Xerox Data Systems

701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX®

Introduction To APL (A Programming Language)

by Dennis Taylor
Xerox Data Systems

WHAT IS APL?

There are two ways to succinctly describe APL:
(1) APL is an interpretive, time-sharing, problem-solving language; and
(2) APL is an interactive implementation of Iverson notation.

(1) What does "interpretive" mean?

In an interpretive environment, the system doesn't wait until a program is completed to compile it into object code and execute it. Instead the system interprets each line of input as it is written to produce code which is immediately executed.

Speaking in generalities -- compilers are more efficient than interpretive systems; but the competitive advantage becomes less for interactive requirements. APL produces extremely tight code - and is, in fact, more 'efficient' than many of the compilers offered today.

(2) What is a "problem-solving" language?

A problem-solving language is one which does not require the user to know anything about the computer or its programming languages. The computer is treated as a "black box" -- into which is put a problem (in the user's own job-oriented language) and from which is received an answer (again, in the user's language.)

The attributes of a good problem-solving language are:

QUICKLY TAUGHT
EASY TO USE
MINIMUM TYPING TIME
AUTOMATIC ERROR CHECKING
LIBRARY CAPABILITY

EASY TO LEARN
MACHINE INDEPENDENT
IMMEDIATE RESPONSE
EASY CORRECTION PROCEDURE
VISUAL FIDELITY

(3) What is Iverson notation?

In 1957, Dr. Ken Iverson was an Assistant Professor at Harvard, teaching mathematics and computer science. He began a book on computing applications, but found he needed to develop a new notation to be able to write on topics such as sorting and machine organization. He had a need for a clear and concise way to describe the algorithms he was attempting to define. The resulting notation is today called Iverson notation -- and it is a strongly algorithmic language.

SAMPLE TERMINAL USAGE

APL operates in three modes. In "desk calculator" mode, expressions may be entered for immediate execution. In "function definition" mode, expressions may be combined into programs and stored for future use. "Function editing" mode allows us to edit functions previously stored.

Let's sit down at a terminal and examine the properties and capabilities of APL. The student's problems are indented about one-half inch from the left margin; the APL response is left-justified on the page.

DESK CALCULATOR

```
      2+2
4

      7-6
1

      1.73  × 6
10.38
```

As mentioned earlier, APL is a problem-solving language. This means that a user can enter a problem as simple (or complicated, depending on how you look at it) as 2+2 and get a simple 4 in return. What other system exists where this type of interaction is possible?

Notice that there are no "format" statements, no I/O statements, and there is no need to "type" data as real or integer.

```
      2+3+5
10

      16 × 3 + 4.3
116.8
```

We can even present a number of scalar values to be operated upon by separating them with native APL operators.

```
      4 +8÷2
8
```

Unlike ordinary algebra, APL has no hierarchy among functions. The order of execution of a statement is from right to left; therefore every function takes as its argument the entire expression to the right of it.

```
      1÷3
0.3333333333

      5÷0
DOMAIN ERROR
      5÷0
      ^
```

Notice that accuracy is provided to 10 decimal digits (and is carried internally to 16 digits.) By the way, you had better start getting acquainted with APL error messages; the processor is quite unforgiving.

```
      8+5
13

      6+5
11
```

One of the most significant features of APL is its ability to work with arrays (and matrices) as easily as it does with scalar values. Suppose the ages of your three children were 8, 6 and 3 and you wanted to see how old they would be five years from now. You could create an APL expression for each child; or you could save a little time by entering the ages of the children as an array (or vector) of numbers (represented simply by leaving spaces between the scalar values) -- and produce all three results simultaneously. Here we are mixing a scalar value and a vector in an APL expression.

```
      3+5
8

      8  6  3 + 5
13  11  8
```

Now suppose that in the grocery store you buy 8 units of commodity A, 6 units of B and 3 units of C; and next week you buy 4 more units of A, 9 units of B and 1 unit of C. You can determine how much of each commodity you have by placing the 'plus' sign between the two arrays of quantities. This is how APL reacts when operating upon two vector quantities in parallel.

```
      8  6  3 + 4  9  1
12  15  4
```

