, 2, 3)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

W = V in this case.

p = (3).

The sub arrays (vectors) are the 3 row segments of W = V. Since the column index is moving fastest, the row next and the planes slowest, they are the rows in each plane of the 4 plane, 2 row, 3 column array A.

Each 3 row set is a vector indexed on the right most column. There are 8 = 24/3 vectors. The single bit vector from BMN generated many sub arrays, determined by the product of the complimentary dimensions of S.

7.6 A 2-Dimensional Bit Vector (Row 5)
For the case where k = 5:

numBits = 2

b = (1 0 1) row 5 of BM in index origin 0.

s = (4, 3), which is extracted by b from S = (4, 2, 3).

v = (2), which is the complement of S using b complement for extraction.

vs = (2, 4, 3) by catenating v and s. This is an N vector

V is odometered vs in index origin 1. (24 index triplets) V is folded here into two (12x3) sets to conserve space. Each triple is a 3-dimensional index of a cell in a sub array.

Figure 4. Odometered (2,4,3)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
1  3  3  2  3  3
1  4  1  2  4  1
1  4  2  2  4  2
1  4  3  2  4  3

Because V is not in the original order of S, we must reorder the columns of V to get W, in the original order. We interchange the left two columns since this yields (2, 4, 3) \rightarrow (4, 2, 3). (W =

Figure 5. Odometered (2, 4, 3) Ordered to (4, 2, 3)

\begin{align*}
1 & 1 & 1 & 1 & 2 & 1 \\
1 & 1 & 2 & 1 & 2 & 2 \\
1 & 1 & 3 & 1 & 2 & 3 \\
2 & 1 & 1 & 2 & 2 & 1 \\
2 & 1 & 2 & 2 & 2 & 2 \\
2 & 1 & 3 & 2 & 2 & 3 \\
3 & 1 & 1 & 3 & 2 & 1 \\
3 & 1 & 2 & 3 & 2 & 2 \\
3 & 1 & 3 & 3 & 2 & 3 \\
4 & 1 & 1 & 4 & 2 & 1 \\
4 & 1 & 2 & 4 & 2 & 2 \\
4 & 1 & 3 & 4 & 2 & 3
\end{align*}

p = (12).

The sub arrays (vectors) are the 12 row segments of W. P/p=2. Since the row index is moving slowest, each 12 indexes determine a horizontal plane cutting across the planes of A. Since the column index runs faster than the plane index of A, the generated planes are generated across the columns in a row and back into the planes of A.

Each 12 row set is plane. There are 2 - 24/12 planes. The single bit vector from BMN generated 2 sub arrays, determined by the product of the complimentary dimensions of S.

8. COMMENTS

We continue for all k = 1 to 6, yielding all of the full span sub arrays. If we require only the t-dimensional sub arrays, we use only those rows of BM that have numBits = t bits in them.

There is a Java 1.4 code that computes this algorithm for all j. It comprises 530 lines of richly commented code, 257 lines of which are executable code, in one driver, and ten support (primitive) methods. There is also an APL WS comprising a number of useful alternative versions. The one comparable to the Java code comprises 6 small functions of 75 total lines or 52 without comments. It computes the "inner loop" i.e., the generation of sub arrays for a given b (of j = numBits). There is an extensive DESCRIBE function in the WS which generates a 61 line description of the WS. Both are available from rfrank@pace.edu.

An appendix below has the main APL FNS that execute the algorithm for a given j. There is also a copy of the output of DESCRIBE.

9. THE GEOMETRY

Figure 6. The Geometry of a (2, 4, 3) LTR or 4x2x3) in RTL.

10. ACKNOWLEDGMENTS

My thanks to Adin Farkoff for many useful discussions and the loan of an APL system.

11. REFERENCES


12. APPENDIX: APL CODE

\[ \text{\textbackslash MAKEBM}[\textbackslash \theta]\text{\textbackslash V} \]
[0]  \text{Z+MAKEBM \textbackslash N};\text{\textbackslash TWOTON} \]
[1]  a MAKES A BIT MATRIX EACH ROW OF WHICH IS THE
[2]  a BINARY OF THE INTEGER \textbackslash K IN \textbackslash IO+0
[3]  \text{\textbackslash IO+0} \]
[4]  \text{\textbackslash TWOTON}+/\times;(\text{\textbackslash Np}2) \]
[5]  \text{Z+\textbackslash TWOTON} \]
[6]  \text{Z+(\textbackslash Np}2)\tau 2 \]
[7]  \text{\textbackslash V} \]

\[ \text{\textbackslash BMN}[\textbackslash \theta]\text{\textbackslash V} \]
[0]  \text{Z+BMN \textbackslash N};\text{BM} \]
[2]  a CREATES A 2+\textbackslash N ROW, \textbackslash N COLUMN BIT MATRIX BM WHOSE
[3]  a ROWS ARE THE BINARY OF THE ROW INDEX IN \textbackslash IO+0.
[4]  a THEN SUMS OVER ROWS TO GET THE NUMBER OF 1 BITS IN
[5]  a EACH ROW AND ADDS THESE SUMS TO A NEW COLUMN ON THE
[7]  \text{BM+MAKEBM \textbackslash N} \]
[8]  \text{V+\textbackslash V} \]
[9]  \text{Z+V,[1]BM} \]

