Scientific
Time Sharing
Corporation

7316 Wisconsin Avenue
Bethesda, Maryland 20014
301-657-8220

P036-0878

Scientific
Time Sharing
Corporation
World Leader in APL Time Sharing

APL★PLUS T.M. Service

COLLECTED WHIZBANGS

by Roy A. Sykes, Jr.

An Anthology
Of Tutorials on APL
Programming Techniques

Volume 1

Scientific
Time Sharing
Corporation

7316 Wisconsin Avenue
Bethesda, Maryland 20014

NOTE TO READERS

In December 1974 Scientific Time Sharing Corporation held a seminar for its APL*PLUS† Service Representatives. One outcome of that seminar was a "wish list" of goodies these marketing folks would like to have. On that list, they expressed a desire for some sort of "APL Whizbang" -- a column describing neat programming tricks that would illustrate some powerful, but perhaps not obvious, features of the language.

A year later, in delayed response to that request, the "Whizbang" was born in News, the APL*PLUS Service newsletter. The first "Whizbang" illustrated a programming technique for finding the current definition of a name. That was the first and last pure "Whizbang". In response to reader requests, the "Whizbang" evolved into a more general tutorial on APL programming techniques.

Over the past two and a half years, the reader response to the "Whizbang" has been gratifying. Today it remains one of the most popular features of News. For those who've never seen the "Whizbang" and for those who'd like to reminisce, we've collected past "Whizbangs" in this handy reference booklet.

Neither the "Whizbang" nor this booklet could exist without the devoted efforts of Karen Kromas and Beth Matsumune of STSC. Many thanks to them -- for patiently translating my barely legible scrawl into finished columns -- and to the other STSC people who review the drafts. And, most of all, thanks to the readers who make it all worthwhile.

Roy A. Sykes, Jr.
Los Angeles, California
August 1978

†APL*PLUS is a service mark and trademark of Scientific Time Sharing Corporation, registered in the United States Patent and Trademark Offices.

WHAT'S IN A NAME?

Finding the current definition of a named object in the workspace is useful in programs that may or may not reference global objects. A common technique, using the APL*PLUS primitive *⎕SIZE*, to find the storage required is:

```
      R←0<⎕SIZE 'OBJECTNAMES'
      ⍝ ⎕SIZE ←→ STORAGE REQUIRED
```

Unfortunately, this algorithm returns a 1 indiscriminately for groups and labels, as well as for functions and variables. Using *⎕IDLOC* (object definition) is more appropriate, but has the disadvantage of producing too much information; we wish only to know what the current definition is, not that at every level of the state indicator.

Before the scan operator's existence, the technique for picking out the proper element of *A←⎕IDLOC 'OBJECTNAMES'* (in *ORIGIN* 1) was cumbersome:

```
      R←((¯1+(¯1=⌽A)⍳1)⌽A)[;1]
```

With the advent of scan, the same fundamental technique was used, incrementally improved to:

```
      R←((+/∧\A=¯1)⌽A)[;1]
```

But using scan in another way gives us a much more elegant solution:

```
      R←(,<\A≠¯1)/,A
```

The last solution is fast and succinct, and works in either origin.

The results of these three algorithms are the same:

R	object is currently
0	undefined
1	a function
2	a variable
4	a group
8	a label

Observe that <\ is similar to ∨\, but only the first 1 on each row is preserved. Because the global definition of an object cannot be ¯1, we're assured of exactly a single 1 in each row of the result of <\.

One use of this technique is the emulation of subscripted or triadic APL functions (*A/B* or *A/[K]B*):

```
      ∇ R←A COMPRESS B;C
[1]     C←,⎕IDLOC 'COORD' ⋄ →(2=(<\C≠¯1)/C)⍴A̲
[2]     R←A/B ⋄ →0
[3]    A̲:R←A/[COORD] B ⋄ C←⎕ERASE 'COORD'
      ∇

      ∇ R←B SUB C
[1]     COORD←C ⋄ R←B
      ∇
```

```
        1 0 1 COMPRESS 3 3ρ⍳9 SUB 2
1 3
4 6
7 9

        1 0 1 COMPRESS 3 3ρ⍳9 SUB 1
1 2 3
7 8 9

        1 0 1 COMPRESS 3 3ρ⍳9
1 3
4 6
7 9
```

December 1975

PERMUTATION VECTORS IN APL - A RANK ANSWER

This article shows how one can use permutation vectors in APL to rearrange city names and population data from separate sources into a convenient report.

One common characteristic of the four expressions:

```
      S?S ⋄ ⍳S ⋄ ⍋V ⋄ ⍒V
```

is that they each generate a permutation vector. A permutation vector of length *N* is an arbitrary arrangement of ⍳*N*. A permutation vector may be recognized by the truth of either of the following expressions, which both assert whether *PV* contains exactly one of all its indices:

```
      ∧/(⍳ρPV)∊PV
      ∧/PV[⍋PV]=⍳ρPV
```

One may also test whether a vector is a permutation vector by:

```
      ∧/PV=⍋⍋PV
```

This statement depends on the fact that every permutation vector has a unique inverse permutation, expressed as ⍋*PV*. If *PV* is indeed a permutation vector, then the inverse of its inverse should be identical to itself.

Because permutation vectors represent all indices of a vector with ρ*PV* elements, they are commonly used in selection and reordering. For instance, *GV*←⍒*V* generates that permutation which, if applied as subscripts to *V*, would arrange *V* in descending order:

```
      V←?10ρ99 ⋄ V2←V[GV←⍒V] ⋄ ∧/V2=⌊\V2
1
      V,[1] GV,[0.5] V2
 85 41 15 40  5 42 80  2 54 78          (V)
  1  7 10  9  6  2  4  3  5  8         (GV)
 85 80 78 54 42 41 40 15  5  2         (V2)
```

The particular permutation *GV* is called a descending grade vector. The inverse permutation of a grade vector is called a ranking vector, which expresses the ordinality of the vector:

```
      V,[1] GV,[0.5] ⍋GV
 85 41 15 40  5 42 80  2 54 78          (V)
  1  7 10  9  6  2  4  3  5  8         (GV)
  1  6  8  7  9  5  2 10  4  3        (⍋GV)
```

⍋⍒*V* is the descending ranking vector on *V* because it is the permutational inverse of the descending grade vector of *V*. One might expect then that ⍋⍋*V* is the ascending ranking vector on *V*, and that is indeed the case.

Although grade vectors occur more commonly, ranking vectors have many uses in merging, rearranging, and reporting data. For example, suppose we have a matrix of city names, *NAMES*, arranged by state, an associated vector *EW* indicating whether the cities are eastern (0) or western (1), and two vectors, *EAST* and *WEST*, containing the population of the eastern and western cities.

```
      NAMES
LOS ANGELES, CA
SAN FRANCISCO, CA
DENVER, CO
WASHINGTON, DC
CHICAGO, IL
BOSTON, MA
NEW YORK, NY
PHILADELPHIA, PA
HOUSTON, TX
      EW
1 1 1 0 0 0 0 0 1

      EAST
757 3367 641 7868 1949
      WEST
2816 716 515 1233
```

We wish to produce a single listing, sorted by both city name and population. This is almost impossible to do (unless, fortuitously, the order of cities and populations correspond), because objects (lines of the report) can only be arranged in one order at a time.

However, ordering can also be indicated by a ranking vector, so we decide to arrange the report by city name, and express the population relationships by a ranking vector. The function *REPORT* uses two ranking vectors and one independent grade vector.

```
    ∇ REPORT;RA;ALP;POP;RANK;GRADE;FORMAT
[1]   ⍝ REQUIRES <NAMES,EW,EAST,WEST>.
[2]    ALP←' ,ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[3]    RA←⍴ALP ⋄ POP←(EAST,WEST)[⍋⍋EW]
[4]    RANK←⍋⍒POP←POP[GRADE←⍋RA⊥ALP⍳⍉NAMES]
[5]    'CITY_NAME________    _POP  RANK'
[6]    FORMAT←'17A1,X3,I4,X2,I3'
[7]    FORMAT ΔFMT(NAMES[GRADE;];POP;RANK)
    ∇
```

On line [3] of *REPORT* we use an ascending ranking vector, ⍋⍋*EW*, to merge the elements of *EAST* and *WEST*:

```
      EW ⋄ ⍋EW ⋄ ⍋⍋EW
1 1 1 0 0 0 0 0 1
4 5 6 7 8 1 2 3 9
6 7 8 1 2 3 4 5 9

      EAST,WEST ⋄ ⎕←POP←(EAST,WEST)[⍋⍋EW]
757 3367 641 7868 1949 2816 716 515 1233
2816 716 515 757 3367 641 7868 1949 1233
```

POP and *NAMES* now correspond; hence, we must reorder both by the same indices to maintain that relationship.

Line [4] of *REPORT* performs several operations. It generates an ascending grade vector by coding the city names into numbers and grading up those numbers; it reorders *POP* into ascending sequence by city name, saving *GRADE* so we can later reorder *NAMES* by the same permutation; and finally, it generates a descending ranking vector on *POP* to express the ordering of populations, although we will be using the ordering of city names to organize the report.

```
      ⎕←CODED←RA⊥ALP⍳⍉NAMES
2.09E24 3.02E24 8.95E23 3.59E24 7.67E23
      6.62E23 2.32E24 2.62E24 1.52E24

      ⎕←GRADE←⍋CODED
6 5 3 9 1 7 8 2 4

      ⎕←POP←POP[GRADE]
641 3367 515 1233 2816 7868 1949 716 757

      (⍒POP),[0.5] RANK←⍋⍒POP
 6 2 5 7 4 9 8 1 3                          (⍒POP)
 8 2 9 5 3 1 4 7 6                          (RANK)
```

Line [7] uses *GRADE*, which we computed on line [4] to sort by city name, to reorder *NAMES* in ascending sequence; it then uses Δ*FMT* to format the report. VOILA! -- Your very own almanac.

```
      REPORT
CITY_NAME________    _POP  RANK
BOSTON, MA            641    8
CHICAGO, IL          3367    2
DENVER, CO            515    9
HOUSTON, TX          1233    5
LOS ANGELES, CA      2816    3
NEW YORK, NY         7868    1
PHILADELPHIA, PA     1949    4
SAN FRANCISCO, CA     716    7
WASHINGTON, DC        757    6
```

April 1976

SUBSCRIPTION

This article discusses the use of subscription for selecting and changing scattered elements of an APL array.

The APL indexing (or subscription) function provides a powerful facility for identifying elements of an array. Once identified, those elements may be selected (DATA←V[I]) or changed (V[I]←DATA). An individual element is identified by one or more indices, depending on the rank of the left argument. Each set of indices sufficient to identify one element of an array is called an index-tuple. The right argument is a list of APL expressions representing the indices for each coordinate of the left argument. The expressions are separated by semicolons, and the entire list is enclosed in brackets.

```
      LEFTARG[R;I;G;H;T;...;A;R;G]
```

If all indices of a coordinate are used, the expression (but not the semicolon) may be elided for that coordinate.

