APL '81: Proceedings of the international conference on APL
October 1981
Pages 281–285

INFINITE ARRAYS AND DIAGONALIZATION

J. O. Shallit
Department of Mathematics
University of California
Berkeley, Ca. 94720
(415) 642-5523

Abstract

This paper discusses the application of infinite arrays to several areas, notably the formation of "do-while" expressions. The functions <u>diagonal</u> and <u>inverse diagonal</u> are defined, with applications to processing both finite and infinite arrays. Infinite arrays are shown to be useful in mathematical exposition. Finally, suggestions are given for the implementation of diagonalization functions.

1. Introduction.

In a previous paper [1], E. E. McDonnell and the author briefly discussed the implications of arrays containing a countably infinite number of elements. The present paper examines some applications in greater detail.

Origin 0 is used throughout. Certain non-standard notation is employed, and the reader is urged to scan the appendix before proceeding. Direct definition is used throughout; for a program to process direct definitions, see [2].

As in [1] and [3], the symbol _ (underbar) is used to denote infinity. The expression ⍳_ denotes the infinite vector *Z* such that *Z*[*K*] ↔ *K*.

2. Replacing the "Do-While" Construct.

Many algorithms which when coded in languages such as PL/I or FORTRAN involve constructs like

```
do i = 0 to n-1
```

can be described in APL as functions on ⍳*N*. The ability to generate vectors of indices and to process arrays without explicitly providing dimensions allows single-line formulation of many problems.

Unfortunately, current implementations of APL do not provide simple ways to replace the construct often called a "do-while" loop. For example, consider the problem of determining the number of terms of the inverse factorial series

```
+/÷!⍳N
```

needed to get an approximation to e accurate to 1E¯5. In a PL/I-like language, this could be solved as follows:

```
Procedure count;

e = exp(1);
sum = 0;
i = 0;

do while 1E-5 < (e-sum);
     sum = sum + 1/fact(i);
     i = i+1;
     end;

return(i);
```

However, using infinite arrays, this program can be replaced by the following APL expression:

```
      1+(1E¯5<(*1)-+\÷!⍳_)⍳0
9
```

This gives the solution of 9 terms necessary to sum the series to the given accuracy. Further examples follow:

A. Find a numerator for a good rational approximation to pi:

```
      1+(∨⌿1E¯4>1|○1 ¯1∘.×1+⍳_)⍳1
113
      ○113
354.9999699
      (○1),355÷113
3.141592654 3.14159292
```

B. Show that not all numbers of the form

$$x^2 + x + 41$$

are prime (see, for example, [4]).

```
      PRIME: 2=+/0=(1+ιω)|ω
      ⍝ PRIME N ↔ 1 if N is prime,
                  0 otherwise

      T←( _ 1 ρι_)⊥1 1 41
      5↑T
41 43 47 53 61
      5↑PRIME¨T
1 1 1 1 1
      (PRIME¨T)ι0
40

      40⊥1 1 41
1681
      41*2
1681
```

Here, the symbol ¨ denotes the "itemwise" or "each" operator, which applies its functional left argument to each element of its array right argument. See [5] and the appendix.

C. Find the least prime greater than or equal to a given integer:

```
      LPGE: ω + (PRIME¨ω+ι_)ι1
      LPGE 10000
10007
```

In these examples, the fundamental concept is that we do not know, a priori, an upper bound on how many terms must be examined. Hence the infinite vector ι_ effectively permits computations to continue until an answer is found.

3. Diagonalization.

The diagonal and inverse diagonal transformations were introduced in [1]. For finite arrays, these functions are given by:

```
      DIAG: (,ω)[⍋,+⌿IOTA ρω]
     IDIAG: αρω[⍋⍋,+⌿IOTA α]

      IOTA: ω⊤ωρι×/ω
```

In terms of syntax and ranks of arguments and results, *DIAG* behaves just like ravel (monadic ,) and *IDIAG* behaves like reshape (dyadic ρ). The functions *DIAG* and *IDIAG* are inverses; we have