\[ \text{\textbackslash BMNSORT}[\textbackslash \theta]\text{\textbackslash V} \]
[0]  \text{Z+BMNSORT BMN};\text{SORTVECTOR} \]
[1]  a GIVEN THE BIT MATRIX BMN WITH A FIRST COLUMN OF
[2]  a NUMBITS IN THE ROW, SORT THE ROWS ON THIS FIRST
[3]  a COLUMN.
[4]  \text{\textbackslash IO+0} \]
[5]  \text{SORTVECTOR+\&BMN[;0]} \]
[6]  \text{Z+BMN[SORTVECTOR];} \]
[7]  \text{\textbackslash V} \]

\[ \text{\textbackslash MAKEVW}[\textbackslash \theta]\text{\textbackslash V} \]
[0]  \text{Z+BMN1 MAKEVW KS;N;K;S;B;LITTLS;LITTLV;LITTLVS;V;W;LP;LVINDEXES;INDEXES}
[1]  ;LSINDEXES \]
[0]  a IO+1 ASSUMED FOR K INDEX AND ROW INDEX FOR BMN1
[1]  a IO+1 \]
[2]  \text{\textbackslash IO+1} \]
[3]  \text{N->(-1+BMN1)-1} \]
[4]  a N \]
[5]  \text{K++/1+KS} \]
[6]  a K AS A SCALAR \]
[7]  \text{S+1+KS} \]
\[ \begin{align*} &B \downarrow, \text{BM}i[K;] \\
&\text{LITTL}S \downarrow B/S \\
&\text{LINDEX}ES \downarrow B/\iota S \\
&\text{LP} \leftarrow \times, \text{LITTL}S \\
&\theta \leftarrow \text{LITTL}P = \text{'} \\
&\theta \downarrow \text{LITTL}S = \text{'} \\
&\theta \downarrow \text{LITTL}S \\
&\text{LITTL}V \leftarrow (\sim B)/S \\
&\text{LINDEX}ES \downarrow (\sim B)/\iota S \\
&\theta \leftarrow \text{LITTL}V = \text{'} \\
&\theta \downarrow \text{LITTL}V \\
&\text{INDEX}ES \downarrow \text{LINDEX}ES, \text{LINDEX}ES \\
&\text{LITTL}V \downarrow \text{LITTL}V, \text{LITTL}S \\
&\theta \downarrow \text{LITTL}V = \text{'} \\
&\theta \downarrow \text{LITTL}V \\
&V \leftarrow \theta \iota O + \text{LITTL}V \tau (\times / \text{LITTL}V) - \theta \iota O \\
&W \leftarrow V[; \text{INDEX}ES] \\
&Z \leftarrow (\times / S) \downarrow \theta \iota O, \text{LP}, N \iota W \\
\end{align*} \]
[22] \[ Z \cdot ((x/S) \cdot LP, LP, N) \cdot p \cdot W \]

\[ \text{V 2003-02-27 14.53.29 (GMT-5)} \]

\[ \text{\texttt{\textbackslash V SUBARRAYS[]}} \]

[0] \[ Z \cdot \text{SUBARRAYS KNS; N; S; P; K} \]
[1] \[ \text{reads in N, the dimension of the original array A} \]
[2] \[ \text{and S, the shape vector of A} \]
[3] \[ \text{computes P, the number of cells in A} \]
[4] \[ \text{outputs N, S, P} \]
[5] \[ \text{calls \texttt{SORTBMN BMN N}, which uses BMN to make the bitmatrix} \]
[6] \[ \text{and then sorts it on the first column, the numbits column} \]
[7] \[ K++ / 1 + KNS \]
[8] \[ N++ / 1 + KNS \]
[9] \[ S++ / KNS \]
[10] \[ P++ / S \]
[11] \[ N \cdot = ' \]
[12] \[ N \cdot = ' \]
[13] \[ S \cdot = ' \]
[14] \[ S \cdot = ' \]
[15] \[ P \cdot = ' \]
[16] \[ P \cdot = ' \]
[17] \[ K \cdot = ' \]
[18] \[ K \cdot = ' \]
[19] \[ Z \cdot \text{BMNSORT BMN N} \]
[20] \[ Z \cdot \text{STRIp Z} \]
[21] \[ Z \cdot Z \cdot \text{MAKEVW}(K, S) \]
[22] \[ Z \cdot Z \cdot = ' \]

\[ \text{V 2003-02-24 15.55.22 (GMT-5)} \]

Example:

A 3 dimensional array [4; 2; 3] of 4 planes, 2 rows, and 3 columns.
We choose row 2 of the bit array [0 1 0] standing for rows.
Since the rows will vary fastest, we will be generating columns. We expect 4x3 = 12 columns.