The elements that are identified represent the cross product (outer product) of all specified indices of each coordinate coupled to all specified indices of the other coordinates. For instance, each element of a three-dimensional (3-D) array,

```
      ⎕←T← 2 3 5 ρι30
  1  2  3  4  5
  6  7  8  9 10
 11 12 13 14 15

 16 17 18 19 20
 21 22 23 24 25
 26 27 28 29 30
```

must be identified by three indices: layer, row, and column.

```
      L←2 ◇ R←1 ◇ C←4
      T[L;R;C]
19
```

If the result of any of the expresions is not a one-element array, each element is conceptually coupled with all elements of the other expressions to produce an array of index-tuples (in this case, each having three elements),

```
      L← 2 1 ◇ R←3 ◇ C← 1 2 5 2
      F←'G⍞ T[9;9;9] ⍞'
      F ΔFMT(L×100)∘.+(R×10)∘.+C
 T[2;3;1]  T[2;3;2]  T[2;3;5]  T[2;3;2]
 T[1;3;1]  T[1;3;2]  T[1;3;5]  T[1;3;2]
```

which are then used to select the data:

```
      T[L;R;C]
26 27 30 27
11 12 15 12
```

If we wish to select elements scattered throughout the array, the outer product index-coupling generally identifies more elements than we want. Essentially, we would like to disable the (outer product) cross-coupling, leaving only (scalar) parallel-coupling. The two techniques for doing this are:

- refine the result of the index function, and
- reduce the rank of the indexed array.

It is important to note that, for vectors, only one index is needed to identify each element. Hence, vector indexing already can do scattered-point selection because no index-coupling is performed.

```
      V←' BRUTE' ◇ V[6 5 1 5 4]
ET TU
```

However, for matrices, two indices are needed to identify each element. Indexing produces these two-element index-tuples by coupling all specified row and column indices:

```
      M←'SPADE  CLUB   HEART  DIAMOND'
      ⎕←M← 4 7 ρM
SPADE
CLUB
HEART
DIAMOND

      M[1 2 4 3 1 ; 1 3 2 5 1]
SAPES
CUL C
DAIOD
HAETH
SAPES
```

Using transpose to refine the result

Suppose that what we really want in the above case is simply a pairwise coupling of the row and column indices, for example:

```
      M[1;1],M[2;3],M[4;2],M[3;5],M[1;1]
SUITS
```

Look back at the major diagonal (top left to bottom right) of the matrix result. Primitive indexing has in fact selected what we want, and more. By using dyadic transpose, we can select that diagonal, thus achieving the effect of a pairwise coupling of the row and column indices:

```
      1 1 ⍉M[1 2 4 3 1 ; 1 3 2 5 1]
SUITS
```

This technique applies to selecting scattered elements out of any array, assuming the index expressions are vectors. For A[V1;V2;V3;...;VN̲] (where N̲←ρρA), one would use

```
      (N̲ρ1)⍉A[V1;V2;V2;...;VN̲]
```

and the result would be a vector of length

```
      (ρV1)⌊(ρV2)⌊(ρV3)⌊...⌊ρVN̲
```

If we generalize the problem somewhat we find that we can primitively select:

scattered points (scalar elements) from vectors by *V*[*C*], and

scattered lines (row/column vectors) from matrices by *M*[*R*;] or *M*[;*C*], and

scattered planes (matrices) from 3-D arrays by *T*[*L*;;], *T*[;*R*;], or *T*[;;*C*].

(Elided coordinate indices may be replaced by any APL expression.)

Remember that we can select scattered points from a matrix, but only if they all fall on a single row or column. The same principle applies to scattered points or lines in a 3-D array. Essentially

```
      M[3; 5 1 2 3 5 2 4]
THEATER
```

is really only selecting scattered points from the vector

```
      M[3;]
HEART
      (M[3;])[5 1 2 3 5 2 4]
THEATER
```

Conversely, we cannot select:

scattered scalars from matrices or higher-order arrays, or

scattered vectors from 3-D or higher-order arrays, or

scattered matrices from 4-D or higher-order arrays,

unless all the selected subarrays are identified by single indices along all other coordinates (thus effectively reducing the rank of the indexed array).

In general, if we wish to select more than one subarray (e.g., a scalar) from an array of rank greater than one higher than that subarray (e.g., a matrix), we must use a slicing transpose to obtain the result. The following table illustrates scattered-array indexing for common cases where *L*, *R*, and *C* are vectors:

Select scat-tered	from a vector	from a matrix	from a 3-D array
points	1⍉*V*[*C*]*	1 1⍉*M*[*R*;*C*]	1 1 1⍉*T*[*L*;*R*;*C*]
cols	1⍉*V*[*C*]*	1 2⍉*M*[;*C*]*	2 1 2⍉*T*[*L*;;*C*]
rows	1⍉*V*[]*	1 2⍉*M*[*R*;]*	1 1 2⍉*T*[*L*;*R*;]
planes		1 2⍉*M*[;]*	1 2 3⍉*T*[*L*;;]*

(*) The ⍉ may be elided from these expressions.

Here's an example of indexing scattered rows:

```
      J←'JELL    JELLO   JELLY   JELLIED'
      G←'GEL     GELD    GELT    GELATIN'
      ⎕←JG←(4 7 ⍴J),[0.5] 4 7 ⍴G
JELL
JELLO
JELLY
JELLIED

GEL
GELD
GELT
GELATIN

      1 1 2 ⍉JG[1 1 1 2 2 ; 1 4 3 4 1 ;]
JELL
JELLIED
JELLY
GELATIN
GEL
```

Reducing the rank of the indexed array

The transpose technique is concise, but it has several disadvantages:

- For large right arguments (the indices) it requires a great deal of temporary workspace storage and extra computer time to select all the superfluous data before the transpose is performed.
- Altering the shape of the result necessitates potentially confusing changes to both the indices and the left argument to transpose.
- It does not allow specification.

An alternative technique is to reduce the rank of the indexed array such that we can use primitive indexing. This requires that we somehow recalculate the indices to correctly identify the desired data. In the preceding example (selecting scattered rows):

```
      (8 7 ⍴JG)[1 4 3 8 5 ;]
JELL
JELLIED
JELLY
GELATIN
GEL
```

The indices were actually calculated by:

```
      IND← 1 1 1 2 2 ,[0.5] 1 4 3 4 1
      ⎕←IND←(⍳1)+(¯1↓⍴JG)⊥IND-⍳1
1 4 3 8 5
```

Notice the dependence on the *ORIGIN* (⍳1) to offset the indices when working in *ORIGIN* 1. Number systems (which is actually what ⊥ deals with) are based in *ORIGIN* 0.

A more common case is selecting scattered points from an arbitrary array. Since only vector indexing primitively allows selection of scattered points, we must reduce the rank of the array to 1 (e.g., ravel it) and calculate the subscripts (assume *ORIGIN* 1):

```
      L← 1 1 1 1 2 2 2 2 1 2 2 2 2 2 2
      R← 1 1 1 1 2 4 4 4 2 2 4 2 3 2 1
      C← 1 2 3 4 7 6 7 5 5 7 4 7 1 2 3
      ⍝  J E L L   I N T O   A   G E L

      (,JG)[1+(⍴JG)⊥(L,[1] R,[0.5] C)-1]
JELL INTO A GEL
```

It may be clearer if we look at ravelled *JG* (after removing spaces), its indices, and the indices we calculated:

```
      B←' '≠V←,JG ◇ ⍉'I2' ΔFMT B/⍳⍴B ◇ B/V
      11111111222222223333333444445555555
123489012567892345678901678934560123456
JELLJELLOJELLYJELLIEDGELGELDGELTGELATIN

      1+(⍴JG)⊥(L,[1] R,[0.5] C)-1
1 2 3 4 42 55 56 54 12 42 53 42 43 37 31
```

Notice how the base value operation (⊥) had the effect of multiplying *L* by 28 (number of elements in each layer), *R* by 7 (number of elements in each row), and *C* by 1.

In fact, we can write a generalized scattered-point selection function as follows:

```
    ∇ R←A SPS B;C
[1]   ⍝ SCATTERED-POINT SELECTION.
[2]   ⍝ <A> IS THE ARRAY TO BE SUBSCRIPTED.
[3]   ⍝ <B> IS THE ARRAY OF COORDINATE
[4]   ⍝ INDICES IN <A> SUCH THAT ITS LAST
[5]   ⍝ COLUMN REPRESENTS COLUMN INDICES,
[6]   ⍝ THE NEXT-TO-LAST (IF ANY) IS ROW
[7]   ⍝ INDICES, THE NEXT IS LAYERS, ETC.
[8]   ⍝ CONFORMABILITY: (¯1↑1,⍴B)∊1,⍴⍴A
[9]   ⍝ ORIGIN DEPENDENT.  (⍴R)=¯1↓⍴B
[10]   R←⍴A ◇ C←''⍳1
[11]   R←(,A)[(R⊥(1⌽⍳⍴⍴B)⍉B)+C-R⊥C]
    ∇
```

A scattered-point assignment function, utilizing a global to hold the data, might be defined as:

```
    ∇ R←A SPA B;C
[1]   ⍝ SCATTERED-POINT ASSIGNMENT.
[2]   ⍝ SIMILAR TO <SPS> BUT THE RESULT IS
[3]   ⍝ THE LEFT ARGUMENT <A> MODIFIED BY
[4]   ⍝ THE GLOBAL <DATA>.
[5]    R←,A ◇ C←''⍳1 ◇ A←⍴A
[6]    R[(A⊥(1⌽⍳⍴⍴B)⍉B)+C-A⊥C]←DATA ◇ R←A⍴R
    ∇
```

In both of these functions, the *ORIGIN* offset is calculated in a slightly different fashion to maximize efficiency for large index arrays.

