APL THINKING: EXAMPLES

Murray Eisenberg
Mathematics & Statistics Dept.
University of Massachusetts
Amherst, MA 01003 USA
(413) 545 - 2859

Howard A. Peelle
School of Education
University of Massachusetts
Amherst, MA 01003 USA
(413) 545 - 0135

Abstract

In an effort to understand "APL thinking", we examine a few selected examples of using APL to solve specific problems, namely: compute the median of a numerical vector; simulate the Replicate function; string search; carry forward work-to-be-done in excess of capacity; rotate concentric rectangular rings in a matrix; find column indices of pivots in an echelon matrix.

These examples are drawn from our teaching experience as well as from APL literature. We are particularly interested in studying thinking processes underlying alternative solutions to such problems -- i.e., our goal is to "get inside the head" of the APL programmer. Analyses include reconstructing thoughts, comparing alternative approaches, and, in general, scrutinizing supposed characteristics of APL thinking.

Introduction

Is there such a thing as "APL thinking"? If so, what is it? When a programmer claims to solve a problem in an 'APL way', we must ask what is meant by that. And, when someone implies that the APL way is better, or that APL is a better language for thinking about problems, we must ask how APL helps (or hinders) problem-solving. Specifically, just what is unique or special or different about thinking with APL? Such questions motivate us to study how APL affects cognitive processes. This should enable us to teach APL more effectively and to promulgate its advantages.

In an effort to understand "APL thinking", we began by focussing on views within the APL community: we have scanned the APL literature for definitions of or references to APL thinking; surveyed opinions on APL thinking (in APL Quote-Quad, Dec. 1985, and by questionnaire at APL86); interviewed students, instructors, and professional programmers; and analyzed APL learning bugs and APL teaching bugs [1,2]. Additionally, we have gathered opinions in response to our talks on the subject at the NY/SIGAPL "APL as a Tool of Thought" conference (April 1986) and at the New England APL User's Group meeting (May 1986) and have conducted a panel discussion at APL86.

We are still at an early stage of studying APL thinking. So far, we have found that there is a wide variety of interpretations extant. To some people, "APL thinking" suggests one-liners (expressions or defined functions). To others it means array (parallel) processing, thereby avoiding branching, iteration, and recursion; combining cases; working with large chunks; or doing data-driven computing. Some believe it lies in the notation -- syntax and the symbols themselves -- or in concise APL expressions, or in rich and powerful primitives. Some relate it to modularizing code (epitomized by direct definition), or a "glass box" program, or the propensity to generalize. A few attest that it involves use of identities and proofs in formal reasoning. It might also suggest tricky, contorted programming techniques. Further, it might even involve imagery, metaphors, and visualization of mental representations and transformations. Perhaps these are all aspects of APL thinking.

We have also recognized a number of related issues and challenges, including: the matter of style, (e.g., use of idioms); how to arrange studies (which seem to be low priority in business and academia); different APLs (e.g., APL2); distinguishing between problem-solving and formula translation; identifying problems which make APL look good or bad; finding reliable methods to get inside someone's head and reveal his or her thoughts; considering what the programmer knows beforehand; how to analyze protocols effectively; and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1987 ACM 0-89791-226-8/87/0005/0433 75¢

whether APL thinking differs among cultures. Nevertheless, we are perservering to focus on underlying cognitive processes.

Accordingly, here we are studying solutions to selected problems which purportedly involve APL thinking. Rather than our supposing what APL thinking is in general, these examples will have to speak for themselves. The examples are drawn from our experience teaching as well as from APL literature; they are necessarily small in scope but are intended to be provocative. We are deliberately exposing thinking processes -- au naturel.

Examples

1. The Median Problem

The problem is to write an APL expression or define a function to compute the median of a vector of numbers. The median is, by definition, the middle value of the vector when arranged in ascending (or descending) order; if there is an even number of numbers, however, the median is the average of the two middle values.

At the onset, we note that this is not really a "problem" in the strict sense, since the algorithm is included in the problem statement itself. Nonetheless, in the process of creating alternative expressions in APL, there are opportunities for varied ways of thinking.