KNS

2 3 4 2 3

\[ \text{SUBARRAYS KNS} \]

\[ N = 3 \]
\[ S = 4 2 3 \]
\[ P = 24 \]
$K = 2$
$LITTLP = 2$
$LITTLS = 2$
$LITTLV = 43$
$LITTLVS = 432$
$Z = $

<table>
<thead>
<tr>
<th>Plane 1</th>
<th>Plane 2</th>
<th>Plane 3</th>
<th>Plane 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 Column 1</td>
<td>2 1 1 Column 1</td>
<td>3 1 1 Column 1</td>
<td>4 1 1 Column 1</td>
</tr>
<tr>
<td>1 2 1</td>
<td>2 2 1</td>
<td>3 2 1</td>
<td>4 2 1</td>
</tr>
<tr>
<td>1 1 2 Column 2</td>
<td>2 1 2 Column 2</td>
<td>3 1 2 Column 2</td>
<td>4 1 2 Column 2</td>
</tr>
<tr>
<td>1 2 2</td>
<td>2 2 2</td>
<td>3 2 2</td>
<td>4 2 2</td>
</tr>
<tr>
<td>1 1 3 Column 3</td>
<td>2 1 3 Column 3</td>
<td>3 1 3 Column 3</td>
<td>4 1 3 Column 3</td>
</tr>
<tr>
<td>1 2 3</td>
<td>2 2 3</td>
<td>3 2 3</td>
<td>4 2 3</td>
</tr>
</tbody>
</table>

**Call Structure:** [Given K And S] OR [B And S]

**SUBARRAYS** OR **BITMAKEW**

**BMW**

**MAKEBM**

**BMNSORT**

**STRIP**

**MAKEVW**

**DESCRIBE**

*This program computes the sub arrays of a given array structure by exhibiting the index sets that define the sub arrays.*

An array structure is defined by its dimension and shape vector.

**Z-SUBARRAYS KNS**

Is the master function.

It takes one vector argument KNS:

* K is the row of the bit matrix BMN that determines which dimensions will vary in this subarray structure.
* N is the dimension of the starting array structure
* S is the shape vector of the starting array structure

**SUBARRAYS uses BMN, BMNSORT, STRIP, and MAKEVW**

**Z+(STRIP(BMNSORT BMW N) MAKEVW (K,S))**

**Z-BMN N**

Given the number of bits, N, = the number of columns;
creates a Z*N row, N column bit matrix BM whose rows are the binary of the row index in [0]*N*.
THEN SUMS OVER ROWS TO GET THE NUMBER OF 1 BITS IN EACH ROW AND ADDS THESE SUMS TO A NEW COLUMN ON THE LEFT OF BM CREATING BMN

BMN CALLS MAKEBM

Z-Make BMN N
MAKES AN 2* N ROW, N COLUMN BIT MATRIX EACH ROW OF WHICH IS THE BINARY OF THE INTEGER K IN \[ \text{IO} \]

Z-BMNSORT BMN
BMN IS THE BIT MATRIX WITH AN ADDED FIRST COLUMN.
THE FIRST COLUMN CONTAINS THE COUNT OF 1 BITS IN EACH OF ITS ROWS
BMN SORTS THE ROWS OF BM IN ASCENDING ORDER OF THIS FIRST COLUMN.
THIS SORTED FORM OF BM IS RETURNED.

Z-STRIP BM
REMOVES THE FIRST ROW OF BM (ALL 0S) AND THE LAST ROW (ALL 1S)

Z-BMN1 MAKEWV KS

BMN1 IS THE STRIPPED BIT MATRIX OF BINARY REPS OF ROW INDEXES.
KS IS K, THE ROW INDEX TO BE USED ON BMN1, AND S THE SHAPE VECTOR OF THE ORIGINAL ARRAY STRUCTURE BEING SUBARRAYED.

B-THE KTH ROW OF BMN1 IS EXTRACTED AND ITS FIRST ELEMENT Dropped.
THE FIRST ELEMENT CONTAINS THE NUMBER OF BITS IN THE ROW, WHICH EQUALS THE DIMENSIONS OF THE SUB ARRAYS TO BE GENERATED.

LITTLS IS THE ELEMENTS OF S EXTRACTED BY THE BIT VECTOR B.

LP IS THE LITTLE P, THE NUMBER OF CELLS IN A SUB ARRAY.
LP IS THE PRODUCT OVER LITTLS.

LITTLV IS THE REST OF S EXTRACTED BY (~B).

THE PRODUCT OVER LITTLV IS THE NUMBER OF SUB ARRAYS TO BE GENERATED, AND IS EQUAL TO P*LP OR \((\times/S)\times(\times/LITTLV)\times(\times/LITTLS)\), WHERE THE TOTAL NUMBER OF CELLS IN THE ORIGINAL ARRAY IS \((\times/S) = ((\times/LITTLV)\times(\times/LITTLS)))

LITTLVS IS THE CATENATION OF THE LITTLV AND LITTLS GIVING A PERMUTATION OF S SUCH THAT WHEN ODOMETRED, THE DIMENSIONS DETERMINED BY LITTLV VARY SLOWEST.
V+THE ARRAY OF CELL INDEXES GENERATED BY ODOMETERING LITTLVS.