A few examples should show the power of *SPS*:

```
      ⎕←M← 5 7 ⍴⍳35
  1  2  3  4  5  6  7
  8  9 10 11 12 13 14
 15 16 17 18 19 20 21
 22 23 24 25 26 27 28
 29 30 31 32 33 34 35

      M SPS 5 2 ⍴ 1 1, 1 4, 2 2, 3 2, 4 4
1 4 9 16 25

      M SPS 2,[1.5]⍳5     ⍝ ROW 2; COLS 1-5
8 9 10 11 12

      M SPS(⍳5),[1.5] 2  ⍝ ROWS 1-5; COL 2
2 9 16 23 30

      M SPS 5 1 ⍴⍳5 ⍝ LAST COORD. EXTENDED
1 9 17 25 33

      ROWS←1,[1.5]⍳5 ◇ COLS←(⍳5),[1.5]7-⍳5

      5 0 2 0 ⍕ROWS,COLS
    1 1    1 6
    1 2    2 5
    1 3    3 4
    1 4    4 3
    1 5    5 2

      7 0 ⍕M SPS ROWS,[2.5] COLS
      1      6
      2     12
      3     18
      4     24
      5     30
```

The function *SPA* might be used as follows:

```
      DATA←¯2
      ⎕←M←M SPA 5 2 ⍴ 1 1,1 4,2 2,3 2,4 4
 ¯2  2  3 ¯2  5  6  7
  8 ¯2 10 11 12 13 14
 15 ¯2 17 18 19 20 21
 22 23 24 ¯2 26 27 28
 29 30 31 32 33 34 35

      DATA← ¯3 ¯3 ¯4 ¯5
      ⎕←M←M SPA 4 2 ⍴ 2 1, 4 5, 3 2, 5 4
 ¯2  2  3 ¯2  5  6  7
 ¯3 ¯2 10 11 12 13 14
 15 ¯4 17 18 19 20 21
 22 23 24 ¯2 ¯3 27 28
 29 30 31 ¯5 33 34 35

      DATA← 4 4 ⍴⍳4
      R← 1 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5
      C← 6 3 5 7 1 4 6 7 1 3 5 7 2 5 6 7
      ⎕←M←M SPA (4 4 ⍴R),[2.5] 4 4 ⍴C
 ¯2  2  3 ¯2  5  1  7
 ¯3 ¯2  2 11  3 13  4
  1 ¯4 17  2 19  3  4
  1 23  2 ¯2  3 27  4
 29  1 31 ¯5  2  3  4
```

June 1976

SOME NOTES ON DATE STORAGE

The processing of calendar dates often involves fairly complex conversions between the stored representation and what is presented to the user. Typically, dates are stored in one of two formats, both designed to allow the representation of a date as a single number:

- Ordinal Representation
  Dates in this form are stored as integers indicating the number of days elapsed since a certain base date. In particular, the APL*PLUS File Subsystem stores *⎕FRDCI* timestamps as ordinal values relative to March 1, 1960. Given a file timestamp *TS*, the number of elapsed days can be calculated by ⌊*TS*÷5184000 (5184000 = the number of sixtieths of a second in one day).

- Packed Representation
  Dates in this form are stored as 100⊥*YEAR*,*MONTH*,*DAY*. The form is commonly abbreviated to YYMMDD, expressing only dates in the twentieth century. Today's date represented as a packed number can be calculated by
  ```
        1000000|100⊥3↑⎕TS
  760714 ⍝ BASTILLE DAY
  ```

Both representations allow qualitative analysis among dates. That is, dates stored in either format may be compared and ordered directly. However, quantitative analysis (e.g., "How many days between ...") is possible only with the ordinal representation.

Ordinal dates are not as commonly used as packed dates because they are significantly more expensive to process, particularly in the complex conversion necessary to display them in a meaningful format. Frequently, dates are stored in packed format, and occasionally converted to ordinal format for quantitative processing.

Dates also have a variety of display formats, such as:

```
7/4/76     SEPTEMBER 2, 1945     31 MAY 69
```

The standard format used in columnar reports is MM/DD/YY (although in Europe the accepted form is DD/MM/YY). Assuming our dates are in a vector *DATES* in YYMMDD form, how can we most efficiently display the standard format?

The most obvious approach is to unpack the dates into three numbers each, reorder them, and format each column:

```
D←⌽1⊖ 100 100 100 ⊤DATES
'I2,⍞/⍞,ZI2,⍞/⍞,ZI2' ⎕FMT D
```

We can reduce the number of format phrases, and speed up *⎕FMT*'s operation, by the following:

```
'I2,2P⍞/⍞ZI3' ⎕FMT D
```

To really speed up the process, however, we should consider formatting only one number per date. This can be done by repacking the dates in the desired order, and using the *G* format phrase to insert the slash (/) decorations:

```
D←100⊥1⊖ 100 100 100 ⊤DATES
'G⍞Z9/99/99⍞' ⎕FMT D
```

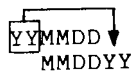
The reordering can be speeded up even more by:

```
D←100⊥⊖ 0 10000⊤DATES
```

which reduces the required number of internal arithmetic and data-movement operations.

A disadvantage of the above techniques is that they all require a large amount of intermediate storage; in fact, they may temporarily require up to six times the working area for each date! The solution is some clever algebraic manipulation of the digits within each date.

Here's a symbolic illustration of what we want:



1. Let's start by picking off the month and day:
   ```
       10000|YYMMDD
   MMDD
   ```

2. Multiplying that by 1000000 (6 digits) produces:
   ```
       1000000×MMDD
   MMDD000000
   ```

3. To which we add the original dates:
   ```
       YYMMDD+MMDD000000
   MMDDYYMMDD
   ```

4. Now all we have to do is divide by 10000 (to eliminate the trailing four digits that represent *MMDD*):
   ```
       MMDDYYMMDD÷10000
   MMDDYY.MMDD
   ```

5. and floor the result:
   ```
       ⌊MMDDYY.MMDD
   MMDDYY
   ```

It could be written in one statement:

```
⌊(YYMMDD+1000000×10000|YYMMDD)÷10000
```

There's yet a further simplification that will eliminate the ⌊, and allow the whole calculation to proceed in integer representation. The fractional part of step (4) is exactly (10000|*YYMMDD*)÷10000. Given *MD*←10000|*YYMMDD*, the statement above becomes:

```
⌊(YYMMDD+1000000×MD)÷10000
```

which is equivalent to:

```
((YYMMDD+1000000×MD)÷10000)-MD÷10000
```

because of the fraction in step (4). Now distribute the denominator (10000):

```
((YYMMDD+1000000×MD)-MD)÷10000
```

Reassociate the numerator:

```
(YYMMDD+(1000000×MD)-MD)÷10000
```

Add common terms (distribute *MD*):

```
(YYMMDD+999999×MD)÷10000
```

and expand *MD*:

```
(YYMMDD+999999×10000|YYMMDD)÷10000
```

This statement requires one-third the dynamic storage of its equivalent:

```
100⊥⊖ 0 10000 ⊤DATES
```

and executes up to three times faster!

As an exercise, try applying these principles to converting dates input in MMDDYY form to their YYMMDD representation.

August 1976

"... programs execute faster in less space". Those are sweet words to APL users. They mean functions will consume fewer CPU cycles, take less connect time, and require less workspace storage. The net effect is lower cost and increased utilization of the APL workspace.

That third benefit is important. Workspaces hold only a limited amount of information. They must accommodate functions, variables, groups, a symbol table (to store object names), execution stack information (the state indicator), and other internal directories. Moreover, during execution of expressions, the active workspace is continually allocating and freeing storage for temporary values.

By making efficient use of the free working area of the active workspace (measured by ⎕*WA*), the APL*PLUS System reduces the incidence of *WS FULL* errors. The user can therefore store more programs or process more data than is otherwise possible, thus leading to additional benefits in programmer productivity, more efficient execution, and reduced internal overhead.

There are four fundamental, as well as several specialized, mechanisms currently employed by the APL*PLUS System to conserve available workspace storage resources. All are invoked automatically, and generally have no immediately apparent effect on running programs. Without them, however, usable working area would be drastically reduced, and *WS FULL* would be a recurring nightmare to programmers.

When executing a function, the APL System typically requires space for both the argument(s) and the result simultaneously. For example, during execution of

```
R←(VEC≠0)/MAT[;2 3 4]
```

the system will, at one instant, require space for the temporary value *VEC*≠0, the other temporary value *MAT*[;2 3 4], <u>and</u> the result of the compression. Essentially,

```
TEMP1←VEC≠0 ◇ TEMP2←MAT[;2 3 4]
R←TEMP1/TEMP2 ◇ ⎕ERASE 'TEMP1 TEMP2'
```

Of course, *VEC* and *MAT* are held in the workspace during the entire process. Given this general rule, let's see how the system tries to conserve space.

1. <u>In-place</u> <u>storage</u>

Often the system recognizes that the result of an operation fits into the space used by one of its temporary arguments, and is "smart" enough to use that same area to store the result. For instance,

```
A←+\⍳100
```

requires space only for ⍳100; the +\ is subsequently stored over that temporary value. This effect would not take place for *A*←+*A* because the argument to scan is not a temporary value.

Other examples of in-place storage are:

```
R←A[⌽⍳50] ⍝ BOTH FOR ⌽ AND []
R←101ρ2+⍳100 ⍝ BOTH FOR ρ AND +
```

Some functions, like reversal and reshape, normally make copies if there is room in the workspace, and resort to in-place storage and slower algorithms only if necessary to avoid a *WS FULL* condition.

2. Chained variables

Chained variables can result in significant savings when using defined functions with named arguments. Most APL systems materialize often superfluous (wasteful) copies of named arguments; the APL*PLUS System defers copying until necessary -- sometimes not at all!

The following function returns an ascending grade vector on its numeric matrix argument:

```
     ∇ R←GRADE MAT;IO;COL
[1]    IO←''⍳1 ◇ R←⍳(ρMAT)[IO]
[2]    →(COL←(ρMAT)[IO+1])↓0 ◇ COL←COL-~IO
[3]   A:R←R[⍋MAT[R;COL]] ◇ →A×IO≤COL←COL-1
     ∇
```

Given *M*←? 100 20 ρ100, executing *GV*←*GRADE M* requires ⎕*WA* to be slightly more than 1400 bytes. The traditional function-calling mechanism would have required ⎕*WA* to be more than 9400 bytes to avoid a *WS FULL*.

3. Datatype conservation

Numbers stored in binary, integer, or floating-point format require 1, 32, or 64 bits per number, respectively. Several enhancements in the APL*PLUS System deal with conserving storage by maintaining datatype wherever possible. For example,

```
60÷⍳6 ◇ ¯2*⍳31 ◇ !⍳12
```

all return 4-byte integer results, rather than the 8-byte floating-point results given by other APL systems. Similarly, dyadic ×, *, ⌈, ⌊, |, and ! operations on Boolean arguments preserve the binary datatype. Also, indexed assignment attempts to maintain the datatype of the target variable whenever possible. For example:

```
A←100ρ0 ◇ A[]←2○0 ⍝ COS OF 0 RADIANS
```

preserves *A* as binary, even though 2○0 is a floating-point 1.

4. Name referencing

Several identity operations perform no actual data movement in the workspace at all. Instead they pass internal pointers to the right argument. This occurs for both named and temporary arguments. For instance, the following expression takes almost no workspace storage beyond that required for *CV*:

```
CV←80000ρ'ABC'
CV←(ρCV)ρ⍕1/,⍎'CV[]'
⍝       ^^ ^^^   ^   PASS POINTERS
```

A particularly common use is functions that ravel arguments which are predominantly vectors anyway.