By the way, the dimensions of the two vectors must be the same for APL to process their elements in parallel.

```
      8  6  3 + 4  9
LENGTH ERROR
      8  6  3 + 4  9
      ^
```

Two commonly-used mathematical operators are those for raising numbers to powers and taking roots. There exists no standard symbol for exponentiation (other than superscript notation,) so APL uses the star (*). Notice again how we can mix scalar and vector quantities; actually this is done by APL assuming that the scalar is a vector of equal components the same length as the opposing vector.

```
      2*3
8

      2 * 1 2 3 4 5
2 4 8 16 32

      25 16 9 4 1 * 0.5
5 4 3 2 1
```

The MAX (⌈) and MIN (⌊) operators are as useful as the others previously covered. The ⌈ symbol placed between two arguments will generate as a result the larger of the two. The ⌊ symbol generates the smaller of the two arguments. Needless to say, these

```
      5⌈3
5

      7⌈106
106
```

```
      38 ⌈ 15 30 50
38  38  50

      10 12 14 ⌊ 6  8  20
6   8  14
```

operators also apply to the various combinations of scalars and vectors:

SCALAR ← *SCALAR* □ *SCALAR*
VECTOR ← *SCALAR* □ *VECTOR*
VECTOR ← *VECTOR* □ *SCALAR*
VECTOR ← *VECTOR* □ *VECTOR*

```
      6=6
1

      6≤4
0

      6 ≠ 1 3 6 4
1 1 0 1

      2 + (4=4)
3
```

Comparison operations assume their normal functions and use the following symbols: less than (<); less than or equal to (≤); equal to (=); greater than or equal to (≥); greater than (>); and not equal to (≠). "How close is equal?" is of importance, and a tolerance of approximately 1.0E-13 is used (and is termed fuzz.) Fuzz is used with all the comparison operations, which produce 1 for true and 0 for false. Thus the result of a comparison operation can be used in an arithmetic or logical expression.

Other basic mathematical functions include:

```
      2!3
3

      2! 2 3 4 5 6 7
1 3 6 10 15 21

      !4
24

      ?5
1

      ?5
4

      ⍟2
0.6931471806

      ○1
3.141592654

      10 ⍟ 20
1.301029996
```

COMBINATION: A!B gives the number of combinations of B things taken A at a time.

FACTORIAL: !A gives the number of distinct arrangements of A things.

ROLL: ?A selects an integer pseudo-randomly from the first A positive integers.

NATURAL LOGARITHM: ⍟A computes $\log_e A$.

"PI TIMES": ○A computes mathematical value of Pi times the operand A.

LOGARITHMS: A⍟B computes the log of B to the base A

```
      1○1
0.8418709848

      2○1 2
0.5403023059   ¯0.4161468365

      1○○÷6
0.5
```

CIRCULAR FUNCTIONS: are expressed as A○X where: (X in radians)

Sin	X is 1○*X*	Arcsin X is ¯1○*X*
Cos	X is 2○*X*	Arccos X is ¯2○*X*
Tan	X is 3○*X*	Arctan X is ¯3○*X*

```
      A←3+1

      A
4
      CLASS ← A + 6

      CLASS
10
      A
4
      CLASS + A
14

      A←140

      CLASS + A
150
      A
140

      NUMBERS←1 2 3 4 5

      NUMBERS
1 2 3 4 5

      NUMBERS × 2
2 4 6 8 10
```

Thus far, we have made no mention of retaining the results of our evaluations. Assignment of values to variable names avoids re-entry of results. The specification (←) symbol is used instead of the equality symbol (=) to avoid nonsense statements such as *A*=*A* + 1.

When specification is made (*A*←3+1), no response is generated. The value of A may be seen by simply typing its name. Notice that the value of the variable is retained until a re-specification is made.

Notice also that we can specify a variable that contains a vector quantity (or for that matter - a matrix.)

```
      +/1 2 3 4 5
15

      A←1 2 3 4 5

      +/A
15

      +/A+1
20             (Why?)
```

It is sometimes useful to place an operator between the elements of a vector and, once evaluated, "reduce" the vector to a scalar value. Σ is the standard symbol for the summation of the elements of a vector; and π is the symbol for the product of these elements. This is as far as "standard" notation goes!