A straightforward, case-by-case approach directly translates the problem statement into APL code:

```
     ∇ Z←MEDIAN X;N
[1]    Z←X[⍋X]
[2]    N←0.5×⍴X
[3]    →(0=2|⍴X)/EVEN
[4]   ODD: Z←Z[⌈N]
[5]    →0
[6]   EVEN: Z←0.5×Z[N]+Z[N+1]
     ∇
```

By contrast, the following "solution", which has become an old chestnut in the APL community, is not so obvious:

```
0.5×+/X[(⍋X)[⌊⌈¯0.5 0.5×1+⍴X]]
```

(from [3], p. 328)

APL novices often consider this as an example of "bizarre" thinking. They tend to make remarks such as "I wouldn't have thought of doing it that way", or even, "I'll never be able to think like that!"

The question is: What thinking was involved in doing it this way? (Would the creator please identify himself or herself?) In the absence of any known explanation of how the above expression was originally conceived, we can only speculate that it was based on a deliberate attempt to combine two cases into a single expression -- "the APL way" -- which entailed forming arithmetically a vector index of two middle values and, for the sake of efficiency, indexing the permutation vector directly.

Incidentally, as we compare these solutions, we should note that we are not considering which expression best conveys the concept of median nor which is most efficient in space and time.

Other variations, perhaps designed for greater clarity, include the following.

One which breaks the code into steps:

```
     ∇ Z←MEDIAN X;N
[1]    X←X[⍋X]
[2]    N←0.5×1+⍴X
[3]    Z←0.5×X[⌊N]+X[⌈N]
     ∇
```

(from [4], p.125)

One which uses subfunctions SORT, MEAN, and ROUND:

```
     ∇ MIDDLE←MEDIAN DATA
[1]    DATA←SORT DATA
[2]    MIDDLE←MEAN DATA[ROUND (0 1+⍴DATA)÷2]
     ∇
     ∇ UP←SORT DATA
[1]    UP←DATA[⍋DATA]
     ∇
     ∇ CENTER←MEAN DATA
[1]    CENTER←(+/DATA)÷⍴DATA
     ∇
     ∇ OFF←ROUND N
[1]    OFF←⌊N+0.5
     ∇
```

(from [5], p.274)

Both amalgamate the even and odd cases by duplicating the middle value in the odd case and then averaging the two values.

A different approach isolates the middle value(s) by dropping the unneeded front and back of the sorted vector:

```
     ∇ ANS←MEDIAN DATA;A
[1]    DATA←DATA[⍋DATA]
[2]    A←⌊¯0.5+(⍴DATA)÷2
[3]    DATA←A↓(-A)↓DATA
[4]    ANS←(+/DATA)÷(⍴DATA)
     ∇
```

(by a student)

Yet another approach involves reproducing the entire vector:

```
MEAN (SORT DATA,DATA)[0 1+⍴DATA]
```

This was offered spontaneously by a graduate student, who explained that it was an example of "APL thinking" because it solved the problem in an unusual way --

that is, it went outside the scope of thinking one would normally do. He also realized that his solution was "excessive", that is, inefficient in space and indulging in powerful primitive functions. We ask: To what extent does 'the unusual' and 'the excessive' typify APL thinking?

A similar approach is embodied in the following program:

```
      ∇ MIDDLE←MEDIAN DATA
[1]     DATA←SORT DATA
[2]     MIDDLE←
        MEAN ,(DATA,[0.5]⌽DATA)[⌈(⍴DATA)÷2]
      ∇
```

([5], p. 275)

This way of collecting two cases into one might also seem excessive and tricky in that it creates a higher-dimensional array. When is it worth trying to think of this kind of solution? And why?

These latter two approaches set up the data so an APL primitive function -- here Index -- can do the key job directly. Is this the essence of "APL thinking"?

The question still remains: What did the people who originated these solutions think of when they were devising the algorithms?

2. The Replicate Problem

Most implementations of APL include the Replicate extension of the standard Compress primitive function, as in:

```
      2 7 0 1 4 / 'ABCDE'
AABBBBBBBDEEEE
```

A few implementations do not include Replicate; even the ISO APL Standard does not include it; in any case, simulating it is an interesting programming problem.

