Proceedings of Student-Faculty Research Day, CSIS, Pace University, May 5th, 2017

Regular (N-Dimension) Arrays are Trees

Dr. Ronald I. Frank
Seidenberg School of CSIS, Pace University, Pleasantville, New York

***Abstract*—Starting with some careful definitions of Regular Array, Shape List, Index List, Lattice, and Odometering in both the FORTRAN order and then the APL order, we first show a recursive construction algorithm for building N-D (N-Dimensional) arrays defined by a given shape list. Using this construction, we map the arbitrary N-D array to a tree in more than one way. Along the way we clarify: array visualization methods, historical arrangements of the array shape list, and the relationship between them. We always show the (ROW, COLUMN PLANE) diagram for array visualization.**

***Index Terms*—APL order, Array, Duality, Egg-Crate View, FORTRAN order, Lattice, Matrix, Pigeon-hole View, Tree, Wire-Frame View.**

I. NOTATION

A. A Regular Array is an Array defined by a Shape List. This Implies That the Array is Full in All Dimensions; it is not ragged.

B. Array Shape List

We use the notation $(n_1, n_2, \dots, n_{N-1}, n_N)$ to represent an N-D array whose k^th^ dimension has length (count) n_k).

C. Array Index List

We use the notation $[i_1, i_2, \dots, i_{N-1}, i_N]$ to indicate a cell in an N-D array whose k^{th} dimension position is at i_k).

D. Array "Odometering"

We use the term "odometering" to indicate the process of systematically outputting the indices of all of an array's cells in the order indicated by its shape list.

Given an array defined by a shape list $(n_1, n_2, \dots, n_{N-1}, n_N)$, we start with the index value $[1_1, 1_2, \dots, 1_{N-1}, 1_N]$ and analogously to a car odometer, we count up to the final value $[n_1, n_2, \dots, n_{N-1}, n_N]$. This systematically generates all of the cell indexes of the array. The total number of cells is $\{n_1 x n_2 x \dots x n_{N-1} x n_N\}$. The index value 0 does not appear.

E. Array Content

The content of an array cell is defined by a mapping from the index of the cell to a codomain of content. We do not consider content here.

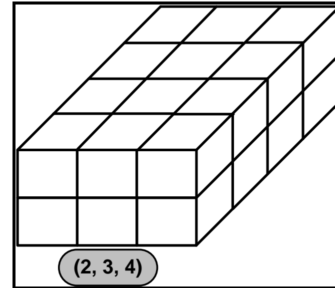
F. Dimension of an Array Cell

An array cell has dimension 0. An array of one cell can be of any non-zero dimension. For any $(N > 0)$ $(1_1, 1_2, \dots, 1_{N-1}, 1_N)$ is an (N-D) array of 1 (0-D) cell.

II. ARRAY VISUALIZATIONTION

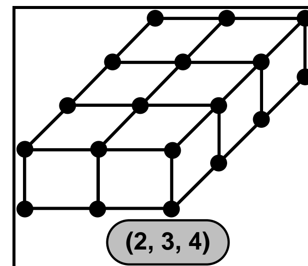
A. Egg-crate or Pigeon-hole View

This is the usual Array picture found in texts. For example, a (2, 3, 4) would look like this: 2 rows, 3 columns, 4 planes.



B. Wire-frame or Lattice view (Array Dual View)

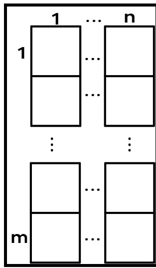
An equivalent visualization but less used in programming is:



This is actually a better representation in that the cells are clearly 0-D entities. Structurally and combinatorically, these two views are equivalent. For this reason we really prefer to refer to RALs – regular array-lattices.

C. Discussion of the Shape List vs. Visualization View

Since the most familiar array is a matrix, there are two historical traditions relating the shape list to the visualization. For a matrix, (m, n) usually means m rows and n columns. The usual visualization is



The row index increase downward and the column index increases left to right. If there were to be a 3rd dimension (planes) of length k, it would often be diagrammed receding into the paper. The plane index increases with depth into the page. This is our canonical visualization.