W=V WITH COLUMNS SORTED BACK TO THE ORIGINAL ORDERING OF S.

Z=RESHAPED W, WITH P\times LP PLANES, LP ROWS, AND N COLUMNS SO THAT IT PRINTS OUT CORRECTLY SHOWING THE SETS OF N DIMENSIONAL INDEXES FORMING THE P\times LP ARRAYS OF DIMENSION p\LITTLVS.

Z+B BITMAKEVW S

THIS IS THE SAME AS MAKEVW EXCEPT IT TAKES IN B AND S.
S IS A SHAPE VECTOR. B IS A BIT VECTOR OF THE SAME LENGTH AS S.
B IS USED TO CHOOSE THE ELEMENTS OF S (THE DIMENSIONS) TO USE IN GENERATING SUB ARRAYS

BITMAKEVW THEN USES THE ALGORITHM TO GENERATE THE SUB ARRAYS AND TO OUTPUT THEM.

----------------------------------------------- END OD CODE -----------------------------------------------
A New “Dual-View” Diagram of Array Structure

Ronald I. Frank, DPS (Computing)
Associate Professor School of CS & IS
Pace University
325 Goldstein Hall
861 Bedford Road
Pleasantville, NY 10570
1-(914) 773-3444
Email rfrank@pace.edu

ABSTRACT
First we present a new structural decomposition of regular arrays that is derived elsewhere [4]. Second, we list a few special cases of the decomposition. Finally we give 11 examples of arrays, their decomposition, their egg crate view, and their dual view (which better represents their true structure).

We present this graphical presentation form (a diagrammatic notation) for a regular array. The notation is a “dual” of the usual “egg crate” view. This view makes clear and intuitive the otherwise unintuitive results derived from the newly found array expansion in [4]. The array expansion results are all structural therefore so are these pictorial results. They do not use knowledge of the particular data contents of the arrays.

Categories and Subject Descriptors
E.1. [Data Structures] Arrays

General Terms
Documentation.

Keywords
Connectivity, Array Structure, Diagramming, Null-Array Expansion, Visualization

1. INTRODUCTION
We present some standard terminology and then the new expansion. We then present the dual view of an Array. The expansion is used to argue that the dual is a better representation.

2. TERMINOLOGY
We take as known or intuitive the terminology:
• Regular Array (all dimensions filled to full span)
• Equilateral Regular Array (ERA) (all axes of equal length) also called a cube
• Non-equilateral Regular Array (NERA) (axes of possibly different length) also called a block or brick
• Shape Vector (list of dimension lengths of array) 0 in a position means 0 length axis
• Dimension or Rank (number of positions in the shape vector)
• Axis (position in the shape vector)
• Length of Axis (value of position in the shape vector)
• Null Array (at least one 0 in the shape vector)
• Index (vector of values indicating a cell in the array) for our purposes we use index origin 1
• Index (a single position (axis) in the index defined above)
• Index Value (of the vector or of a single position (axis))
• Full Set of Index Values indicates a cell
• Scalar (has no dimension, no length; also considered the element in a cell)
• Domain of the index mapping (the set of cells)
• Co domain of the index mapping (the set of scalar “contents” of the cells)
  • Even though they may have intrinsic structure, that structure is not used here.
  • This discussion is not concerned with these cell contents.
• The axes are independent of each other.
• Adjacency of cells is implied by the index ordering’s adjacency

3. THE ARRAY EXPANSION
\( T^N \{n_1, n_2, n_3, \ldots, n_{N-2}, n_{N-1}, n_N \} = \)

\[ \sum_{j=0}^{N} T_0^j \left( \sum_{N}^N \left[ \left( n_1 \right) \times \left( n_2 \right) \times \cdots \times \left( n_{N-1} \right) \times \left( n_N \right) \right] \right) \]

From [4] we have the generating polynomial of the count of all of the full-span sub arrays of a regular array:

This is a purely algebraic-combinatorial result. We observe that the first line can be thought of as representing an array. The superscript N is the dimension of the array and the subscript vector is the shape vector of the array.
The right hand side algebraic expansion can be interpreted as having numerical coefficients multiplying null arrays of dimension j. The inner summation is over all combinations of N shape vector positions taken (j) at a time. The outer sum is over all dimensions from 0 to N. The expansion is invariant under arbitrary permutations of the axes. The expansion says nothing about the contents of the array's cells. The coefficients also can be derived using a purely combinatorial argument on the shape vector.