The APL operator for reduction is the symbol /. The operator is written to the left, and the elements (or vector name) are written to the right of the operator.

```
      ×/A
120

      ⌈/A
5

      ⌊/A
1
```

```
      1 0 1 / 10 20 30
10 30

      0 0 1 0 0 / 1 2 3 4 5
3

      A← 1 2 3 6 2

      (A≥3)/A
3  6

      0 0 0 / 1 5 7

      (the null vector)
```

Another useful concept is that of COMPRESSION, which also uses the symbol /, but differently than does reduction. In compression a logical (0's and 1's) vector is placed on the left side of the operator and a vector on the right. The logical vector must be of a length equal to that of the problem vector (or can be a scalar 0 or 1.)

Notice what happens! We have returned to us only those elements which correspond to a "1" in the logical vector. Compression will be referred to again when we discuss function definition.

```
      CHAR← 10 12 1 7

      ρCHAR
4

      ι6
1 2 3 4 5 6

      A←ι5
      A
1 2 3 4 5
```

Two new operators introduced at this point would be helpful. The Greek character rho (ρ) is the APL symbol for dimension. Asking the dimension of a vector is asking how many elements are in the vector.

The Greek symbol iota (ι) is the APL index generator. Stated simply, ιA generates a vector of positive integers from 1 to A.

We have now progressed far enough to compare a simple FORTRAN program (to average an array of numbers) with its APL counterpart.

APL

```
      X←⍳10

      (+/X)÷⍴X
5.5
```

FORTRAN

```
      DIMENSION  X(N)
      READ 2, N, (X(I),I=1,N)
    2 FORMAT (    )
      SUMX = 0.0
      DO 3  I=1,N
    3 SUMX = SUMX + X(I)
      AVE = SUMX / FLOAT N
      PRINT 4, AVE
    4 FORMAT (    )
      STOP
      END
```

FUNCTION DEFINITION

Let us now examine the procedure by which we can augment the set of standard APL operators with some of our own "functions." The del (∇) character signals function definition and is followed by a function name. The system responds with [1], and waits for your first "program" line. Each successive line is also numbered until the routine is closed by another ∇.

Let us assume that we wanted to define a function AVR to average a vector of numbers.

```
      ∇C←AVR X
[1]   C←(+/X)÷⍴X
[2]   ∇

      AVR 1 2 3
2

      N← ⍳10

      AVR  N
5.5
```

The next program we may wish to write is one for calculating the length of the hypotenuse of a right triangle, given the lengths of the two sides. If the hypotenuse is C, then:

$$C = \sqrt{A^2 + B^2}$$

```
      ∇C←A HYP B
[1]   C←((A*2)+B*2)*0.5∇

      3 HYP 4
5

      3 HYP 5
5.830951895
```

```
     ∇AVE
[1]  'GOOD MORNING'
[2]  'ENTER STRING OF NUMBERS'
[3]  A←⎕
[4]  AVR A∇

     AVE
GOOD MORNING
ENTER STRING OF NUMBERS
⎕:
     27 58 3 5 107
40
```

But APL is an interactive system! Let's allow our student to sit down at the terminal and "interact" with some pre-defined functions. First, a slight revision of the AVR program to average numbers. Notice in [1] that literal character strings can be stored and displayed by simply enclosing them in single quotes.

In step [3] the quad symbol (⎕) is used to denote input from the terminal, and that input in this case is being assigned to the variable A. In step [4] we call our previously-defined program to average numbers; the result will be typed out.