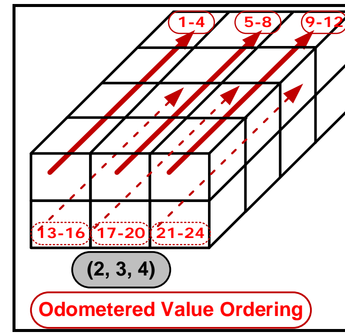
Unfortunately, there are two ways of showing this in a *shape list*: the "FORTRAN" way (n, m, k) and the "APL" way (k, m, n). The effect on odometering is the following:

> (m, n, k) would generate the cells along vectors anchored in the front face of the array (m, n) and running back through the planes (k). The anchors would move along columns in a row and then change rows.

We use R, C, P, HP, HHP, for row column, plane, hyper plane, hyper-hyper plane etc. The two shape list forms are: FORTRAN - (R, C, P, HP, HHP) and APL- (HHP, HP, P, R, C).

FORTRAN Shape List (2, 3, 4) (R, C, P) Convention. [Cross plane vectors anchored in the front plane].

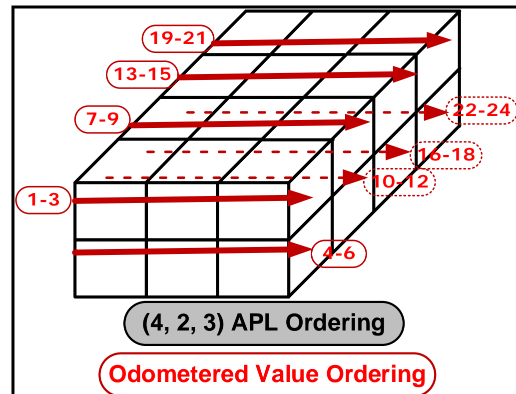
#	(R,	C,	P)
1	1	1	1
2	1	1	2
3	1	1	3
4	1	1	4
5	1	2	1
6	1	2	2
7	1	2	3
8	1	2	4
9	1	3	1
10	1	3	2
11	1	3	3
12	1	3	4
13	2	1	1
14	2	1	2
15	2	1	3
16	2	1	4
17	2	2	1
18	2	2	2
19	2	2	3
20	2	2	4
21	2	3	1
22	2	3	2
23	2	3	3
24	2	3	4



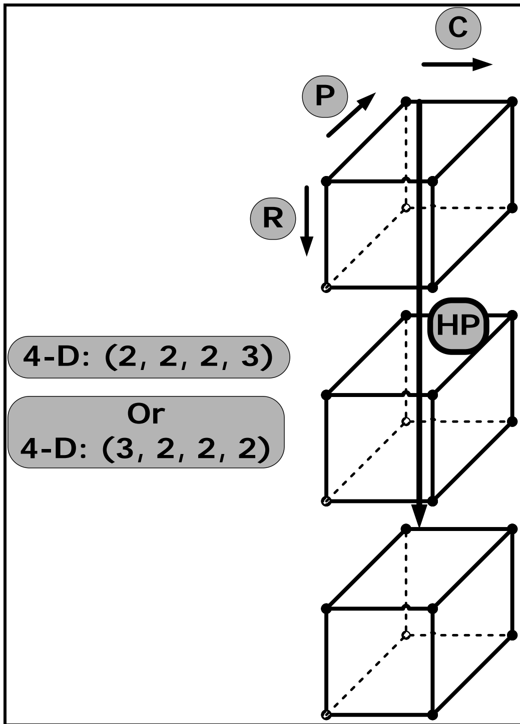
> (k, m, n) would generate the cells in a plane row-wise., then increase the plane index.