5. Specialized mechanisms

Besides the techniques described above, several others are used when specific syntactic constructions are encountered. For instance

```
'I80' ⎕FMT 10000ρ1
```

requires only 1500 bytes for execution, even though the "result" would seem to require 800,000 bytes. Similarly,

```
BV/⍳ρBV
```

requires space only for the result. In the case where *BV*←¯500000↑1, ⎕*WA* need be only 76 bytes to execute the expression, rather than the over 2 million bytes that might otherwise be necessary. Another special case is →⎕*LC*, which requires only 40 bytes for execution, regardless of the size of the state indicator. In fact, →⎕*LC* may work when ⎕*LC* alone engenders a *WS FULL* error!

Of course, a large workspace always helps. As recently as 1972, an APL*PLUS workspace was limited to 32,000 bytes -- now it's over 100,000! Theoretically, we could allow even larger workspaces. But keeping workspaces reasonably-sized and making maximum use of that storage increases the throughput of the APL*PLUS System, and assures maximum machine utilization, economical operation, and the good response time which users depend on.

The ability to bring data and functions in from files, to dynamically expunge objects, and to automatically load other workspaces also helps the user avoid *WS FULL* errors. And, of course, there's always)*ERASE*! But the automatic mechanisms used by the APL*PLUS System are the silent sentries in stretching storage.

October 1976

BRANCHING AND ITERATION

This article discusses some guidelines and techniques for branching and iteration in APL. Many people have asked me, "What's the fastest way to branch?" I hope to shed some light on this often-maligned subject.

Guidelines for Branching

The following three guidelines will simplify the maintenance of your code and improve its readability. The techniques you use are not as important as how consistently you use them.

1. Branch to labels, not line numbers. If your branch targets are line numbers rather than labels, simply inserting or deleting lines from a program can become a heinous task. Furthermore, minor benefits in storage and execution speed are gained by using label variables rather than constants.

2. Adopt a consistent notation for labels. Labels like *LOOP*, *AGAIN*, *DO*, *MORE*, and *OOPS* seldom add clarity to programs, especially large programs. Furthermore, they give the reader little or no clue as to where in the function they refer, or whether they are in fact labels. Common notations are {*L*1;*L*2;*L*3;...} and {*A̲*;*B̲*;*C̲*;...}. More mnemonic conventions can help the reader recognize program segments (for example, *LP*1 or *ER*2). The notation or convention that you choose is irrelevant as long as you adhere to it.

3. Adopt consistent techniques for branching and iteration. For instance, some people favor

```
      →LABEL×⍳condition
```

while others use

```
      →(condition)⍴LABEL
```

Again, it's not so important which techniques you use (except for trivial differences in execution speed), but that you use them consistently. Among my personal favorites are:

```
→(condition)⍴LABEL   or →(condition)↓LABEL
→(×⍴vector)⍴LABEL    or →(⍴vector)↓LABEL
→(conditions)/LABEL  or →(∧/conditions)⍴LABEL
→LABEL×condition     or →LABEL×~condition
```

Some Useful Techniques

1. Subroutines to handle branching logic.

Programs can often be made more readable by using functions with mnemonic names for branch calculations. For instance,

```
... ◇ →THISL IF M≥C←C+1
```

is certainly more understandable than

```
... ◇ →(M≥C←C+1)↑⎕LC
```

or

```
T̲: ... ◇ →T̲⌈⍳M≥C←C+1
```

You can define the functions *IF* and *THISL* as:

```
    ∇ R←A IF B              ∇ R←THISL
[1]   R←B/A             [1]   R←1↑1↓⎕LC
    ∇                       ∇
```

IF can also be used for multiple conditions. For instance,

```
      →(L4,L5) IF ¯1 1=×A
```

will branch to *L*4 if *A* is negative, *L*5 if *A* is positive, and fall through if *A* is zero.

These functions are especially beneficial in programs with complex branching logic and many transfers of control. However, in simple iteration (see below) they exact a small but noticeable penalty in CPU resources.

2. Faster techniques for iteration.

The following techniques, while quite fast, generally detract from readability and should therefore be used judiciously.

Typically, loops involve a leading test as exemplified by *ITERATE*.

```
    ∇ ITERATE M;C
[1]   SETUP ◇ C←1
[2]   →0 IF C>M ◇ PROCESS ◇ C←C+1 ◇ →⎕LC
    ∇
```

We can produce a speedier version of *ITERATE* by eliminating the *IF* subroutine and using a label.

```
    ∇ ITERATE2 M;C
[1]   SETUP ◇ C←1
[2]   A̲:→(C>M)⍴0 ◇ PROCESS ◇ C←C+1 ◇ →A̲
    ∇
```

By employing a trailing test, the branch overhead can be reduced significantly. The once-executed leading test, →*M*↓0, accommodates the zero case.

```
    ∇ ITERATE3 M;C
[1]   SETUP ◇ →M↓0 ◇ C←1
[2]   A̲:PROCESS ◇ →(M≥C←C+1)⍴A̲
    ∇
```

Further, by precalculating all branch points, we can avoid the serial iteration-by-iteration testing. APL permits (and even encourages) parallel processing on data -- why not parallel calculations, testing, and branch-point determination? *ITERATE*4 illustrates a simplified form of this technique.

```
     ∇ ITERATE4 M;C;L
[1]    SETUP ◇ C←1 ◇ →L←(MρA̲),0
[2]    A̲:PROCESS ◇ →L[C←C+1]
     ∇
```

Note that all of the above algorithms correctly handle the case where zero iterations are requested. They are also restartable; that is, if *PROCESS* produces a *WS FULL* or *VALUE ERROR*, the problem can be corrected, and →⎕*LC* will resume execution normally. It is true that *ITERATE*2 could have been shortened to

```
[1]    SETUP ◇ C←0
[2]    A̲:→(M<C←C+1)ρ0 ◇ PROCESS ◇ →A̲
```

However, resumption after a *VALUE ERROR* on *PROCESS* would have incremented *C* twice (once before the error, and once after the restart), thereby skipping one iteration.

The following table illustrates the relative CPU timings of the above techniques with varying numbers of iterations. The ratios are to *ITERATE*4; *SETUP* and *PROCESS* were set as empty matrices.

	Iterations (*M*)				
	0	1	10	100	1000
ITERATE	1.11	1.60	2.47	2.89	2.97
*ITERATE*2	.90	1.08	1.45	1.67	1.68
*ITERATE*3	.65	.92	1.23	1.40	1.41
*ITERATE*4	1.00	1.00	1.00	1.00	1.00

(Executed 11/29/76 at 7:27 P.M. EST on *APLPLUSC* with 20 users)

3. ⍎ Recursed!

Did you know that you can use the execute function (⍎) to effect iteration in desk calculator mode? Assume *M* is the number of iterations (*M*≥1), and *C* is the counter (initialized as *C*←1). Now compare the following two lines -- one in a function, the other in desk calculator mode:

```
[N]  L: PROCESS ◇ →(M≥C←C+1)/L
     ⍎L←'PROCESS ◇ ⍎(M≥C←C+1)/L'
```

Both have the effect of executing *PROCESS* as many times as called for by *M*. Observe that whereas the first line uses branching to the label *L*, the second uses ⍎ to recursively execute the statement represented by *L*. Unpleasantly, the execute construction consumes significantly more CPU time and is far more extravagant in its use of workspace storage. It does, however, illustrate the interesting relationship between → and ⍎.

The ability to rapidly locate string occurrences in other strings is often useful. The many techniques applicable to the problem also provide enlightening examples of efficient problem-solving in *APL*. This article discusses several algorithms for string searching to illustrate efficiency of execution and some interesting techniques.

A <u>string</u> is a vector, typically character. String searching means locating the occurrences of one string, called the <u>substring</u>, in another string, called the <u>target string</u>. The result can be a Boolean vector of the same length as the target string with 1's denoting the leading element of each match. More commonly (see each function below), the result is the origin-sensitive indices of the 1's in the Boolean vector (i.e., *BV*/⍳ρ*BV*). Overlapping matches are possible. In all functions, the left argument *A* is the target string and the right argument *B* is the substring for which we will search.

The most fundamental *APL* solution is quite straightforward:

```
     ∇ R←A SS1 B;C;D;E
[1]    C←ρA←,A ◇ D←ρB←,B ◇ E←0⌈C-0⌈D-1
[2]    R←(E↑∧/((⍳D)-⍳1)⊖A∘.=B)/⍳E
[3]   ⍝ OR ALTERNATIVELY,
[4]   ⍝R←(E↑∧⌿((⍳D)-⍳1)⌽B∘.=A)/⍳E
     ∇
```

All characters in the target string are compared with all characters in the substring, the logical matrix is shifted to align each potential occurrence, and the indices of complete matches are selected. Notice that if the length of the substring *B* is greater than that of the target string *A*, then no matches are possible. Furthermore, if the substring is empty, all indices are returned. The number of potential matches, *E*, is normally 1 plus the difference in lengths of the strings, that is, *E*←1+*C*-*D* or *E*←*C*-*D*-1. The two 0⌈'s are to correctly handle empty or overlong substrings.

A somewhat more elegant solution employs inner product to detect matches:

```
     ∇ R←A SS2 B;C;D;E
[1]    C←ρA←,A ◇ D←ρB←,B ◇ E←0⌈C-0⌈D-1
[2]    R←(E↑B∧.=(D,C+1)ρA)/⍳E
[3]   ⍝ OR ALTERNATIVELY,
[4]   ⍝R←(E↑B∧.=((⍳D)-⍳1)⌽(D,C)ρA)/⍳E
     ∇
```

Rather than rotating a logical array, this technique builds a matrix of all possible matches and then uses ∧.= to select those that are complete. Observe the pleasant effect of the reshape on line [2] -- it both builds <u>and</u> shifts the matrix. *SS*2 is somewhat faster than *SS*1, although it requires more workspace area (⎕*WA*) in which to execute.

Unfortunately, both of these algorithms are inherently inefficient because they must compare <u>all</u> possible combinations of characters in the two strings; that is, they are "blind", and continue to process even if no matches are possible. A somewhat better approach is to determine the indices of the first character, and compare only the remaining candidates with the rest of the string:

```
     ∇ R←A SS3 B;C;D;E
[1]     C←ρA←,A ◊ D←ρB←,B ◊ →B̲⌊A̲+D
[2]   A̲:R←⍳C ◊ →0 ⍝         EMPTY STRING
[3]     R←(A∊B)/⍳C ◊ →0 ⍝ SINGLETON STRING
[4]   B̲:E←0⌈C-D-1 ◊ →(ρR←(E↑A∊1↑B)/⍳E)↓0
[5]     R←(A[R∘.+(¯1*⍳1)↓⍳D]∧.=1↓B)/R
     ∇
```

This function employs some new techniques worth discussing. Notice the unusual branch on line [1] which has the effect of trapping both the empty and one-element cases. If desired, line [2] could then be modified to return ⍳0 in the empty case rather than all indices -- a more usable result.