Here's one approach to simulating Replicate for vector arguments: Let V be the vector of elements to be replicated (the right argument) and let N be the vector of corresponding numbers of copies (the left argument). First, remove from V elements with 0s in N (and notice that this takes care of scalar arguments as well):

```
      N←(B←N>0)/N
      V←B/V
```

The number of elements in the desired result is +/N. So, form a vector B of that length:

```
      B←(+/N)⍴0
```

The strategy here is to make B a vector of indices into V needed to produce the result. In the example above, these would be 1 1 2 2 2 2 2 2 2 3 4 4 4 4 (recall that V was compressed to four elements earlier). Now mark where each block of repeated indices should begin:

```
      B[1,1+¯1+⍳+\N]←1
```

In the example, this produces the vector 1 0 1 0 0 0 0 0 0 1 1 0 0 0. Actually, in order to protect against having nothing left to select, modify the above to:

```
      B[(⍴N)↑1,1+¯1+⍳+\N]←1
```

Finally, Plus-Scan B to produce indices into V, as in the defined function below:

```
      ∇ Z←N REPLICATE V;B;I
[1]     N←(B←N>0)/N
[2]     V←B/V
[3]     B←(¯1↑I←⍳+\N)⍴0
[4]     B[(⍴I)↑1,1+¯1↓I]←1
[5]     Z←V[⍳+\B]
      ∇
```

(Of course, a more direct definition is: `Z←V[+\¯1⌽(⍳+/N)∈⍳+\N]` for `V←(N>0)/V` .)

This is characteristic of a common APL programming technique: use a bit pattern to mark where blocks start, then produce the required indices.

Another approach is illustrated in the following function, which is based on Jill Wade's solution in [3], p. 124:

```
      ∇ Z←N REPLICATE V
[1]     V←,V
[2]     N←,N
[3]     Z←V[(,N∘.≥⍳⌈/N)/,(⍳⍴N)∘.×(⌈/N)⍴1]
      ∇
```

Looking again at the example, the indices we want could be represented in a "ragged" array (which begs for enhanced APL):

```
1 1
2 2 2 2 2 2 2
                        ⍝ NO INDICES FOR V[3]
4
5 5 5 5
```

To obtain these indices, form a matrix of the possible indices repeated in columns,

```
      (⍳⍴N)∘.×(⌈/N)⍴1
1 1 1 1 1 1 1
2 2 2 2 2 2 2
3 3 3 3 3 3 3
4 4 4 4 4 4 4
5 5 5 5 5 5 5
```

a horizontal bar chart indicating how many copies of each index,

```
      N∘.≥⍳⌈/N
1 1 0 0 0 0 0
1 1 1 1 1 1 1
0 0 0 0 0 0 0
1 0 0 0 0 0 0
1 1 1 1 0 0 0
```

and, finally, Ravel those two matrices and Compress.

Already, this solution is noteworthy in the way it goes outside the domain of the original problem -- that is, it involves forming two-dimensional arrays in order to solve a one-dimensional array problem.

Wade's actual solution (see [3]) is general in that it replicates an array of arbitrary rank along any dimension. It is intriguing to know whether her solution was derived from a version for rank-one, such as the above, or was developed in full generality from the start.

3. The String Search Problem

The problem is to locate all occurrences of a substring in a given string.

A first attempt at an array-oriented solution embodies in a matrix representation the straightforward, essentially iterative approach (sliding the substring along one character at a time and testing for matches):

```
     ∇ R←T STRINGSEARCH W
[1]    R←(∧⌿(¯1+⍳⍴W)⌽W∘.=T)/⍳⍴T
     ∇
```

(from [3] p. 205)

But, alas, this expression gives a false match for a special end condition, e.g.:

```
     'LOOK FOR APL WRAP' STRINGSEARCH 'APL'
10 16
```

Use of the Rotate primitive ought to cue a potential problem with wrap-around. One might have expected that APL would have helped in handling the special case here -- as it often does generalize automatically to empty arrays, scalars or 1-element arrays. At least, one might have hoped that APL would have provided a better way to integrate the special case than an ad hoc patch:

```
     ((1-⍴W)↓∧⌿(¯1+⍳⍴W)⌽W∘.=T)/⍳1+(⍴T)-⍴W
```

Or, alternatively:

```
     (¯1↓∧⌿(¯1+⍳⍴W)⌽(W∘.=T),0)/⍳⍴T
```

(from [6], p. 19)

Another approach incorporates a preliminary search:

```
     I←⍳(I=1↑W)/⍳⍴I←(1-⍴W)↓T
     (T[I∘.+¯1+⍳⍴W]∧.=W)/I
```

(from [7], p. 7)