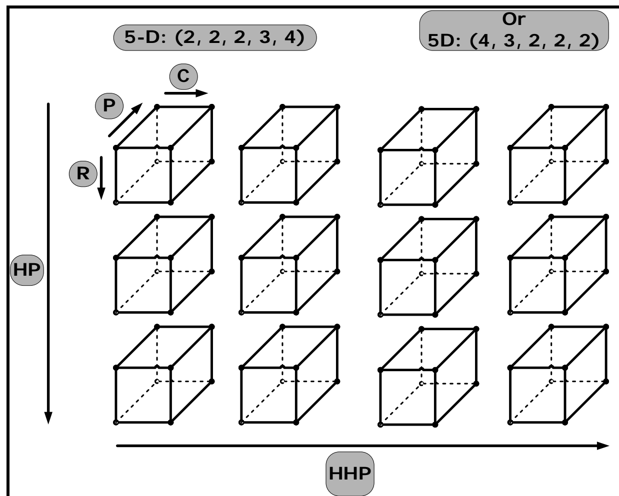
APL Shape List (4, 2, 3) (P, R, C) Convention [Row-wise by plane].

#	(P,	R,	C)
1	1	1	1
2	1	1	2
3	1	1	3
4	1	2	1
5	1	2	2
6	1	2	3
7	2	1	1
8	2	1	2
9	2	1	3
10	2	2	1
11	2	2	2
12	2	2	3
13	3	1	1
14	3	1	2
15	3	1	3
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



One can argue that in the multi-dimensional case, the APL form is clearly easier to visualize since we are generating the cells in 3-D groups maintaining the usual matrix visualization (R, C) for the planes. The array is organized into HP and HPP form. The 4-D case (HP, P R C) has the 3-D cubes displayed down the page. The 5-D case (HHP, HP, P, R, C) has the 4-D groupings displayed left to right across the page.





III. RECURSIVE (N-D) ARRAY CONSTRUCTION ALGORITHM) [ASSUMING THE FORTRAN SHAPE LIST CONVENTION].

Both conventions, APL & FORTRAN, can construct either left-to-right or right-to left in the shape list. This one is left-FORTRAN to-right: (m , n, k).

A. *Start With a {(0)-D} Array of Length 1*

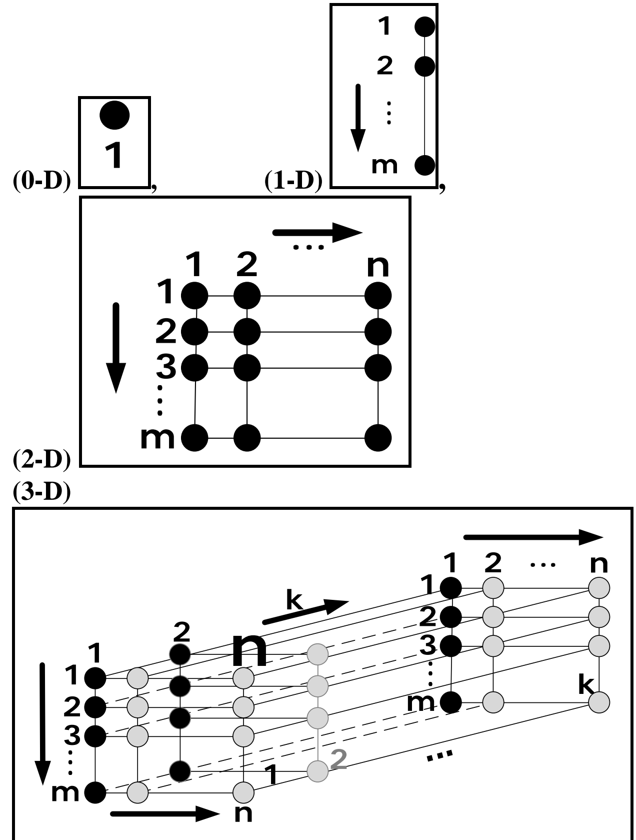
B. *Copy it n_1 Times, Along A New Dimension*

C. *Copy the Previous Result n_{i+1} Times Along A New Dimension*

D. *Repeat C Until the Last Dimension (n_N) is full.*

You start with a 0-D cell and do B to make a 1-D array of length n_1. C repeats B (N-1) more times using the previous result of from C. This puts the last dimension on the right end of the shape list.