Line [4] calculates the number of potential matches, and locates the occurrences of the first character. (For character arguments, ∊ is typically faster than = because of the internal algorithm used.) If the first character is not found, the function exits immediately with an empty result. Otherwise, a matrix of all remaining candidates is built via outer product and subscription, and ∧.= is used to produce the compression vector which selects only complete matches. The (¯1*⍳1)↓⍳*D* construct cleverly accounts for the index origin while removing the index of the character already examined.

While *SS*3 is a considerable improvement over *SS*1 and *SS*2, it is still relatively naive. For instance, if the first character of the substring occurs frequently in the target string, little or no savings may accrue. Furthermore, the risk of a *WS FULL* is increased because of the larger space required by the matrix of indices. A more sophisticated algorithm employs basically the same technique, but attempts to minimize the number of initial "hits" by searching for the character in the substring that has the <u>least</u> number of <u>expected</u> occurrences:

```
     ∇ R←A SS4 B;C;D;E;F;G
[1]     C←ρA←,A ◊ D←ρB←,B ◊ →B̲⌊A̲+D
[2]   A̲:R←⍳C ◊ →0 ⍝         EMPTY STRING
[3]     R←(A∊B)/⍳C ◊ →0 ⍝ SINGLETON STRING
[4]   B̲:F←' ETAISONRLCFHUDMPBWYGKVJQXZ.,'
[5]     E←0⌈C-D-1 ◊ G←F⍳B ◊ F←G⍳⌈/G
[6]     →(ρR←(E↑(F-⍳1)↓A∊B[F])/⍳E)↓0
[7]     G←(F≠⍳D)/⍳D
[8]     R←(A[R∘.+G-⍳1]∧.=B[G])/R
     ∇
```

Line [4] specifies a letter distribution vector arranged in descending order of expected frequency in normal text. By line [6], *F* has become the index in the substring of the letter with the fewest probable occurrences in the target string. This letter is used for the first comparison. Its index is then removed on line [7] for the outer product calculation of indices and ∧.= comparison on line [8].

Since we have concluded that reducing the number of potential hits on one character is beneficial, why not do it on two or three? In fact, why not conclude the entire process by constantly refining the potential matches until none remain, or until we have exhausted the substring? *SS*5 does just that:

```
     ∇ R←A SS5 B;C;D;E
[1]     C←ρA←,A ◊ D←ρB←,B ◊ →B̲⌊A̲+D
[2]   A̲:R←⍳C ◊ →0 ⍝         EMPTY STRING
[3]     R←(A∊B)/⍳C ◊ →0 ⍝ SINGLETON STRING
[4]   B̲:E←0⌈C-D-1 ◊ →(ρR←(E↑A∊1↑B)/⍳E)↓0
[5]     B←(⍳C←1)↓B
[6]   C̲:→(ρR←(A[R+C]∊B[C])/R)↓0
[7]     →C̲×D>C←C+1
     ∇
```

Through line [4] the function is identical to *SS*3; that is, both empty and one-element strings are accommodated, and the first character of longer strings has been compared. Line [5] initializes the substring element counter, and modifies *B* so that the counter is appropriate in either origin. Lines [6] and [7] compose the loop; on line [6] the *C*th character beyond existing matches in the target string is compared with the *C*th character of the remaining substring. If no matches remain, the function terminates. Otherwise, the counter on line [7] is incremented and the function branches back to line [6] if there are more characters to compare in the substring.

In most cases, *SS*5 is highly efficient. It slows down somewhat if the substring is quite long because it must iterate on each element. On the other hand, its antecedents are all dangerously prone to *WS FULL* problems in similar situations, especially if the target string is also long. However, *WS FULL* errors are still possible in *SS*5, particularly if the first character of the substring occurs often in the target string. This situation can be remedied in much the same way as was illustrated in the progression from *SS*3 to *SS*4:

```
     ∇ R←A SS6 B;C;D;E;F;G
[1]     C←ρA←,A ◊ D←ρB←,B ◊ →B̲⌊A̲+D
[2]   A̲:R←⍳C ◊ →0 ⍝         EMPTY STRING
[3]     R←(A∊B)/⍳C ◊ →0 ⍝ SINGLETON STRING
[4]   B̲:F←' ETAISONRLCFHUDMPBWYGKVJQXZ.,'
[5]     E←0⌈C-D-1 ◊ B←B[G←⍋F⍳B]
[6]     G←G-F←⍳C←1
[7]     →(ρR←(E↑G[F]↓A∊B[F])/⍳E)↓0
[8]     G←F↓G ◊ B←F↓B
[9]   C̲:→(ρR←(A[R+G[C]]∊B[C])/R)↓0
[10]  →C̲×D>C←C+1
     ∇
```

An expected letter distribution vector is used to rearrange the elements of the substring so that the fewest possible matches are likely for each iteration.

Besides reducing the expected storage requirements, this technique normally speeds up the search further because fewer comparisons need be performed.

Some conclusions can be drawn from the above discussion:

- the shortest solution is not necessarily the fastest;

- handling special cases as such is often beneficial;

- iteration can be faster than "closed-form" code; and

- intelligent analysis of an algorithm can yield substantial gains in CRU efficiency and workspace conservation.

In case you're dubious, the following timing comparisons should dispel any doubts. The target string is the variable *INSTRUCTIONS* (3683 characters) from workspace 1 *FILEAID* on the APL*PLUS System. The times are in CPU milliseconds; the origin is 1.

B	ρ*R*						
SUBSTRING	HITS	*SS*1	*SS*2	*SS*3	*SS*4	*SS*5	*SS*6
''	3683	19	19	2	2	2	2
' '	783	85	60	8	8	8	8
'ω'	0	82	56	2	2	2	2
' '	342	164	105	27	28	14	14
'*AN*'	33	163	104	10	9	7	7
'αω'	0	162	103	3	4	3	4
' *THE*'	20	262	204	55	9	14	8
'*THE* '	16	261	203	14	9	9	9
'*MATRIX*'	5	357	300	15	5	10	9
'*RATMIX*'	0	356	301	23	5	9	5
' *PERMITS*'	15	456	403	113	17	18	13
'*PERMITS* '	15	455	401	16	17	12	13
'*NOTFOUND*'	0	454	399	22	19	7	7
'α∊ι○∪ρ~ω'	0	455	399	3	4	4	4

(Executed 2/25/77 at 9:19 P.M. EST on APLPLUSC/470-10011 with 8 users)

March 1977

TIMING ALGORITHMS

APL is a lovely notation for expressing algorithms. Its rich set of primitives provides a wealth of identities -- alternative ways of formulating a problem. In a theoretical or pedagogical environment, these formal equivalences allow us to write expressions and derivations in the manner that seems most appropriate. This use of APL as a mathematical notation preceded its implementation as a computer language by several years.

When APL was implemented in 1966, one could, by simply assigning values to variables and entering an expression, have a computer evaluate it and return the result. Henceforth, rather than being an abstract notation, APL became widely known as a computer language. In such a practical environment, users soon discovered that alternative formulations of the same problem often took widely disparate amounts of computer time to execute. Since computer time costs money, users had a definite interest in knowing the relative costs of alternative algorithms. And so were born timing programs.

A timing program simply measures the time required to perform an algorithm. The elapsed time, or the CPU time, or both can be measured. However, the elapsed time is usually ignored for two reasons: (1) it varies due to unpredictable external influences (such as what else the computer is doing at the moment), and (2) its cost is typically insignificant as compared to CPU time. On most large-scale computers today, the cost of connect time is typically less than one percent of equivalent CPU time.

Thus, it is CPU time we wish to measure, and the system function ⎕*AI* (accounting information) is our tool. Its second element is the CPU time accumulated since sign-on, which is measured in thousandths of a second (milliseconds). If we wish to measure the CPU time used to perform an algorithm, we first record the time, then execute the algorithm, then note the time again and subtract.

```
      V←?40ρ1000 ◇ M←? 40 40 ρ100
      ⎕AI[2]
27
      R←V⌹M
      ⎕AI[2]
164
      164-27
137
```

Thus, it took less than one-seventh of a second to solve a set of 40 linear equations with 40 unknowns.

Alternatively, to avoid reentering desk calculator mode twice (which itself takes a small amount of computer time) we could perform:

```
      CPU←⎕AI[2] ⋄ R←V⌹M ⋄ ⎕AI[2]-CPU
131
```

Two problems are inherent in all measurements: (1) the accuracy of the measuring device, and (2) the impact of the measuring device on the process it is measuring. Observe that if we repeat the test above, the time may differ slightly.

```
      CPU←⎕AI[2] ⋄ R←V⌹M ⋄ ⎕AI[2]-CPU
133
```

This phenomenon is normal and reflects imprecise timing resolution within the computer. We can improve the statistical reliability of our measurement by performing the algorithm several times, and dividing the total time by the number of iterations.

```
    ∇ CPU←FOR N;TIMES
[1]    CPU←⎕AI ⋄ TIMES←1
[2]   A̲:→(TIMES>N)⍴B̲
[3]    R←V⌹M
[4]    TIMES←TIMES+1 ⋄ →A̲
[5]   B̲:CPU←(⎕AI-CPU)[2]÷N
    ∇

      FOR 10
131.4
      FOR 10
131.2
```

If the algorithm takes a very small amount of computer time, as in the following example,

```
      CPU←⎕AI[2] ⋄ R←99+⍳1300 ⋄ ⎕AI[2]-CPU
2

      CPU←⎕AI[2] ⋄ R←99+⍳1300 ⋄ ⎕AI[2]-CPU
1
```

the coarse resolution of the timer makes it impossible to obtain reliable measurements. Thus it is essential to repeat the algorithm many times just to accumulate a measurable amount of time.

```
      ∇FOR[3] R←99+⍳1300∇
      FOR 100
1.84

      ∇FOR[3] R←99+⍳1300∇
      FOR 100
1.84
```

While *FOR* improves the accuracy of our test, it imposes a measurable overhead of its own, that of the testing and looping time. If we remove line [3] and rerun *FOR*,

```
      ∇FOR[~3]∇
      FOR 100
0.32
```

we discover that the timing algorithm itself uses one-third of a millisecond per iteration. Thus, we should subtract this overhead time from the result. This technique is used in workspace 303 *RASMARK*.

A better and more general approach is illustrated in the following algorithm.

```
    ∇ R̲←A̲ TIMES B̲;C̲;D̲;RUN
[1]   ⍝CPU FOR A̲ ITERATIONS OF EXPRESSION B̲
[2]    C̲←'⋄',B̲ ⋄ C̲←1↓(A̲×⍴C̲)⍴C̲
[3]    C̲←⎕DEF '∇RUN',⎕TCNL,'[1]',C̲,'∇'
[4]    R̲←⎕AI ⋄ RUN ⋄ R̲←⎕AI-R̲
[5]    C̲←⎕DEFL 'RUN[1]',1↓A̲⍴'⋄'
[6]    D̲←⎕AI ⋄ RUN ⋄ D̲←⎕AI-D̲
[7]    R̲←(R̲-D̲)[2],A̲
    ∇
```

TIMES accurately isolates the time required to execute the given algorithm. For instance,

```
      ⎕←T←10 TIMES 'R←V⌹M'
1306 10
      ÷/T ⍝ PER ITERATION
130.6
      ÷\100 TIMES 'R←99+⍳1300'
147 1.47
```

Line [2] replicates the expression A̲ times, inserting diamonds between each instance. This vector is converted into a function, and its execution time measured on line [4]. Lines [5] and [6] measure the overhead cost of the diamond statement separator, and the last line subtracts this overhead from the total time and catenates the number of iterations.