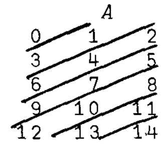
```
A ↔ (ρA) IDIAG DIAG A.
```

We propose use of the (currently unassigned) symbol ø for both of the diagonal transformations. Monadically, ø would behave like *DIAG*; dyadically, it would be *IDIAG*. Hence the above identity may be more elegantly expressed as

```
A ↔ (ρA)øøA.
```

This choice has the advantage of form following function, since the shape of the symbol suggests the transformation:

```
      A←5 3 ρ ι 15
      A
```



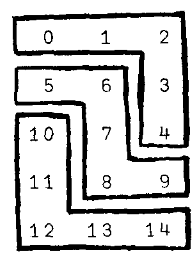

```
      øA
0 1 3 2 4 6 5 7 9 8 10 12 11 13 14
```

Using ø with finite arrays can produce unusual restructuring. For example, consider the expression

```
(⌽ρA)øøA.
```

```
      A←3 5 ρ ι 15
      A
 0   1   2   3   4
 5   6   7   8   9
10  11  12  13  14

      (⌽ρA)øøA
```



As another example of the use of ø, consider the function *DET*3 below which gives the determinant of a 3 x 3 matrix:

```
      DET3: (ALT ⌽ω) - ALT ω
       ALT: +/×/3 3ρ3↓øω,ω

      DET3 3 3ρι9
0
      DET3 (ι3)∘.≠ι3
2
```

The function ø also exhibits its utility in conjunction with infinite arrays. For example:

A. List all composite integers:

```
      ∪ø(2+ι_)∘.×2+ι_
4 6 8 9 10 12 15 16 14 18 20 21 24 25 ...
```

Here the symbol ∪ is Iverson's "nub" function, which selects distinct elements from its array right argument. See the Appendix and [3].

B. Let *PR* be an infinite vector of the prime numbers in ascending order. Compute a vector consisting of all integers that are the product of precisely 3 primes (counting multiplicities):

```
      ∪⍉PR∘.×PR∘.×PR
8 12 20 18 28 30 27 44 42 50 45 52 66 ...
```

C. Generate rows of Pascal's triangle:

```
      MS: α!α+ω-1
      ⍉(ι_)∘.MS 1+ι_
1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 ...
```

Here we are performing an outer product with respect to the user-defined function *MS*.

4. Infinite Arrays in Exposition.

In the previous three sections we have restricted operations using infinite arrays to those that could easily be implemented in the sense of [1]. Use of APL in exposition, however, is not subject to such constraints.

For example, we have the identities:

```
      *1 ←→ +/÷!ι_
```

and

```
      ○1 ←→ -/4÷1+2×ι_.
```

More examples follow:

A. Let *V* be an infinite vector. Then

```
      ⌊/V
```

is the greatest lower bound or "inf" of *V*; in the same fashion, ⌈/*V* is the least upper bound or "sup". See [6].

B. Prove that the infinite series

```
      +/÷1+ι_
```

diverges.

Proof: Note that

```
      1 ←→ (1+ι_)∧.≤(2*ι_)/2*1+ι_.
```

(Here we are using the symbol / to mean "replicate"; see the Appendix.)

Hence (+/÷1+ι_) ≥ +/÷(2*ι_)/2*1+ι_; and the expression on the right is seen to equal +/(2*ι_)÷2*1+ι_; this is just +/_ρ÷2 or _. Thus the infinite series diverges.

C. Let ⍂ denote a new operator which we will call "right-scan"; for vectors *V* we have

```
      (f⍂V)[K] ←→ f/K↓V
```

where f is any scalar dyadic function.

Then if *V* is an infinite vector, the expression

```
      ⌊/⌈⍂V
```

is the "lim sup" and, in a similar fashion, ⌈/⌊⍂*V* gives the "lim inf". See [6].

D. Define *J*←ι_. Then show that

```
      4 ←→ +/(J+1)÷2*J.
```

Proof:

```
      +/(J+1)÷2*J
      +/(J+1)/2*-J
      +/2*-(J+1)/J
      +/2*-⍉J∘.+J
      +/+/2*-J∘.+J
      +/2*1-J
      4.
```

Here we are using the proof style of Iverson [7] where equivalent statements are written below each other.