3.1 Special case: The General Equilateral Array (Cube)
When all of the N-dimensional lengths are equal the equation becomes:

\[ T_{n}^{N} = \sum_{j=0}^{N} T_{j}^{j} \binom{N}{j} n^{N-j} \]  

3.2 Special Case: n=0
When all of the j-dimensional lengths are equal to 0, we notate the corresponding null array as

\[ T_{0}^{j} \]  

an atom in the expansion (1.2).

3.3 Special Case: n=0, j=0
When j itself is 0 we have the 0-dimensional array (the scalar):

\[ T_{0}^{0} \]  

4. PICTURING ARRAYS TO CLEARLY SHOW THESE RESULTS AND TO SHOW ARRAY SUBSTRUCTURE
The usual picture of an array is the "egg crate" view where each cell is represented as a box, even for a one-dimensional vector! There is another "dual" diagram type that pictures nodes and edges. This is the usual processor and connections diagram used when discussing entities such as hypercube machines [1, 2, 3, 5]. The equation (0.1) suggests a modification of this view that better represents data arrays in the sense that it shows the expansion results in a clearer way. We show the "dual-view" then the "egg crate view".

The vector of length 1. \[ T_{1}^{1} = T_{0}^{0} + T_{0}^{1} \] (Egg crate view ambiguity c.f. matrix 1x1.)

Figure 3.

The vector of length 2. \[ T_{2}^{1} = 2T_{0}^{0} + T_{0}^{1} \]

Figure 4.

The vector of length 3. \[ T_{3}^{1} = 3T_{0}^{0} + T_{0}^{1} \] (Notice segments don't count.)

Figure 5.

The matrix of length (1x1). \[ T_{1}^{2} \equiv T_{1,1}^{2} = T_{0}^{0} + 2T_{0}^{1} + T_{0}^{2} \] (C.f.1-vector egg crate.)

Figure 6.

The matrix of length (2x2). \[ T_{2}^{2} \equiv T_{2,2}^{2} = 4T_{0}^{0} + 4T_{0}^{1} + T_{0}^{2} \]

Figure 7.

The 3-D array of length (1x1x1). \[ T_{1,1,1}^{3} = T_{0}^{0} + 3T_{0}^{1} + 3T_{0}^{2} + T_{0}^{3} \]
The 3-D array of length (2x2x1).

\[ T_{2,2,1}^3 = 4T_0^0 + 8T_0^1 + 5T_0^2 + T_0^3 \]

It is interesting to note in this last figure on the left that the 6th plane of the cube is missing in the diagram (no nodes in back) and in the expansion. This diagrammatic notation more accurately represents the array structure.

The 3-D array of length (2x3x4).

\[ T_{2,3,4}^3 = 24T_0^0 + 26T_0^1 + 9T_0^2 + T_0^3 \]

First, we count the 24 scalars, 12 in the top plane and 12 in the bottom plane. There is one 3-dimensional object.

However, the planes and vectors require some discussion. There are 3 vertical planes running from front to back on the left, the right, and in the middle. There are 4 vertical planes running from left to right in the front, the back, and inside adjacent to the front and back planes. There is 1 top plane and 1 bottom plane for a total of 9 planes. QED.

There are 12 vertical vectors connecting two notes at a time. There are 8 horizontal vectors running from left to right, 4 in the top plane and 4 in the bottom plane. There are 6 horizontal vectors running from front to back, 3 in the top plane and 3 in the bottom plane for a total of 26 vectors. QED.

The Shape holder (2x3x0).

\[ T_{2,3,0}^3 = 0T_0^0 + 6T_0^1 + 5T_0^2 + T_0^3 \]

\[ T_{2,3,0}^3 = T_0^0 \otimes T_0^0 = (0T_0^0 + T_0^1) \otimes (6T_0^0 + 5T_0^1 + T_0^2) = \\
(6T_0^0 + 5T_0^1 + T_0^2) = (0T_0^0 + 6T_0^1 + 5T_0^2 + T_0^3) \]

We interpret this as the 0 term kills the 6 scalars, the 5 vectors and the plane in the original array but the vector in the new third dimension adds 6 vectors where there were scalars. This gives rise to 5 planes all extending into the third dimension. The result is a 3D array containing structure but no cells. This is an articulated null array (as opposed to the atoms such as \( T_0^0, T_0^1, T_0^2 \), and \( T_0^3 \)).

6 vectors, 3 on top and 3 on bottom running front to back (into the 3rd dimension). There are 5 planes: top horizontal, bottom horizontal, left vertical (front to back), middle vertical (front to back), and right vertical (front to back). This fully organizes the (2x3x0) arrangement of 6 vectors into a 3-dimensional object.
QED. There are no front or back planes because there are no vector edges to define them.

5. SUMMARY
We have shown a diagrammatic notation that better represents multi-dimensional arrays in that it faithfully exhibits the substructure of regular arrays as given by the newly found regular array expansion (0.1). More strongly, we can state that the egg crate view is misleading (cf. the one element vector vs. the one element matrix). It is also harder to manipulate to find substructure, and so is the wrong way to picture array structure.