Another handles scalar and null arguments:

```
     ∇ POSITIONS←STRING MATCH SUBSTRING
[1]    SUBSTRING←,SUBSTRING
[2]    POSITIONS←∧/[1]
       (¯1+⍳⍴SUBSTRING)⌽SUBSTRING∘.=,STRING
[3]    POSITIONS←
       (1-⍴SUBSTRING)↓POSITIONS∧0<⍴STRING
[4]    POSITIONS←POSITIONS/⍳⍴POSITIONS
     ∇
```

([8], p.146)

Concern for efficiency might motivate the following revision:

```
     ∇ P←STRING MATCH SUB;LSUB;LAST
[1]    LAST←0⌈(⍴,STRING)+1-LSUB←⍴,SUB
[2]    P←((LSUB>0)∧((LAST↑STRING)=1↑SUB)∧
         ((-LAST)↑STRING)=¯1↑SUB)/⍳LAST
[3]    P←((,STRING)[P∘.+(⍳0⌈LSUB-2)]∧.=
                              1↓¯1↑SUB)/P
     ∇
```

([8], p.156)

Now the question is: how close is this to the way one would (like to) think about this problem?

4. The Work Problem

This was posed as a contest problem in [9]. You are given a vector of work needing to be done (DEMAND) in successive time periods together with a second vector of capacity (SUPPLY) to perform that work in those time periods. Work not performed in one time period is carried forward to the next one; capacity not used, however, is lost forever. The problem is to compute the work done in all successive periods.

A looping solution is straight-forward:

```
     ∇ RESULT←SUPPLY WORK DEMAND;CARRY;I;
                          PERIODS;NEWDEMAND
[1]    RESULT←(PERIODS←⍴SUPPLY)⍴CARRY←I←0
[2]   LOOP: →(PERIODS<I←I+1)/0
[3]    RESULT[I]←SUPPLY[I]⌊NEWDEMAND←
                          DEMAND[I]+CARRY
[4]    CARRY←0⌈NEWDEMAND-RESULT[I]
[5]    →LOOP
     ∇
```

The crux is that the work done at any time period is the smaller of supply and demand plus the unsatisfied demand from before.

There is an obvious recursive version of this solution:

```
     ∇ RESULT←SUPPLY WORK DEMAND
[1]    RESULT←⍳0
[2]    →(0∈⍴SUPPLY)/0
[3]    RESULT←(1↑DEMAND),1↑SUPPLY
[4]    RESULT←(⌊/RESULT),(1↓SUPPLY) WORK
         (1↓DEMAND)+(¯1+⍴DEMAND)↑0⌈-/RESULT
     ∇
```

Another recursive approach incorporates some parallel processing:

```
     ∇ RESULT←SUPPLY WORK DEMAND;CARRY
[1]    RESULT←SUPPLY⌊DEMAND
[2]    →(∧/0=CARRY←SHIFT 0⌈DEMAND-SUPPLY)/0
[3]    RESULT←SUPPLY WORK RESULT+CARRY
     ∇
     ∇ R←SHIFT V
[1]    R←0,¯1↓V
     ∇
```

The published statement of the problem gave the following recurrence relation:

$$RESULT[I] \leftarrow SUPPLY[I] \lfloor (+/I \uparrow DEMAND) - +/(I-1) \uparrow RESULT$$

But this is a different (and more complicated) way of stating the problem. It uses "complete recursion", expressing the RESULT at any time I in terms of <u>all</u> previous times.

Instead, let us start with one-step recurrence relations in order to obtain an "array-oriented" (non-looping) solution:

$$R[I] \leftarrow S[I] \lfloor D[I] + C[I-1]$$
$$C[I] \leftarrow 0 \lceil D[I] + C[I-1] - R[I]$$

(where we have abbreviated DEMAND, CARRY, SUPPLY, and RESULT, respectively, by D, C, S, and R).

The thought is to eliminate C from this pair of relations, thereby to obtain a formula for R in terms of S and D alone.