TIMES will help you evaluate alternative formulations of a problem. Here is an example of a typical test (finding the unique values in a numeric vector):

```
      ALG1←'R←((A⍳A)=⍳⍴A)/A'
      ALG2←'R←A[⍋A] ⋄ R←(1,(1↓R)≠¯1↓R)/R'

      A←?7⍴20
      ÷\50 TIMES ALG1 ⋄ ÷\50 TIMES ALG2
20 0.4
38 0.76

      A←?70⍴200
      ÷\50 TIMES ALG1 ⋄ ÷\50 TIMES ALG2
115 2.3
115 2.3

      A←?700⍴2000
      ÷\5 TIMES ALG1 ⋄ ÷\5 TIMES ALG2
810 162
115 23
```

The first algorithm is faster for small vectors, but the second is substantially faster for large vectors.

May 1977

USING BOOLEANS TO CONTROL COMPUTATION

One of APL's unique contributions to computing languages is its ability to generate and manipulate Boolean arrays (those containing only 1's and 0's). Although binary (or bit) datatype is available in many other languages, it is often clumsy to use, requiring conversion and indirect manipulation. APL, on the other hand, handles Boolean data directly and efficiently; the data is stored compactly (one value per bit) and processed rapidly.

Boolean values are most often generated by one of the six relational functions (< ≤ = ≥ > ≠) or by epsilon (∊). Five other functions perform logical calculations: and (∧), or (∨), nand (⍲), nor (⍱), not (~). Together, these provide all the nontrivial functions of logical calculus. For instance, exclusive-or is ≠ and logical implication is ≤.

But APL's strength with Boolean values lies in its ability to use them with every primitive function that allows numeric arguments. Some functions, such as compression (/) and expansion (\), are designed specifically for Boolean arguments; these allow us to select from, and insert values into, arrays. Other functions for which Boolean arguments are most useful are rotate (⌽), grade up (⍋), grade down (⍒), decode (⊥), and several scalar dyadic functions (+ - × ÷ | !). The reduction and scan operators also gain new significance with Boolean arguments.

Most uses of Boolean values deal with representing logical true-false conditions. For instance,

```
      V←?14⍴14 ⋄ V,[0.5] V>8
  8   7 11   5   9 12   8   3   6   1 14   5   7 13
  0   0  1   0   1  1   0   0   0   0  1   0   0  1
```

V>8 generates a Boolean vector (or bit vector) that asserts whether the corresponding element of *V* is greater than 8; the result contains a 1 if the assertion is true and a 0 if it is false. We can use this result in a variety of ways:

```
      +/V>8       How many greater than 8?
5
      ∧/V>8       All greater than 8?
0
      ∨/V>8       Any greater than 8?
1
      <\V>8       The first greater than 8.
0 0 1 0 0 0 0 0 0 0 0 0 0 0

      (V>8)⍳1     What is its index?
3

      (V>8)/V     All elements greater
11 9 12 14 13     than 8.

      (V>8)/⍳⍴V   Indices of all ele-
3 5 6 11 14       ments greater than 8.
```

The entry *V*[⍒*V*>8] moves all elements greater than 8 to the front of the vector:

```
      V[⍒V>8]
11 9 12 14 13 8 7 5 8 3 6 1 5 7
```

Boolean values can also represent values in binary representation. The following expression gives the base-2 representation of 165 in 8 bits:

```
      2 2 2 2 2 2 2 2 ⊤165
1 0 1 0 0 1 0 1
```

We can convert back to the base-10 value:

```
      2⊥ 1 0 1 0 0 1 0 1
165
```

With some imagination, we can also use Boolean values to control other calculations. A common operation is multiplication by a bit vector. For instance, assume we have a matrix *REGS* of sales region names and an associated vector *NACC* indicating how many new accounts each region has generated. We'd like to produce a new-account report.

```
      DISP←NACC>0      Show only regions
                       with new accounts.
      NA←NACC×NACC>1   Show the number only
                       if it's more than 1.

      DISP⌿REGS,'P⎕ (⎕Q⎕)⎕BLI6' ⎕FMT NA
NEW ENGLAND          (3)
MIDDLE ATLANTIC      (10)
EAST NORTH CENTRAL
EAST SOUTH CENTRAL   (5)
WEST SOUTH CENTRAL
PACIFIC              (2)

      (~DISP)⌿REGS
SOUTH ATLANTIC
WEST NORTH CENTRAL
MOUNTAIN
```

By using logical multiplication, we "zeroed out" *NA* for regions with only one new account, and used the *B* modifier to suppress display.

Another application is to return 0 rather than 1 for 0÷0:

```
      A← 6 7 0 5 3 0 5 0 0 7
      B← 3 4 0 2 3 0 4 5 8 2
      A÷B ⋄ A÷B*B≠0
2 1.75 1 2.5 1 1 1.25 0 0 3.5
2 1.75 0 2.5 1 0 1.25 0 0 3.5
```

Here we used exponentiation by a logical vector to change 0's into 1's because 0÷1 will produce the desired 0. We could also have used *A*÷*B*+*B*=0 in this case.

The following table (the first two rows were illustrated above) shows the most useful manipulations and what they produce when *BOOLEAN* is 0 or 1.

Operation	BOOLEAN←0	BOOLEAN←1
A×BOOLEAN	0	A
A*BOOLEAN	1	A
BOOLEAN!A	1	A
BOOLEAN|A	A	0 (#)
A×~BOOLEAN	A	0
A*~BOOLEAN	A	1

(#) Only for integer *A*.

Often when dealing with arrays, we wish to process part of the array while leaving the rest untouched. For instance, add 3 percent to only those bills over 30 days old:

```
      BILLS←BILLS×1.03*DAYSOLD>30
```

Other times, we may wish to alter the order of the arguments of the computation itself based on a conditional array (e.g., *A-B* or *B-A*). For functions with an inverse, we may wish to conditionally alter the function used in a computation (e.g., *A+B* or *A-B*). If a function has an inverse, we can use ¯1 and one of ×*|! on the conditional array to give us the proper answer. Or, as shown above, we can use the conditional array to control whether the operation is done or not. In the table below, *C* is the conditional array. *C* controls the function used in the first two lines, the order of the function's arguments in the next two lines, and whether or not the function is performed in the last three lines.

If *C*=0 Return	If *C*=1 Return	By Using This Expression
A+B	A-B	A+B ×¯1*C
A×B	A÷B	A×B *¯1*C
A-B	B-A	(A-B)×¯1*C
A÷B	B÷A	(A÷B)*¯1*C
A	A+B	A+B×C
A	A×B	A×B*C
A	A*B	A*B*C

A more general way to interchange scalar function arguments conditionally is by reducing a two-row array that has been rotated by a conditional vector. For example, the following performs *A*B* if *C*=0 and *B*A* if *C*=1:

```
      A,[1] B,[0.5] C
 5  1  7  1  9  2  5  1  2  2  7  3  3
 1  6  1  8  1  4  2 27  5  6  2  4  4
 0  1  0  1  0  1  0  1  0  1  0  1  0
      (1,ρA)ρ*⌿C⊖A,[0.5] B
 5  6  7  8  9 16 25 27 32 36 49 64 81
```

Booleans can also be used to select one of two values via subscription. (Note that if the index origin is 0, the Boolean values can serve directly as indices, obviating the need for the +1.)

```
      B← 0 1 0 1 1 1 0 0 1 1 0 0
      7 11[B+1]     (or 7+B×4 or 7+B\4)
7 11 7 11 11 11 7 7 11 11 7 7
```

This capability allows simple plotting:

```
      ⎕←FREQ←8?42
10 24 15 21 35 31 5 18
      ' ⎕'[1+FREQ∘.≥ι⌈/FREQ]
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Another way to use Booleans is to control the merge order of two vectors:

```
      VOWELS←'AEIOUY'
      CONS←'FCTSL'
      MERGE←1 0 1 0 1 0 0 0 1 1 0
      (ρVOWELS)=+/~MERGE
1
      (ρCONS)=+/MERGE
1
      (VOWELS,CONS)[⍋⍋MERGE]
FACETIOUSLY
```

MERGE controls the selection of data from *VOWELS* (*MERGE*=0) or *CONS* (*MERGE*=1).

Booleans can be used to control computations in many other ways. Bob Smith, known as the "Boolean Bomber" at Scientific Time Sharing Corporation, has done significant work in this area. His paper, "Boolean Functions and Techniques" is available from STSC as Working Memorandum No. 106.

July 1977

APL POTPOURRI

(Note: All examples are in origin 1.)

Did you know that ... ?

''⍴*A* selects the first element of *A* as a scalar. What happens if *A* is empty?

3 5↑*SCALAR* is acceptable, and creates a matrix with all zeros or blanks except for position [1;1] which is the same value as *SCALAR*.

(*N*⍴0)↑*SCALAR* has the same effect as (*N*⍴1)⍴*SCALAR* or (*N*⍴1)↑*SCALAR* -- it creates a singleton of rank *N*.

A scalar is allowed as the right argument to compression and expansion. In each case, the scalar is first replicated to the appropriate length. Thus, *B*/*SCALAR* is identical to (+/*B*)⍴*SCALAR*.

A←1/*A* changes *A* into a one-element vector if *A* was a scalar, but leaves all other arrays unchanged.

0=1↑0⍴*A* returns 1 if *A* is numeric, 0 if *A* is character.

∧/,*B*=*B*=1 returns 1 if *B* is Boolean valued, 0 otherwise. ∧/,*B*∊0 1 can also be used but is somewhat slower.

You can demote a numeric array *R* to its most compact internal representation with the following expression:

```
  A←⍴R ⋄ L←(×/A)⍴0 ⋄ L[]←,R ⋄ R←A⍴L
```

Note particularly the third statement, a syntactic construction not often seen in APL programs.

Applying reversal (⌽[*K*]*A*) or rotation (*B*⌽[*K*]*A*) along a nonexistent coordinate (that is, where ~*K*∊⍳⍴⍴*A*) is acceptable and has no effect. This can be useful. For instance, *V*[⌽[*K*]⍋*V*] sorts *V* in ascending order if *K* is 1, and in descending order otherwise.

The expression +/[2] 3 0 4 ⍴5 returns a 3 by 4 matrix of zeros, thereby actually increasing the storage requirements. Can you guess what ⌊/ 2 3 0 ⍴6 returns, and why?

Relational scans (<\ ≤\ =\ ≥\ >\ ≠\) do not always return Boolean values. For example,