E. Prove that the positive rational numbers are countable.

Proof. The expression

```
      ∪⍉(1+ι_)∘.÷1+ι_
1 0.5 2 0.333 3 0.25 0.667 1.5 4 0.2 ...
```

exhibits a one-one correspondence between ι_ and the positive rationals.

F. Define a function *FACDIV* such that

```
      P FACDIV N
```

gives the number of times a given prime *P* divides !*N*. (See [8].)

Solution:

```
      FACDIV: +/⌊ω÷α*1+ι_

      5 FACDIV 10000
2499
```

G. Prove that the set S = $\{x:\ 0<x<1\}$ is uncountable.

Proof: (Cantor diagonalization). Assume S is countable. Then we can represent S by the infinite vector *S*, and there exists a matrix *M* of shape _ _ such that the *K*-th row of *M* is the base-10 representation of the *K*-th element of *S*, i. e.

```
S[K] ←→ M[K;]+.×10*-1+ι_.
```

Now consider the vector

```
D←9 - 1 1 ⍉ M.
```

Then *D* is the base-10 representation of a number between 0 and 1 and so

```
(D+.×10*-1+ι_) ∊ S;
```

but *D* cannot appear anywhere as a row of *M* since we have

```
D[K] ≠ M[K;K].
```

Hence our original assumption that the set S was countable must be false.

5. Implementation of ∅.

In the case where the right argument to ∅ is a finite array, implementation is provided by the functions *DIAG* and *IDIAG* given in section 3.

Implementation is much more difficult in the case of infinite arrays, however. The functions *DI* (diagonal index) and *IDI* (inverse diagonal index) below suggest one possible approach.

These functions are defined such that

```
(∅A)[ιK] ←→ A INDEX K DI ρA
```

and

```
(W∅V) INDEX U ←→ V[(((+/U) IDI W)∧.=U)ι1]
```

where *A* is an array, *U*, *V*, and *W* are vectors, *K* is a non-negative integer, and *INDEX* is Iverson's generalized indexing function given by

```
INDEX: (,α)[⍉(ρα)⊥⍉ω].
```

See [9].

```
    ∇ Z←K DI P;J
      Z←(0,ρP)ρJ←0
     L1:→(K≤ρZ)/L2
      Z←Z,[⎕IO]⊖J PART P
      J←J+1
      →L1
     L2:Z←(K,ρP)↑Z
    ∇

    ∇ Z←K IDI V
      Z←(0,ρV)ρ0
     L1:→(K<0)/L2
      Z←Z,[⎕IO] K PART V
      K←K-1
      →L1
     L2:Z←⊖Z
    ∇

    ∇ Z←K PART V;I;T;B;R
      ⍝ THE RESULT <Z> IS A MATRIX SUCH
      ⍝ THAT THE ROWS CONSIST OF ALL
      ⍝ INTEGER VECTORS <W> WITH K=+/W
      ⍝ AND (0≤W)∧W<V; THE VECTORS ARE
      ⍝ PRODUCED IN REVERSE LEXICO-
      ⍝ GRAPHICAL ORDER BY A NON-RECURSIVE
      ⍝ ALGORITHM
      Z←(0,ρV)ρ0
      →(0∊V)/0
      T←(ρV)ρI←0
     L0:B←K-+/I↑T
      R←B BREAK I↓V
      →(B≠+/R)/L1
      T←(I↑T),R
      Z←Z,[⎕IO] T
      I←¯1+⎕IO+ρV
     L1:I←((T>0)∧I>ιρT) RIOTA 1
      →(I<⎕IO)/0
      T[I]←T[I]-1
      I←I+1-⎕IO
      →L0
    ∇

    ∇ Z←K BREAK V;R;T
      ⍝ THE RESULT <Z> IS A VECTOR SUCH
      ⍝ THAT (ρV) = ρZ AND K = +/Z (IF
      ⍝ POSSIBLE) AND (0≤Z)∧Z<V AND THIS
      ⍝ IS THE LAST SUCH <Z> IN LEXICO-
      ⍝ GRAPHICAL ORDER
      T←(K≤+\V-1)ι1
      R←(T-⎕IO)↑V-1
      Z←(ρV)↑R,K-+/R
    ∇

    ∇ Z←V RIOTA K
      ⍝ LIKE DYADIC IOTA, BUT GIVES INDEX
      ⍝ OF FIRST OCCURRENCE OF <K> IN THE
      ⍝ VECTOR <V> SCANNING FROM THE RIGHT
      ⍝ TO THE LEFT; IF <K> DOES NOT OCCUR
      ⍝ IN <V>, THE RESULT IS ⎕IO-1.
      Z←(¯1+2×⎕IO)+(ρV)-(⌽V)ιK
    ∇
```