The identities

$$X+(Y \lceil Z) \leftrightarrow (X+Y) \lceil (X+Z)$$
$$X-(Y \lfloor Z) \leftrightarrow (X-Y) \lceil (X-Z)$$

lead to:

$$\begin{aligned} C[I] &\leftrightarrow 0 \lceil (D[I]+C[I-1]) - \\ &\qquad (S[I] \lfloor (D[I]+C[I-1])) \\ &\leftrightarrow 0 \lceil (D[I]+C[I-1]-S[I]) \lceil \\ &\qquad (D[I]+C[I-1]-(D[I]+C[I-1])) \\ &\leftrightarrow 0 \lceil ((D[I]+C[I-1]-S[I]) \lceil 0) \\ &\leftrightarrow 0 \lceil 0 \lceil D[I]+C[I-1]-S[I] \\ &\leftrightarrow 0 \lceil (D[I]-S[I]) + C[I-1] \end{aligned}$$

Abbreviating $D[I]-S[I]$ by $E[I]$ (E for excess), we get:

$$C[I] \leftrightarrow 0 \lceil E[I] + C[I-1]$$

Together with the initial condition $C[0] \leftrightarrow 0$ the above relation gives, in turn:

$$\begin{aligned} C[1] &\leftrightarrow 0 \lceil E[1]+C[0] \leftrightarrow 0 \lceil E[1] \\ C[2] &\leftrightarrow 0 \lceil E[2]+C[1] \leftrightarrow 0 \lceil (E[2] + 0 \lceil E[1]) \\ &\leftrightarrow 0 \lceil E[2] \lceil (E[1]+E[2]) \end{aligned}$$

Similarly:

$$\begin{aligned} C[3] &\leftrightarrow 0 \lceil E[3] \lceil (E[2]+E[3]) \lceil \\ &\qquad (E[1]+E[2]+E[3]) \end{aligned}$$

Now the pattern seems clear:

$$C[I] \leftrightarrow 0 \lceil \lceil /+\backslash \phi I \uparrow E$$

This simplifies slightly to the following (recall $E \leftrightarrow D-S$):

$$C[I] \leftrightarrow \lceil /0, +\backslash \phi I \uparrow D-S \qquad (*)$$

Together with

$$R[I] \leftrightarrow S[I] \lfloor D[I] + C[I-1] \qquad (**)$$

this provides a new solution -- one that still seems to be iterative. However, (**) can be vectorized as $R \leftrightarrow S \lfloor D+0, ¯1 \uparrow C$. So, to get a non-iterative solution, it remains to recast (*), that is, to calculate all the sums there at once. These are row sums of the triangular (non-APL) array:

```
E[1]
E[1]  E[2]
E[1]  E[2]  E[3]
E[1]  E[2]  E[3]  E[4]
...
```

Then the numbers to be summed can be obtained by multiplying by the appropriate bit-mask, generated by:

```
     ∇ M←LOWERΔ N
[1]    M←(⍳N)∘.≥⍳N
     ∇
```

Now C can be vectorized as:

$$C \leftrightarrow \lceil /0, +\backslash \phi (LOWER\Delta\ \rho S) \times (2 \rho \rho S) \rho D-S$$

At last, a complete non-iterative solution is at hand:

```
     ∇ RESULT←SUPPLY WORK DEMAND;CARRY;N
[1]    N←⍴SUPPLY←,SUPPLY
[2]    CARRY←⌈/0,+\φ(LOWERΔ N)×
                  (2⍴N)⍴DEMAND-SUPPLY
[3]    RESULT←SUPPLY⌊DEMAND+0,¯1↓CARRY
     ∇
```

Notice the critical role of identities and formal mathematical manipulations in arriving at this solution.

All the preceding solutions have in common the expression of work done as the smaller of supply and total demand (current plus carried over). By way of contrast, the winning contest entry, submitted by David Hosier and published in [10] (along with a formal proof of correctness) is:

$$SUPPLY-(1 \uparrow T)-¯1 \uparrow T \leftarrow \lceil \backslash 0, +\backslash SUPPLY-DEMAND$$

Hosier's solution seems to be based on a different idea: the work done is obtained by reducing the available supply by its unused portion. It is not clear what T represents conceptually, nor what suggested using Max-Scan.

5. The Matrix Rotation Problem

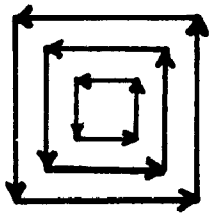
This problem was offered as a prize competition at APL86. Consider a matrix as comprised of concentric rectangular rings. The problem is to write a dyadic function (MATROT) which rotates each ring a given number of positions about a central axis perpendicular to the plane of the matrix. A positive left argument causes counter-clockwise rotation. For example:

```
      3 MATROT 4 4⍴⍳16
 4  8 12 16
 3 10  6 15
 2 11  7 14
 1  5  9 13
```

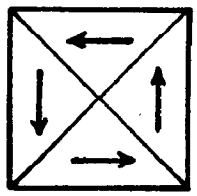
Here we offer a post-hoc account of one person's attempt at developing a solution -- albeit incomplete and revealing various shortcomings in thinking.