```
  >\ 0.5 0.5 0.5
0.5 0 1
```

Any outer product can be rewritten as an inner product. For instance, *A*∘.+*B* can be restated as (((⍴*A*),1)⍴*A*)o.+(1,⍴*B*)⍴*B* where o can be any scalar dyadic function.

Any reduction on numeric data can likewise be written as an inner product. For instance, ×/[*K*]*A* can be restated as 0×.+(⍋⍋*K*≠⍳⍴⍴*A*)⍉*A*.

Base value (⊥) can be written as a +.× matrix product. Thus *A*⊥*B* (without scalar or unitary coordinate extension on the left argument) can be restated as (⌽×\(⍴*A*)↑1,⌽*A*)+.×*B*.

A+*A* is faster than *A*×2, and *A*×*A* is faster than *A*⋆2. Generally the simplest function is fastest; but the speed difference is small, so use the most natural construct.

The expression (1⋆⍴*A*)⍉*A* selects the major diagonal of any array.

'*I*0' or '*A*0' or '*G*⍞⍞' can be used with ⎕*FMT* to suppress display of unwanted columns. For example:

```
     'I0,I2,A0,I4,3G⍞⍞,F8.3' ⎕FMT 4 8⍴⍳32
 2    4    8.000
10   12   16.000
18   20   24.000
26   28   32.000
```

Many experienced APL programmers cannot name all APL primitive scalar dyadic functions. Can you? (Hint: + is one of them, and there are twenty more.)

September 1977

DATA SELECTION

Selecting data from an array is one of the most common operations in APL. There are three primary facilities for APL data selection:

- positional, using subscription (*A*[*B*])
- conditional, using compression (*B*/[*K*]*A*)
- bounded, using take (*B*↑*A*) or drop (*B*↓*A*)

There are also several specialized techniques for frontal and diagonal selection using reshape (*B*⍴*A*) or transpose (*B*⍉*A*). Besides selecting data, some of these functions can replicate data (subscription, compression of a scalar, and reshape), extend data (take), or reorganize data (subscription, reshape, and transpose). This article discusses these techniques as applied to the matrix *A* in origin 1:

```
      ⎕IO←1 ◇ ⎕←A←(10×⍳5)∘.+⍳9
 11 12 13 14 15 16 17 18 19
 21 22 23 24 25 26 27 28 29
 31 32 33 34 35 36 37 38 39
 41 42 43 44 45 46 47 48 49
 51 52 53 54 55 56 57 58 59
```

Subscription is the most generalized selection function. It can be used to select, reorder, and replicate data.

```
      A[2 5 1 ; 9 1 5 7 5 5]
 29 21 25 27 25 25
 59 51 55 57 55 55
 19 11 15 17 15 15
```

Because of its flexibility, subscription is typically the slowest of all selection functions applied to a matrix. However, it alone can reorder or selectively replicate data, and it can be used to the left of assignment (←). In the absence of these requirements, subscription is best used only to select single rows or columns, or small subarrays. Subscription is somewhat faster on vectors than it is on matrices.

Compression is less flexible than subscription; it can select only along a single coordinate, and it cannot replicate or reorder data. However, it can select non-contiguous rows or columns in the same order they occur in the array.

```
      1 1 0 0 1 ⌿A
 11 12 13 14 15 16 17 18 19
 21 22 23 24 25 26 27 28 29
 51 52 53 54 55 56 57 58 59

      1 0 0 0 1 0 1 0 1 /A
 11 15 17 19
 21 25 27 29
 31 35 37 39
 41 45 47 49
 51 55 57 59
```

Because of its more limited capability, compression is typically faster than

subscription. Of course, it is ideal when selecting on the basis of a Boolean vector from epsilon (∊) or one of the relational (< ≤ = ≥ > ≠) or logical (∧ ∨ ⍲ ⍱ ~) functions. As noted, compression can select along only one coordinate. Thus the matrix must be compressed twice in order to select both rows and columns.

```
      0 1 1 0 0 ⌿ 0 0 0 1 1 1 1 0 0 /A
 24 25 26 27
 34 35 36 37
```

If indices are already provided for one coordinate, it is usually faster to use *B*/⍳⍴*B* and subscription. For instance,

```
      A[B/⍳⍴B; 5 4 7 6]
```

should be used rather than

```
      B⌿A[; 5 4 7 6]
```

Take and drop are the most restrictive selection functions. They allow only contiguous blocks of data to be selected, and they cannot replicate or reorder. They are quite fast, however, at selecting large blocks of data. One application of either function is sufficient to select any corner subarray.

```
      3 5↑A ⍝¯2 ¯4↓A           3 ¯5↑A ⍝¯2 4↓A
 11 12 13 14 15          15 16 17 18 19
 21 22 23 24 25          25 26 27 28 29
 31 32 33 34 35          35 36 37 38 39

     ¯3 5↑A ⍝ 2 ¯4↓A          ¯3 ¯5↑A ⍝ 2 4↓A
 31 32 33 34 35          35 36 37 38 39
 41 42 43 44 45          45 46 47 48 49
 51 52 53 54 55          55 56 57 58 59
```

Any contiguous block of data can be selected by two applications of either function. The subarray

```
      A[2 3 ; 4 5 6 7]
 24 25 26 27
 34 35 36 37
```

can be selected by each of 16 expressions:

```
  ¯2 ¯4 ↑  3  7 ↑A      ¯2  4 ↑  3 ¯6 ↑A
  ¯2 ¯4 ↑ ¯2 ¯2 ↓A      ¯2  4 ↑ ¯2  3 ↓A
   1  3 ↓  3  7 ↑A       1 ¯2 ↓  3 ¯6 ↑A
   1  3 ↓ ¯2 ¯2 ↓A       1 ¯2 ↓ ¯2  3 ↓A

   2 ¯4 ↑ ¯4  7 ↑A       2  4 ↑ ¯4 ¯6 ↑A
   2 ¯4 ↑  1 ¯2 ↓A       2  4 ↑  1  3 ↓A
  ¯2  3 ↓ ¯4  7 ↑A      ¯2 ¯2 ↓ ¯4 ¯6 ↑A
  ¯2  3 ↓  1 ¯2 ↓A      ¯2 ¯2 ↓  1  3 ↓A
```

Unless extension (padding with zeros or blanks) is needed, convenience should determine whether to use take or drop.

Reshape and transpose are normally considered structural functions, but are often used for specialized selection. Reshape can select data from the front of an array. For instance, the following expression selects the first element as a scalar:

```
      ''⍴A
11
```

The next expression selects the first row as a vector:

```
      9⍴A
11 12 13 14 15 16 17 18 19
```

Transpose can select the major diagonal, or the other diagonals if used with reversal.

```
      1 1 ⍉A
11 22 33 44 55
      1 1 ⍉⌽A
19 28 37 46 55
      1 1 ⍉⊖A
51 42 33 24 15
      1 1 ⍉⌽⊖A
59 48 37 26 15
```

Both reshape and transpose are very fast.

In general, the speed of all selection functions depends on the amount of data being selected, not on the size of the original array. As more data is selected, more CRUs will be consumed. By avoiding superfluous data selection and by using the most specialized function appropriate to your needs, you can increase the efficiency of your programs.

November 1977

WHY NOT TO LOOP

Most APL systems are implemented as interpreters. This allows users great flexibility in creating and modifying programs. There is no need for compilation, and errors can be diagnosed and corrected during execution.

The major penalty of an interpreter is the cost of syntax analysis and dynamic data checking. Fortunately, APL functions tend to be quite short relative to programs in other languages, and therefore the interpretive overhead is minimized.

On the other hand, when looping is performed in APL, the interpretive overhead skyrockets because the same code must be reanalyzed for each iteration. The general rule, therefore, is to try to minimize looping in APL.

As anyone familiar with APL will attest, most programs can be written with no loops at all. APL's rich set of primitive functions and operators allows "closed form" (nonlooping) solutions to many complex problems. The control structures that must be written explicitly in other languages (e.g., DO loops) are performed implicitly in APL primitives (e.g., scan). But difficulties can arise when people who have been schooled in languages like FORTRAN, BASIC, and COBOL try to move their looping programs intact into APL.

For example, the following portion of a FORTRAN program calculates cash flow for a series of 360 investments using variable interest rates. DEP is N+1 deposits, the first being the initial amount; RATE is N interest rates between 0 and 1; CF is the result.

```
   DIMENSION CF(361)
   DIMENSION DEP(361)
   DIMENSION RATE(360)
     .
     .
     .
   CF(1)=DEP(1)
   DO 20 I=1,360
20 CF(I+1)=DEP(I+1)+CF(I)*(RATE(I)+1.)
```

This segment can be translated almost directly into the following APL program:

```
    ∇ CF←RATE CASHFLOW DEP;I;N
[1]   CF←DEP
[2]   N←⍴RATE
[3]   →N↓0 ◇ I←1
[4]  L20:CF[I+1]←DEP[I+1]+CF[I]×RATE[I]+1
[5]   →L20×N≥I←I+1
    ∇
```

and will run successfully, albeit slowly:

```
      (⍴DEP),⍴RATE ⍝ 30-YEAR MONTHLY LOAN
361 360
      T←⎕AI ◇ CF←RATE CASHFLOW DEP ◇ ⎕AI-T
0 2070 483 0  (0,MILLICRUS,MILLISECONDS,0)
```

More than 2 computer resource units are required to run *CASHFLOW* -- not a very cost-effective solution!

Rather than entirely rethinking the problem, some users will take great pains to optimize the existing algorithm. This approach is fine if no alternative algorithms are considered, but it often results in a more obscure program:

```
    ∇ CF←R CASHFLO2 D;I;J;N;T
[1]   T←D[I←1] ⋄ D←1↓CF←D
[2]   R←R+1
[3]   →N←((ρR)ρL),0
[4]   L:T←CF[J←I+1]←D[I]+T×R[I] ⋄ →N[I←J]
    ∇
```

The techniques used in *CASHFLO2* all serve to reduce interpretive overhead in the loop on line [4]. Interest rates *R* and branch targets *N* are calculated in advance, and subscript calculations are simplified. (Note that the use of one-letter variable names does not affect the cost of running the program.)