6. Acknowledgements.

The author would like to thank the referees for many helpful comments and suggestions.

Appendix: Simulation of Non-Standard Functions and Operators

A. *F*¨*A* denotes itemwise application of the function *F* to the array *A* such that

```
(F¨A) INDEX K ←→ F A INDEX K.
```

In the case where *F* is a scalar function, this can be simulated with

```
'F' IW A
```

where

```
   IW: (ρω)ρα ITEM ,ω
ITEM: (⍎α,' ',⍕1↑ω),α ITEM 1↓ω :
              0=ρω : ω
```

B. ∪A is Iverson's *nub* function (see [10]) and is a vector of the distinct elements chosen from the array *A*. The function below performs this task:

```
NUB: ((⍳ρω)=ω⍳ω)/ω←,ω
```

C. User-defined outer product may be mimicked with the use of the function *OP* below, which performs an outer product with respect to the function *F*; i. e.

```
A OP B ↔ A ∘.F B.

OP:  ((((ρρα)+⍳ρρω),⍳ρρα)⍉
       ((ρω),ρα)ρα) F ((ρα),ρω)ρω
```

D. Replication is an extension of the compression function, and is available as a primitive on some systems. It is denoted by *A/B* and replicates its right argument according to the pattern given by the left argument. For example,

```
      3 2 0 1 2/10 20 30 40 50
10 10 10 20 20 40 50 50
```

Replication can be simulated for vector arguments by the function *REP* below.

```
    ∇ Z←A REP B;M;T
[1]   ⍝ REPLICATES VECTOR <B> ACCORDING
[2]   ⍝ TO PATTERN <A>; THIS ALGORITHM
[3]   ⍝ FOR VECTORS IS BASED ON AN
[4]   ⍝ IDEA OF R. HEIBERGER
[5]    A←(T←A≠0)/A
[6]    M←(+/A)ρ0
[7]    M[⎕IO++\¯1↓A]←1
[8]    Z←(T/B)[⎕IO++\M]
    ∇
```

References

1. E. E. McDonnell and J. O. Shallit, "Extending APL to Infinity", Proceedings of the APL 80 Conference.

2. K. E. Iverson, "Notation as a Tool of Thought", Communications of the ACM, V. 23, No. 8 (August, 1980) pp. 444-465.

3. K. E. Iverson, "Operators and Functions", RC 7091, IBM Corporation, Yorktown Heights, N. Y., 1978.

4. Albert H. Beiler, *Recreations in the Theory of Numbers*, Dover Publications, New York: 1966, p. 219.

5. Z. Ghandour and J. Mezei, "General Arrays, Operators, and Functions", IBM Journal of Research and Development, V. 17, No. 4 (July, 1973), p. 339.

6. H. L. Royden, *Real Analysis*, The Macmillan Company, London: 1968, p. 31, 36.

7. K. E. Iverson, *Elementary Analysis*, APL Press, Swarthmore, Pa.: 1976.

8. William LeVeque, *Fundamentals of Number Theory*, Addison-Wesley, Reading, Mass.: 1977, p. 132.

9. K. E. Iverson, "The Derivative Operator", APL 79 Conference Proceedings, pp. 347-354.

10. K. E. Iverson, "Programming Style in APL", An APL Users Meeting, I. P. Sharp Associates, Ltd., 1978, pp. 200-224.