E. *Example: (m, n, k) left-to-right.*



IV. RECURSIVE (N-D) ARRAY CONSTRUCTION ALGORITHM) [ASSUMING THE FORTRAN SHAPE LIST CONVENTION].

Both conventions, APL & FORTRAN, can construct either left-to-right or right-to left in the shape list. This one is FORTRAN right-to-left: (m, n, k).

A. *Start With a {(0)-D} Array of Length 1*

B. *Copy it n_N Times, Along A New Dimension*

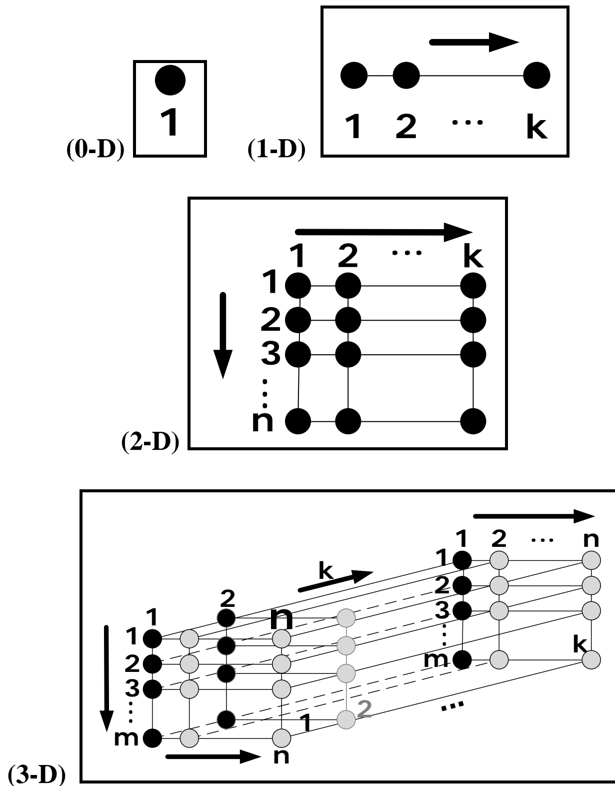
C. *Copy the Previous Result n_{i-1} Times Along A New Dimension*

D. *Repeat C Until the Last Dimension (n_1) is full.*

You start with a 0-D cell and do B to make a 1-D array of length n_N. C repeats B (N-1) more times using the

previous result of from C. This puts the last dimension on the left end of the shape list.

E. Example: (m, n, k) right-to-left.



Either way we end up with the same array.

V. RECURSIVE (N-D) ARRAY CONSTRUCTION ALGORITHM) [ASSUMING THE APL SHAPE LIST CONVENTION].

Both conventions, APL & FORTRAN, can construct either left-to-right or right-to left in the shape list. This one is APL left-to-right: (k, m, n).

A. Start With a {(0)-D} Array of Length 1

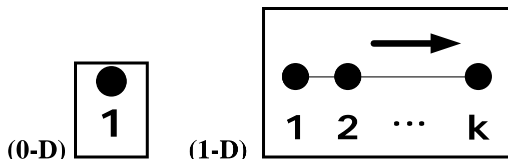
B. Copy it n_1 Times, Along A New Dimension

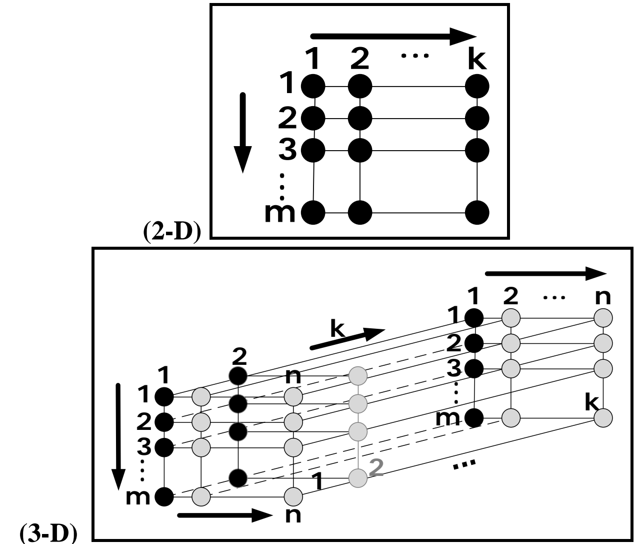
C. Copy the Previous Result n_{i+1} Times Along A New Dimension