```
      T←⎕AI ⋄ CF←RATE CASHFLO2 DEP ⋄ ⎕AI-T
0 1301 330 0
```

We've reduced the cost by one third -- a significant savings, but still not enough to be cost effective.

By far the most effective approach is to reformulate the problem into an APL solution, rather than a warmed-over FORTRAN solution. Consider the following:

```
    ∇ CF←RATE FASTFLOW DEP
[1]   CF←1,×\RATE+1 ⋄ CF←CF×+\DEP÷CF
    ∇
```

CF is set first to the cumulative present-value discount factors for future deposits. The deposits are divided by those factors, summed, and then multiplied by the factors to produce the result. A good exercise is to derive this algorithm from the looping algorithm; the key is that multiplication and division are both distributive with respect to addition (division is only right distributive).

FASTFLOW is indeed an elegant and inexpensive solution to the problem:

```
      T←⎕AI ⋄ CF←RATE FASTFLOW DEP ⋄ ⎕AI-T
0 26 7 0
```

It performs 50 times faster than even the optimized looping solution, and illustrates the benefits that can be derived from writing nonlooping APL code.

January 1978

CHARACTER SEARCHES

With the recent introduction of the system function ⎕*SS* for character string searching, and the major speedups to character ∧.= and ∨.≠, a variety of new techniques are now practical. Although these techniques were usable before the enhancements, they were often infeasible because of *WS FULL* conditions or the large number of computer resource units needed to accomplish them. Frequently, obscure or clumsy code was written to circumvent the previous snailish pace of ∧.=.

For example, a typical table lookup

```
   MATRIX∧.=VECTOR
```

was occasionally coded as

```
   ∧/MATRIX=(ρMATRIX)ρVECTOR
```

to reduce cost (at the risk of *WS FULL*). Now, the first expression is many times faster.

The improvements to ∧.= also dramatically altered the relative speeds of the six different string-searching algorithms presented in "String Searching" beginning on page 21. In particular, *SS2*, *SS3*, and *SS4* all use ∧.= for part or all of the search. Below is a table comparing their performance on 25 February 1977 with their current performance and with that of their ⎕*SS*-equivalent (*R*←(*A* ⎕*SS B*)/ιρ*A*). The times are in milliseconds.

B	ρ*R*	25 FEB 1977			24 FEB 1978			
SUBSTRING	HITS	*SS2*	*SS3*	*SS4*	*SS2*	*SS3*	*SS4*	⎕*SS*
''	3683	19	2	2	8	2	2	6
' '	783	60	8	8	11	8	8	8
'ω'	0	56	2	2	6	2	2	2
' '	342	105	27	28	10	18	19	7
'*AN*'	33	104	10	9	7	8	8	2
'αω'	0	103	3	4	6	3	4	2
' *THE*'	20	204	55	9	8	24	7	5
'*THE* '	16	203	14	9	7	9	7	3
'*MATRIX*'	5	300	15	5	8	9	6	2
'*RATMIX*'	0	301	23	5	8	12	6	2
' *PERMITS*'	15	403	113	17	10	41	10	5
'*PERMITS* '	15	401	16	17	8	9	9	2
'*NOTFOUND*'	0	399	22	19	8	10	10	2
'αειOuρ~ω'	0	399	3	4	8	3	5	1

(Executed 24 Feb 78 at 6:46 PM EST
on APLPLUSC/470-10011 with 34 users)

Although they still suffer a propensity for *WS FULL*s with large arguments, it's nice to know their cost has dropped so dramatically!

Pleasantly, the discussion of string-search techniques is now moot because we have ⎕*SS*. In fact, even if ⎕*SS* weren't available, the following simple function would perform the same task:

```
    ∇ R←A ∆SS B;C
[1]   R←ρA ◇ C←ρ,B
[2]   R←R↑(-C)↓B∧.=(C,R+×R)ρA
    ∇
```

The function ∆SS takes roughly twice as long as ⎕SS to perform a given search. Moreover, ⎕SS has the advantage of requiring no additional or intermediate workspace storage.

Not only does ⎕SS provide cost and storage advantages, but it also allows novel or simpler approaches to problems. Below are a few examples.

Determine if function DSPELL is currently suspended or pendent; that is, whether it's in the state indicator.

```
        FN←'DSPELL'
OLD     ∨/(((1↑ρ⎕SI),1+ρ,FN)↑⎕SI)∧.=FN,'['
        ∨/(((1↑ρ⎕SI),7)↑⎕SI)∧.='DSPELL['
NEW     ∨/(,' ',⎕SI)⎕SS ' ',FN,'['
        ∨/(,' ',⎕SI)⎕SS ' DSPELL['
```

Mark the line-ending carriage returns in a function.

```
        VR←⎕VR 'FNNAME' ◇ VR[⎕IO-6-ρVR]←'['
OLD     (VR=⎕TCNL)∧1⌽VR='['
NEW     VR ⎕SS ⎕TCNL,'['
```

And finally, here's a function that locates a string in any row of a matrix:

```
    ∇ R←M ROWSS V
[1]   R←∨/(0,1-ρ,V)↓(ρM)ρ(,M) ⎕SS V
    ∇
```

⎕SS locates all occurrences in the ravelled matrix, and its result is reshaped to the shape of the matrix. Then, those columns that might identify matches spanning more than one row are dropped. Finally, the ∨/ singles out the proper rows. ROWSS will locate matches in a character matrix:

```
        ρSTATES
50 14
        (STATES ROWSS 'NIA')⌿STATES
CALIFORNIA
PENNSYLVANIA
VIRGINIA
WEST VIRGINIA
        (STATES ROWSS 'LIN')⌿STATES
ILLINOIS
NORTH CAROLINA
SOUTH CAROLINA
```

March 1978

BACK TO BASICS

One of the beauties of APL is the ability to get useful results while using only a small subset of the language. Gradually, through curiosity and experimentation, users discover more advanced features which they can apply to their problems. In the interest of hastening that discovery, I'd like to introduce you to a powerful function in APL called base value. I'll discuss its performance on scalars and vectors only, leaving you to experiment with matrices if you wish.

Base value, also called decode, is represented by the ⊥ symbol (shift B). It is dyadic and accepts only numeric arguments. Let's see what happens in a simple case.

```
      10⊥ 8 2 9 5
8295
```

The effect of "putting numbers together" is actually the result of evaluating the polynomial expression

$$8x^3 + 2x^2 + 9x + 5$$

for x=10, or, for you APL'ers out there:

```
      X←10
      (8×X*3)+(2×X*2)+(9×X)+5
8295
```

or

```
      5+X×9+X×2+X×8
8295
```

Notice that the coefficients are on the right and the independent variable is on the left. Thus, base value evaluates polynomials. This may seem esoteric, but it actually proves quite useful. For example, you can use it to combine month, day, and year into one number,

```
      100⊥ 6 5 77 ⍝ (10000×6)+(100×5)+1×77
60577
```

or to determine the number of seconds in 2 hours, 10 minutes, and 38 seconds.

```
      60⊥ 2 10 38  ⍝ (3600×2)+(60×10)+1×38
7838
```

Note that the following three expressions are equal:

```
      2+3×4 ◇ 3⊥ 4 2 ◇ 4⊥ 3 2
14
14
14
```

From what you've seen so far, can you explain why 1⊥VEC is the same as +/VEC, and why 0⊥VEC is the same as ¯1↑VEC?

For scalars and vectors, A⊥B is equivalent to +/W×B (or W+.×B), where W is a vector of weights derived from A. As we've seen above, if the left argument is a scalar, then (assuming ⎕*IO*←1) *W* is calculated as follows:

```
      W← A*(⍴B)-⍳⍴B
         60* 3  -1 2 3
         60* 2 1 0
         3600 60 1
```

More generally, if the left argument is a vector,

```
      A← 0 7 24 60 60 1000
      B← 1 2 3  4  5  6
      A⊥B
788645006
```

base value is evaluating a <u>mixed</u> <u>radix</u> polynomial. In the example above we're calculating the number of milliseconds in 1 week, 2 days, 3 hours, 4 minutes, 5 seconds, and 6 milliseconds. In this case the left argument represents:

```
         0 (placeholder)
         7 days per week
        24 hours per day
        60 minutes per hour
        60 seconds per minute
      1000 milliseconds per second
```

The weighting vector is computed by performing a multiplication scan from right-to-left on all but the first element of the left argument, and catenating a 1 to that result:

```
W←⌽1,×\⌽1↓A
W←⌽1,×\⌽1↓0 7 24 60 60 1000
W←⌽1,×\⌽7 24 60 60 1000
W←⌽1,×\1000 60 60 24 7
W←⌽1,1000 60000 3600000 86400000 604800000
W←⌽1 1000 60000 3600000 86400000 604800000
W←604800000 86400000 3600000 60000 1000 1
  week      day       hour     min.   sec. ms
  (milliseconds in)
```

Hence we multiply the number of milliseconds in a week by the number of weeks, add it to the number of milliseconds in a day multiplied by the number of days, and so on, to produce the result:

```
      +/W×B
788645006
```

If either argument is scalar, it is extended to the length of the other argument, and the same weighting vector calculation is used. Thus the following expressions are equivalent:

```
      12 12 12 ⊥ 2
314
      12 12 12 ⊥ 2 2 2
314
            12 ⊥ 2 2 2
314


              60 ⊥ 10 0 120
36120
        60 60 60 ⊥ 10 0 120
36120
         0 60 60 ⊥ 10 0 120
36120
      ¯743 60 60 ⊥ 10 0 120
36120
```

As illustrated, the first element of the left argument does not affect the result, but serves only as a placeholder. If either argument is not a scalar or one-element vector, the lengths of the arguments must match.

```
      60 60⊥10 0 120
LENGTH ERROR
      60 60 ⊥ 10 0 120
          ∧
```

This rule is identical to that used by inner product (for example, +.×), although in inner product the first element of the left argument does affect the result.

SOME USEFUL APPLICATIONS

Right justify a character vector *V*:

```
      (1-(V=' ')⍳1)⌽V
```

Actually, this expression will right justify an array of any rank. It uses the "reverse multiplication scan" weighting vector calculation as an and-scan (∧\), and the +/*W*×1 to calculate (1+) the number of contiguous ending spaces.

"Why Not to Loop" beginning on page 37 contains the function *FASTFLOW* to compute a cash flow series for a starting balance (*SB*) and *N* subsequent deposits (*DP*), all compounded at varying interest rates (*IR*).

```
    ∇ CF←RATE FASTFLOW DEP
[1]   CF←1,×\RATE+1 ◇ CF←CF×+\DEP÷CF
    ∇
      SB← 100
      DP← 0 50 100 ¯216 0 123.26
      IR← 0.2 0.25 0.08 0 0.05 0.06
      IR FASTFLOW SB,DP
100 120 200 316 100 105 234.56
```

If you don't need the intermediate values and only wish to know the ending balance, the following expression will suffice:

```
      (0,IR+1)⊥SB,DP
234.56
```

Given an array *YM* containing months packed as *YYMM* or *YYYYMM*, add *A* months and return the dates in the same format:

```
    ∇ R←A MONTHADD YM
[1]   R←1+100⊥10000 12⊤A+12⊥10000 100⊤YM-1
    ∇
      7 MONTHADD 197805 7806 197912
197812 7901 198007
      ¯5 ¯11 ¯16 MONTHADD 197804
197711 197705 197612
      (⍳6) MONTHADD 197806+⍳6
197808 197810 197812 197902 197904 197906
```

This is a lovely illustration of the relationship between decode (⊥) and encode (⊤). It uses both to convert back and forth between decimal and base-12 representation.

May 1978