We begin with some overall thoughts about alternative approaches: "First, I considered doing it recursively, using a mask over the inner rings and joining together the pieces.... But I preferred to try doing it directly -- that is, all at once with array processing -- since that seemed to be the spirit of the competition. Further, I realized that I could do it with index mapping, but it would be too tedious (and inelegant). I also briefly considered the generic approach of creating a larger rank array, doing some transformation on it, and then selecting the appropriate part(s), perhaps using Dyadic Transpose. But then I got the idea of treating the problem geometrically -- based on how I visualized the matrix rings actually rotating."

Now we follow this "geometric" approach. "The idea came from drawing a diagram showing the movement of vectors:



"This, in turn, suggested sliding triangular sections of the matrix (like faults in plate tectonics):"



(We note the mental slip into the convenient case of a square matrix.)

"Then I refined the diagram to make sure how to represent the elements discretely in an APL matrix. And I decided to use a logical mask to get each of the four triangular pieces, shift them (with Rotate, of course!), and then add them together." (Note the slip into numerical matrices only.)

"Incidentally, I knew I was diagramming a matrix of odd shape but expected that I could make the function work for the even case also.

"Then I wrote APL code:

```
N←1↑⍴M
T←((⍳N)∘.≤(⍳N))∧(⌽(⍳N)∘.≤(⍳N))
(1⌽M×T)+(¯1⊖M×⌽⌽T)+(¯1⌽M×⌽⌽⌽⌽T)+(1⊖M×⌽⌽T)
```

"As I was writing this down, I was thinking of slight variations to economize on code (and perhaps speed). I chose to rewrite it in direct definition form:

```
ROT: (1⊖ω×⌽⌽T)+(¯1⌽ω×T←⌽⌽T)+(¯1⊖ω×T←⌽⌽T)+
                1⌽ω×T←(N∘.<N)∧⌽N∘.≤N←⍳1↑⍴ω
```

"While I was doing this, I realized that the function only causes a rotation of one position and that I would have to recurse after all. So, I quickly built a supra-function:"

```
MATROT: (⍺-1) MATROT ROT ⍵ : ⍺=0 : ⍵
```

Next we will see an excursion into enhanced APL: "Then I tried a condensed expression using Rationalized APL ([11]) for one-position rotation:

```
ROT: +⌿ω×⍤2 (⌽⍤⌽)⌷⍳4 T
```

And I even sketched an expression using the Dual operator, wishing that there were a function F (and its inverse) for splitting rings into vectors:"

```
ROT: α⌽⍨Fω
```

F:

Now, back to earth. "Agggh. After a short pause and a glance back at the original problem statement, I realized that my MATROT function didn't extend to rectangular matrices. And, I really wanted to do positive or negative rotation as well."

"Later, I got a chance to test it on my computer. In trying some examples, I soon saw that -- uh oh -- I had forgotten the element in the center of an odd-shaped matrix. (The even case was OK.) So, I patched it in (without ruining the even case), along with some small improvements:

```
     ∇ R←N MATROT M
[1]    R←M
[2]    →(N≤0)/0
[3]    R←R∧⌽⍉~R←R∘.≤R←⍳1↑⍴M
[4]    R←(N-1) MATROT (M×R∧⍉R←ID 1↑⍴M)+
       (1⍉M×⌽⍉R)+(¯1⊖M×⊖⍉R)+(¯1⍉M×⍉⌽R)+1⊖M×R
     ∇
     ∇ R←ID N
[1]    R←(⍳N)∘.=⍳N
     ∇
```

"I couldn't help but wonder if there were a better way to include the center in the original mask.... Finally, I began to reconsider the whole approach, because I needed to deal with rectangular matrices anyway. Maybe even try to generalize to arrays? But I had no more time available."

The reader may wish to compare this approach -- which involves some spatial transformations -- with the winning solution (see [12]) which generates a spiral of indices.