D. Repeat C Until the Last Dimension (n_1) is full.

You start with a 0-D cell and do B to make a 1-D array of length n_1. C repeats B (N-1) more times using the previous result of from C. This puts the last dimension on the right end of the shape list.

E. Example: (k, m, n)





VI. RECURSIVE (N-D) ARRAY CONSTRUCTION ALGORITHM) [ASSUMING THE APL SHAPE LIST CONVENTION].

Both conventions, APL & FORTRAN, can construct either left-to-right or right-to left in the shape list. This one is APL right-to-left: (k, m, n).

A. Start With a {(0)-D} Array of Length 1

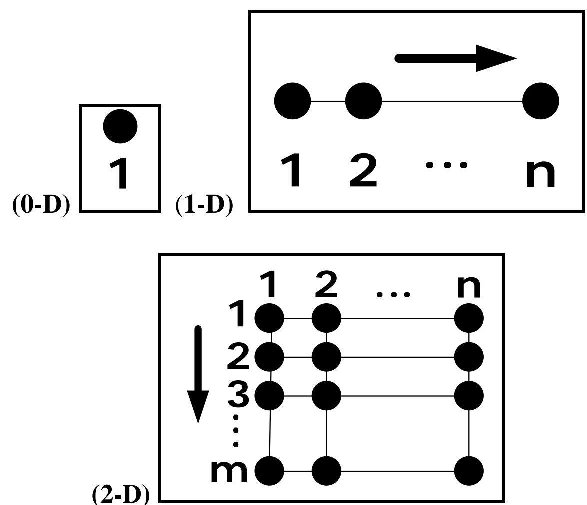
B. Copy it n_N Times, Along A New Dimension

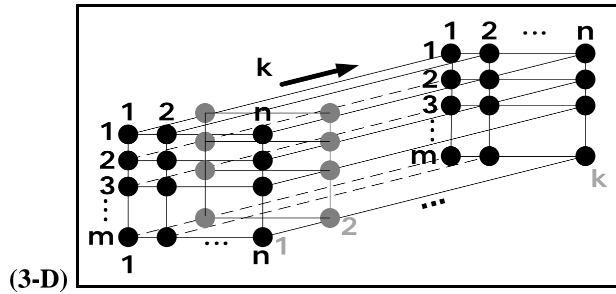
C. Copy the Previous Result n_{i-1} Times Along A New Dimension

D. Repeat C Until the Last Dimension (n_1) is full.

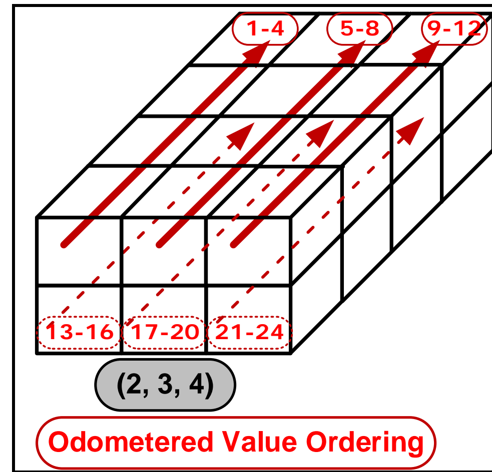
You start with a 0-D cell and do B to make a 1-D array of length n_N. C repeats B (N-1) more times using the previous result of from C. This puts the last dimension on the left end of the shape list.

E. Example: (k, m, n)





(3-D)



We see that we always end up with the same array regardless if we use a FORTRAN shape list or an APL shape list, and regardless of the left/right order of creation. We get the same cell structure.

The difference lies in the indexing. The index order of the cells differs by odometering.