6. ACKNOWLEDGMENTS
My thanks to Adin Falkoff for many useful discussions.

7. ACKNOWLEDGMENTS
Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

8. REFERENCES


**Algorithm Driver**

Computes all of the sub arrays of a given array A.

A must be of dimension N >= 1.

A must have axes lengths >= 1 \{ S[i] >= 1 for all i \}.

The algorithm is in index origin 1

The code is in Java index origin 0

main() reads in N and S, every thing else is generated.

N (scalar) is the dimension (rank) of the array A

TwoToN is 2^N.

S (vector) is the N element shape vector of A.

P is the product of the elements of S,

(= the number of 0-D cells in A or rows of M).

M[][] (matrix) of P rows containing all possible cell indexes of A.

B[][] (matrix) is the matrix of all TwoToN bit combinations of N bits.

B's rows contain numbers from 0 to TwoToN-1.

r (scalar) is the iteration index corresponding to the TwoToN rows of B.

B's row index r runs from r = 0 to r = TwoToN-1.

WE ONLY OUTPUT r = 1 TO r = TwoToN-2 LEAVING OUT ALL 0s AND ALL 1s.

This corresponds to rows 2 TO TwoToN-1 in origin 1 row indexing.

WE DO OUTPUT THE INDEX SET OF ALL 0 DIMENSIONAL CELLS.

b (vector) is the current bit vector (row of B).

numBits is the number of 1 bits in b, the current row of B.

numBits = j = dimension of the sub array of A being generated

B[][][0] (vector) contains numBits, the number of 1 bits in each row.

s (vector) is the elements of S corresponding to the 1 bits of b.

p (scalar) is the product of the elements of s.

v (vector) is the elements of S corresponding to the 0 bits of b.

vs (vector) is v catenated with s, an N-vector.

vs is a permutation of S.

V[][] (matrix) is the odometered vs as a matrix. It has P rows.

W[][] (matrix) is V[][]'s columns returned to the order of S.
public class AlgorithmDriver
{
    public static void main (String[] args)
    {
        // read in N

        int  N = Integer.parseInt(args[0]);

        // Check N for positivity >= 1
        if (N <= 0)
        {
            System.out.print("INPUT N must be >= 1");
            System.out.println(" TERMINATING RUN.");
            System.out.println("================================");
            System.exit(0);
        } // end check negative or 0 N

        // read in the N elements of S

        int[]  S = new int[N];

        // Check N elements of S for positivity >= 1

        for(int i = 1; i <= N; i++)
        {
            S[i-1] = Integer.parseInt(args[i]); // maximum position value

            if (S[i-1] <=0)
            {
                System.out.print("INPUT position i = " + i);
            }
System.out.println("", INPUT S[i] must be >= 1. TERMINATING RUN.");
System.out.println("=---------------------------------=");
System.exit(0);
} // end check negative or 0 elements
} // end read in S

////////////////////////////////////////////////////////////////////////////////////

// Compute P, the product of the elements of S,
// (number of cells in A or rows of M)

int P = 1;

for (int i = 1; i <= N; i++)
{
    P=P*S[i-1];
}

// Compute 2^N

int TwoToN = (int)Math.pow(2, N);

// Print out master S of A
System.out.println("S is: (");

for(int i = 1; i <= N-1; i++)
{
    System.out.print(S[i-1]+", ");
}
System.out.print(S[N-1]+"");

// Generate all of the indices of the 0-D cells of A
// M[][] has P rows. Then write out M[][].
int M[][]=Primitives.Odometer(N, S);

    Primitives.WriteM(P, N, 0, 0, M, "M");

    // Generate all of 2^N rows (bit configurations) with numbits at the
    // head of each row. This is done by odometering and S with N 2s
    // and then subtracting 1 from each element then adding a column
    // containing each row's numBits.

    int B[][] = Primitives.MakeBitMatrix(N);

    System.out.print("This is the Bit Matrix. The first column");
    System.out.print(" is the numBits in the row.");
    if (N == 1)
    {
        System.out.print("\n\tIf N = 1");
        System.out.println(" the Bit Matrix is is NULL.");
        System.out.println("\tThe 0-D cells (would be first row)");
        System.out.println(" are the same as the 1-D vector ");
        System.out.println("\twould be second and last row.");
        System.out.println(" Therefore there are no further outputs.");
        System.exit(0);
    }

    Primitives.WriteM( (TwoToN-2), N+1, (TwoToN-2), 2, B, "B");

    // Generate the working holder of an N-bit vector, b

    int[] b = new int[N];

    // The master loop that runs through all of the bit configurations
    // (combinations of N axes j at a time, for all j
    // except we leave out two combos, all 0s and all 1s.
/ * j = numBits = dimension of sub arrays of A *

for (int r = 1; r < (TwoToN-1); r++)
{
    System.out.print("Bit Matrix Row r = "+ r + ", ");

    b = Primitives.extract_b(N, B, r);
    System.out.print("b = ");
    for (int i = 0; i < N; i++) {System.out.print(b[i]+" ");}