6. The Echelon Pivots Problem

This last problem is a "quickie", one solution to which involves uncomplicated use of only a few functions. You are given an "echelon" matrix, such as:

```
      M
4 1 7 4 9 0 1 3
0 5 8 6 2 7 2 5
0 0 0 3 0 4 0 4
0 0 0 0 0 0 6 5
0 0 0 0 0 0 0 0
```

The first nonzero element -- the so-called "pivot" -- in each row of M is located to the right of the pivot in the row above it; and any zero rows come at the bottom. (Such matrices are produced in the course of solving systems of linear equations or testing column vectors for linear independence.) The problem is to locate the columns containing these pivots.

To solve it, first see which elements in M are nonzero:

```
      M≠0
1 1 1 1 1 0 1 1
0 1 1 1 1 1 1 1
0 0 0 1 0 1 0 1
0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0
```

Turn off all bits after the first 1 in each row (of course, it helps to know this idiom):

```
      <\M≠0
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
```

Mark the desired columns,

```
      ∨⌿<\M≠0
1 1 0 1 0 0 1 0
```

and, finally, select their indices:

```
      (∨⌿<\M≠0)/⍳¯1↑⍴M
1 2 4 7
```

Another Scan idiom might come to mind initially, but it leads to a bug for rows of all 0s:

```
      1++⌿∧\M=0
```

The question this example raises is: to what extent does "APL thinking" consist of being familiar with very powerful primitives (including their idiomatic combinations) and merely selecting them when needed? That is, does a lot of APL thinking really consist of not having to think very much at all?

Conclusion

The examples here have illustrated a variety of ways of solving problems with APL, such as:

- combining cases
- creating "excessive" arrays (beyond the domain of the problem)
- setting up arrays in order to apply appropriate primitives or idioms
- coercing ragged arrays into rectangular arrays
- forming bit masks
- considering edge conditions and special cases
- seeking efficiency
- using examples as guides
- transforming spatial representations

In several of the example problems we were able to capture to some degree the actual thinking-in-progress toward solutions (however imperfect). In others, we had no alternative but to speculate, that is, invent plausible scenarios for how solvers were led to their solutions.

By contrast, unfortunately, much of the literature on APL programming consists of "textbook" examples: typically, the line of reasoning leading to the final result is presented as linear, polished, and pristine. Rarely do we see a detailed account of how a mortal problem-solver really thinks, replete with all paths taken -- including blind alleys -- and various errors encountered.

Doubtless, many APL programmers are too busy going about their work to take time to describe how they do it. Even if they have time, they may be unwilling to air their "dirty laundry" and are unaccustomed to thinking about their thinking -- or at least unpracticed in recording their thoughts.

If we, in the APL community, believe that APL is a good (or better) tool for solving problems -- and especially if we proselytize about this -- then it is incumbent upon us to understand why. Indeed, we need to be able to articulate just what is "APL thinking".

References

[1] "APL Learning Bugs", M. Eisenberg and H.A. Peelle, APL83 Conference Proc., APL Quote-Quad, Vol. 13, No. 3, March, 1983

[2] "APL Teaching Bugs", H.A. Peelle and M. Eisenberg, APL85 Conference Proc., APL Quote-Quad, Vol. 15, No. 4, May, 1985

[3] APL: An Interactive Approach (3rd Ed.) L. Gilman and A.J. Rose, Wiley, New York, 1984

[4] APL: The Language and Its Usage, R.P. Polivka and S. Pakin, Prentice-Hall, Englewood Cliffs, NJ, 1975

[5] APL: An Introduction, H.A. Peelle, Holt, Rinehart and Winston, New York, 1986

[6] "The APL Idiom List", A.J. Perlis and S. Rugaber, Yale University, Dept. of Computer Science, Research Report #87, April, 1977

[7] FinnAPL Idiom Library (2nd Ed.), Finnish APL Association, Helsinki, July, 1982

[8] Learning APL: An Array Processing Language, J.A. Mason, Harper & Row, New York, 1986

[9] "Contest 13", I.P. Sharp Newsletter, Vol. 12, No. 4, (Technical Supplement) July/August, 1984, p. T3

[10] "Contest 13 Results", I.P.Sharp Newsl. Vol. 12, No. 6, (Technical Supplement) November/December, 1984, p. T2

[11] "Rationalized APL", K.E. Iverson, Sharp Research Report #1, I.P. Sharp Associates, Toronto, January, 1983

[12] "APL86 Competition: Matrix Rotation", Vector, Vol. 3, No. 2, October, 1986