Logically, we expect the tree to start (level 1) with the left most structure (n_1) and end with the rightmost (n_N) regardless of the APL/FORTRAN ordering. This is because that is the way odometering works and we would like the leaves of the tree to match the odometering order.

VII. A TREE CONSTRUCTION ALGORITHM USING RECURSIVE ARRAY CREATION [FORTRAN] LEFT-TO-RIGHT.

A. *The Tree Root (Level 0) Is the Entire N-D Array (Shape List)*

$(n_1, n_2, \ldots, n_{N-1}, n_N)$.

B. *(Level 1) is the n_1, {(N-1)-D)} Arrays Defined by Enumerating {(N-1)-D) Elements One-at-a-time.*

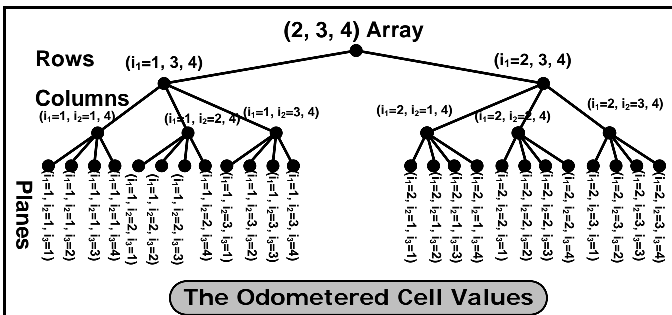
$(i_1, n_2, \ldots, n_{N-1}, n_N)$.

C. (Level *k {k ≤ N}) is the n_k, {(N-k)-D)} Arrays Defined by Enumerating {(N-k)-D) Elements One-at-a-time for EACH of the nodes defined at the previous level.*

$(i_1, i_2, \ldots, i_k, \ldots, n_{N-1}, n_N)$.

The nodes at level N are the (0-D) cells of the array.

D. *Example: FORTRAN Order (R, C, P) (2, 3, 4) Left-to-right.*



The cells are in odometer order.

VIII. A TREE CONSTRUCTION ALGORITHM USING THE RECURSIVE ARRAY CREATION [FORTRAN] RIGHT-TO-LEFT.

A. *The Tree Root (Level 0) Is the Entire N-D Array (Shape List)*

$(n_1, n_2, \ldots, n_{N-1}, n_N)$.

B. *(Level 1) is the n_N, {(N-1)-D)} Arrays Defined by Enumerating {(N-1)-D) Elements One-at-a-time.*

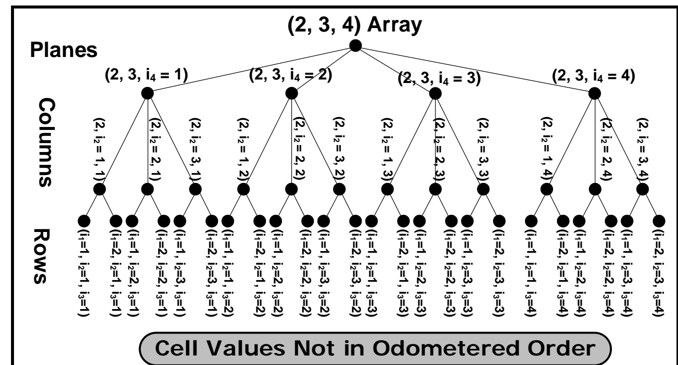
$(i_1, n_2, \ldots, n_{N-1}, n_N)$.

C. (Level *k {k ≤ N}) is the n_k, {(N-k)-D)} Arrays Defined by Enumerating {(N-k)-D) Elements One-at-a-time for EACH of the nodes defined at the previous level.*

$(i_1, i_2, \ldots, i_k, \ldots, n_{N-1}, n_N)$.

The nodes at level N are the (0-D) cells of the array.