    // Get the number of 1 bits in this vector b
    int numBits = B[r-1][0];
    System.out.println("", and numBits = "+ numBits);

    // Create the s for this r (numBits)
    int[] s = new int[numBits];

    // Get the axis lengths corresponding to the bit positions
    s = Primitives.extract_s(N, S, b, numBits);

    // Define p (for s, corresponds to P for S) the product of
    // the axis lengths in s.
    int p = 1;
    for (int i = 1; i <= numBits; i++)
    {
        p = p * s[i-1];
    }
System.out.println("Product of s = # of rows in these subarrays = \( p = " + p);}

// Create the \( v \) for this \( b \), that holds the axis lengths of
// the axes corresponding to the 0 bits of \( b \)

int[] \( v \) = new int[N-numBits];

\( v = \) Primitives.extract_v(N, S, b, numBits);

// Create \( v_s \), the catenation of \( v \) and \( s \) yielding an \( N \) vector of
// a permutation of \( S \)

int[] \( v_s \) = new int[N];

\( v_s = \) Primitives.vCatenate_s(N, \( v \), \( s \), numBits);

// Create \( V[][] \), the matrix of odometered values of \( v_s \).
// \( V[][] \) has \( p \) rows.

int \( V[][] \)=Primitives.Odometer(N, \( v_s \));

// Permute the columns of \( V[][] \) back to their normal order

int \( W[][] \)=Primitives.reorder(N, numBits, P, \( V \), b);

Primitives.WriteM( P,N, p, numBits, \( W \), "W");
}
} // end of looping through the rows of \( B \).
}
} // end main()

} // end class AlgorithmDriver.
import java.util.*; // For Classes Arrays and Comparator

public class Primitives {

//****************** Comparator ARRAY_ROW_COMPARE

// ************ void bumpM (int N, int pos, int j, int[][] M, int[] S)
// ************ int [] extract_b (int N, int[][] B, int k)
// ************ int [] extract_s (int N, int[] S, int[] b, int numBits)
// ************ int [] extract_v (int N, int[] S, int[] b, int numBits)
// ************ int [][] MakeBitMatrix (int N)
// ************ int [][] Odometer (int N, int S[]) 
// ************ int [][] reorder (int N, int numBits, int P, int[][] V, int[] b)
// ************ int [] vCatenate_s (int N, int[] v, int[] s, int numBits)
// ************ void WriteM (int P, int N, int p, int numBits, int[][] M, String st)

// *** ARRAY_ROW_COMPARE Used in sort inside MakeBitMatrix
// *** bumpM Used in Odometer - does the real recursive work
// *** extract_b Gets a bit vector row from BitMatrix
// *** extract_s Gets the corresponding elements of S using b
// *** extract_v Gets the complementary elements of S using ~b
// *** MakeBitMatrix Makes Matrix of all the bit combos of an N-bit bit vector
// *** reorder Reorders V[][] (from vs) back to original order of S
// *** vCatenate_s Concatenates v with s
// *** WriteM A parameterized matrix output routine

// Used in sorting the bit matrix on numbits

static final Comparator ARRAY_ROW_COMPARE = new Comparator() {
    public int compare( Object o1, Object o2)
{  int[] a1 = (int[]) o1;  int[] a2 = (int[]) o2;

  if ((a1[0]) < (a2[0])) return -1;  
  if ((a1[0]) == (a2[0])) return 0;

  return 1;
}

/*-------------------------------------------------------------------------------

// Used in odometer.
// BumpM recursively calls itself.
// Each call starts at the right and uses
// the remembered last state of the position,
// A return is called every time the remembered state is
// incremented by one, even if there has been internal overflows.
// The calling is halted in odometer by keeping track of how many
// configurations have been outputted.

public static void bumpM(int N, int pos, int j, int[][] M, int[] S) {

  M[j-1][pos-1]=M[j-1][pos-1]+1;

  if(M[j-1][pos-1] > S[pos-1]) {
    M[j-1][pos-1] = 1; // We have position overflow.
    // Reset this position and check the one
    // to the left.  j is never 1, so j-1 is never 0.

    bumpM(N, (pos-1), j, M, S);

    return;
  }
}
return; // just to make it clear

} // end bumpM

// Extract the kth row of the bit matrix
// k in index origin 1, running from 1 to TwoToN-2
// K leaves out first and last rows of bit matrix
// which is all 0s and all 1s.