D. *Example: FORTRAN Order (R, C, P) (2, 3, 4) Right-to-left.*

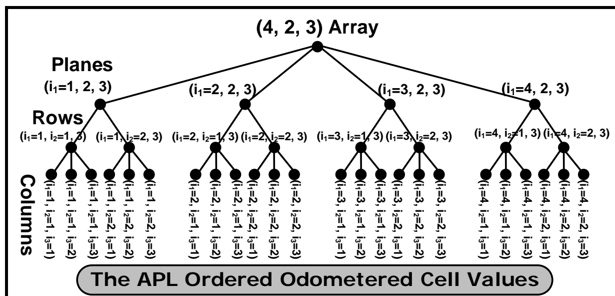


We see that we have to create the tree from a FORTRAN shape list using a FORTRAN ordering (left-to-right).

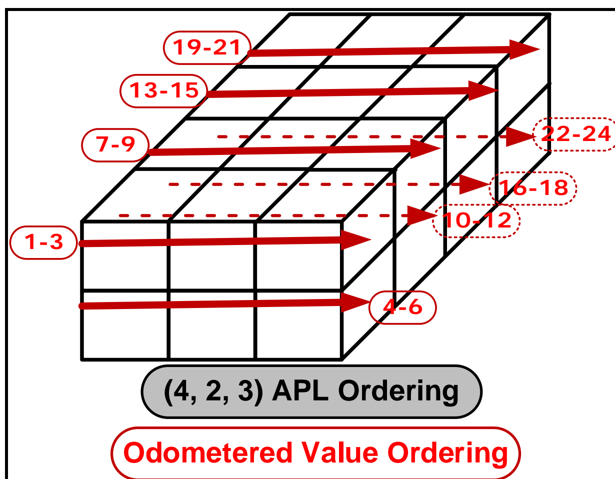
IX. A TREE CONSTRUCTION ALGORITHM USING THE RECURSIVE ARRAY CREATION [APL] LEFT-TO-RIGHT.

The algorithm is the same as in VII above.

A. *Example: APL Order (P, R, C) (4, 2, 3) Left-to-right.*



The cells are in odometer order.

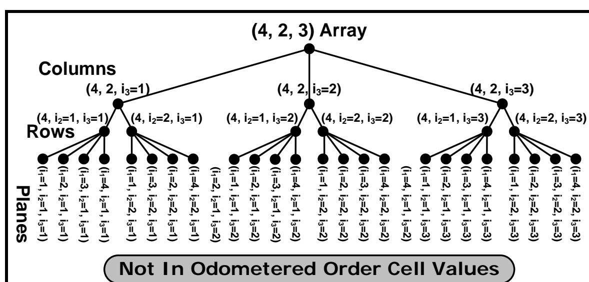


In both cases (2, 3, 4) -FORTRAN and (4, 2, 3)-APL, left-to-right tree construction led to leaves in the correct odometered order. However, notice that the two odometered orderings are *DIFFERENT*.

X. A TREE CONSTRUCTION ALGORITHM USING THE RECURSIVE ARRAY CREATION [APL] RIGHT-TO-LEFT.

The algorithm is the same as in VIII above.

A. *Example: APL Order (P, R, C) (4, 2, 3) Right-to-Left.*



The cells are NOT in odometer order.

B. *Discussion of the Examples: Non-uniqueness.*

The tree of an array is not unique if we don't care about the order of the cells.

C. *The Tree Construction Algorithm Reverses the N-D Array Construction Algorithm.*

The tree construction algorithm starts with the final output of the array construction algorithm – the array. It then moves to the penultimate construction step for the first level of the tree, and proceeds to use the results of the construction algorithm, but in reverse.

D. ***Tree Diagrams must be created from the shape list left to right to maintain odometer ordering in the tree leaves***.

XI. DISCUSSION.

A. *Trees are Sometimes Implemented Using Arrays*

This does not imply that the array is a tree. The mappings are totally different.

REFERENCES

[1] FORTRAN Array Shape List Format, https://docs.oracle.com/cd/E19957-01/805-4940/z400091044d0/index.html Last accessed 4/5/17.

[2] APL Array Shape List Format, https://en.wikipedia.org/wiki/APL_syntax_and_symbols Last accessed 4/5/17.

Dr. Ronald I. Frank