public static int [] extract_b(int N, int[][] B, int k) {
  int [] b = new int[N];

  for(int h = 1; h <= N; h++)
  {
    b[h-1] = B[k-1][h];
  }

  return b;
}

} // end make_b

// Extract s from S using b, the kth row of the bit matrix

public static int [] extract_s(int N, int[] S, int[] b, int numBits) {
  int s[] = new int[numBits];

  int i = 0;

  for(int k = 0; k < N; k++)
  {
    if(b[k] == 1)
```java
{ 
    s[i] = S[k];
    i++;
} // end if
} // end for
return s;
} // end extract_s

/******************************************************************************

   // Extract v from S using the complement of b compliment

public static int [] extract_v(int N, int[] S, int[] b, int numBits)
{
    int v[] = new int[N-numBits];

    int i = 0;

    for(int k = 0; k < N; k++)
    {
        if(b[k] == 0)
        {
            v[i] = S[k];
            i++;
        } // end if
    } // end for
    return v;
} // end extract_v

/******************************************************************************

public static int[][] MakeBitMatrix(int N)
{
    int[] S     = new int[N];
    int TwoToN = (int)Math.pow(2, N);
    int[][] B   = new int [TwoToN][N];
    int[][] BM  = new int [TwoToN-2][N+1];
```
// Set S to be a vector of N 2s
for (int i = 0; i < N; i++) {S[i] = 2;}

// Odometer S getting a matrix of 1s and 2s
B = Odometer (N, S);

// Make it a matrix B of 0s and 1s by subtracting 1 everywhere
for (int i = 0; i < TwoToN; i++)
{
    for (int j = 0; j < N; j++)
        B[i][j] = B[i][j] - 1;
} // end subtract 1 from B

// Now create a new matrix BM with an added first column
// and put the sum of bits into each row's first element
// but DROP the all 0s first row and N 1s last row
for (int i = 0; i < TwoToN - 2; i++)
{
    BM[i][0] = 0;
    for (int j = 0; j < N; j++)
    {
        BM[i][j + 1] = B[i + 1][j];
        BM[i][0] = BM[i][0] + BM[i][j + 1];
    } // end adding up numbits in each row j
} // end for i - all rows

// sort on the first column number of non-zero bits
Arrays.sort (BM, ARRAY_ROWCOMPARE);

return BM;

}*******************************
public static int[][] Odometer (int N, int S[])
{
    // This is a control driver that sets up and calls
// BumpM which generates a row using recursive calls.

// The calling of BumpM is halted by keeping track of how many
// configurations have been outputted.

// Get P, the number of indices

int P = 1;
for (int i = 1; i <= N; i++)
{
    P = P * S[i - 1];
}

// Create index set array M

int[][] M = new int[P][N];

// Initialize the first row of M

for (int i = 1; i <= N; i++)
{
    M[0][i - 1] = 1;
}

int rowCountj = 2;

while (rowCountj <= P)
{
    for (int k = 0; k < N; k++)
    {
        M[rowCountj - 1][k] = M[rowCountj - 2][k];
    }

    bumpM(N, N, rowCountj, M, S);  // bump M[][]

    rowCountj++;
}
return M;
}
// end Odometer

public static int[][] reorder(int N, int numBits, int P, int[][] V, int[] b)
{
    int W[][] = new int[P][N];

    int v0 = 0; // pointer into V for b = 0 choices (v)
    int v1 = 0; // pointer into V for b = 1 choices (s)

    for(int k = 0; k < N; k++)
    {
        if(b[k] == 0)
        {
            for (int i = 0; i < P; i++)
            {
                W[i][k] = V[i][v0];
            } // for move column
            v0++;
        } // end if == 0

        if(b[k] == 1)
        {
            for (int i = 0; i < P; i++)
            {
                W[i][k] = V[i][(N-numBits)+v1];
            } // for move column
            v1++;
        } // end if == 0
    } // end for N columns of W

    return W;
public static int[] vCatenate_s(int N, int[] v, int[] s, int numBits)
{
    int vs[] = new int[N];

    for(int k = 0; k < (N-numBits); k++)
    {
        vs[k] = v[k];
    } // end if

    for(int k = (N-numBits); k < N; k++)
    {
        vs[k] = s[k - (N-numBits)];
    } // end if

    return vs;
} // end vCatenateS

//*****************************

public static void WriteM(int P, int N, int p, int numBits, int[][][]M, String st)
{
    System.out.println("\n + st + "[][] for N = " + N + " and P = " + P);
    if (p==0)
    {
        System.out.println("There are P = " + (P) + ", " + numBits + "-D arrays here.");
        System.out.println("These are shown as the index set of the whole array.");
    } // end if;
    else
    System.out.println("There are P/p = " + (P/p) + ", " + numBits + "-D arrays here.");

    System.out.println(" ");

    for( int i = 0; i < P; i++)
    {

// Prevents division by p = 0 for the 0-D case inside the if
// AND suppresses an extra blank line before the other cases

if( (p != 0) && (i != 0) )
{
    // Otherwise separates index sets for different j-D Arrays
    if(0 == (i*p) ) System.out.print("\n");
}

for( int j = 0; j < N; j++)
{
    System.out.print(M[i][j] + " ");
}
System.out.print("\n");

System.out.print("\n");

} // end WriteM

Thông số

} // end class Primitives
The School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. *Technical Reports* is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from the pages you have submitted. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix  
School of CS & IS - Suite 412 Graduate Center  
Pace University  
1 Martine Avenue  
White Plains, NY 10606-1932