The closing del (∇) at the end of step 4 ends the function definition mode and returns us to desk calculator, or immediate execution, mode. Once the AVE program is executed, we must call it out again to re-use it as no return within the program was provided.

```
     ∇A GEO B
[1]  'GIVEN A RECTANGLE OF '
[2]  'SIZE ';A;' BY ';B
[3]  ' PERIMETER: ';2×A+B
[4]  ' AREA:      ';A×B
[5]  ' DIAGONAL:  ';A HYP B
[6]  ∇

     3 GEO 5
GIVEN A RECTANGLE OF
SIZE 3 BY 5
 PERIMETER: 16
 AREA:      15
 DIAGONAL:  5.830951895
```

Let's take another simple function - one to calculate the perimeter, area, and diagonal of a rectangle. Note here that we are using the previously-defined HYP function. The (;) allows us to catenate dissimilar outputs (character versus numeric values.)

```
     ∇AV
[1]  'GOOD AFTERNOON'
[2]  'ENTER STRING OF NUMBERS'
[3]  A←⎕
[4]  →(A=0)/0
[5]  AVR A
[6]  →3
[7]  ∇
```

If we wish to make AVE a repeating function, we must include in the program return and test-for-exit statements. The Branch symbol (→) is used for the return and precedes either a statement number or a line label. The exit test in this case consists of comparing the input variable with zero and branching to line 0 (exit) if that comparison is true.

```
      AV
GOOD AFTERNOON
ENTER STRING OF NUMBERS
⎕:
      ι5
3
⎕:
      ι10
5.5
⎕:
      0
```

A branch to statement 0 (which is non-existent) exits us from the program. Notice the use of comparison and compression:

If A=0, we evaluate 1/0 or 0 and branch to 0 which exits us from routine.

If A≠0, we evaluate 0/0 or NULL and drop through the test.

```
      ∇R←SORT X
[1]   R←ι0
[2]   R←R,(X=⌊/X)/X
[3]   X←(X≠⌊/X)/X
[4]   →2× 0≠⍴X∇


      STRING←8 2 6 2 4 107

      SORT STRING
2 2 4 6 8 107

      STRING←¯3 ¯8  52

      SORT STRING
¯8  ¯3  52
```

Let's become a little more sophisticated! A sort program is an easy one to write if you have mastered the material up to this point. The basic ascending sort algorithm we will use is this:

(1) Determine the lowest value of the vector, and create a new vector containing only those elements of the original that equal those lowest elements.

(2) Remove the transferred value(s) from the original vector.

(3) Repeat steps (1) and (2), only this time catenate the next-lowest values to the new vector. When no values remain in the original vector, the sort is complete.

```
      ARRAY←8 10 75 16 12

      ARRAY[3]
75
      ARRAY [1 3 5]
8 75 12
      ARRAY [5 4 3 2 1]
12 16 75 10 8
      ARRAY[⍋ARRAY]
8 10 12 16 75
      ARRAY[⍒ARRAY]
75 16 12 10 8
```

The bracket symbols [] are used to enclose indices (think of array subscripts) and when appended to an array name, generate the values of those indexed elements.

Now that we have gone to the trouble of writing a sort program, we can see that sorting (ascending or descending) is really a native operation of APL. The grade-up (⍋) symbol and grade-down symbol (⍒), when used in index notation generate the values of the vector is either sorted sequence.

Sorting non-numeric data isn't quite as easy -- due to the fact that the order of special characters is not well defined.

```
      ORDER←'ABCDEFGHIJKLMNOPQRSTUVWXYZ ()'
```

Let us assume the order of all characters to be used and state that order in the variable *ORDER*.

```
      TEXT←'XDS (FORMERLY SDS)'
```

We define the characters to be sorted in the variable *TEXT*. The operation *ORDER*ι*TEXT* can now be used to determine for each element of *TEXT* its position in *ORDER*.

```
      J←ORDERιTEXT

      J
24 4 19 27 28 6 15 18 13 5 18 12 25 27 19 4 19 29
```

We now apply our sort program (or the native function) to the numeric vector J, which is a vector of the subscripts of *TEXT*.

```
      K←SORT J

      K
4 4 5 6 12 13 15 18 18 19 19 19 24 25 27 27 28 29
```

K now represents the ordered subscripts from the array TEXT.

```
      ORDER[J]
XDS (FORMERLY SDS)

      ORDER[K]
DDEFLMORRSSSXY  ()
```

The final sorted output can now be obtained.

The entire process could have been more clearly stated with the statement: *ORDER*[*SORT ORDER*ι*TEXT*].

We have discussed scalar and vector quantities so far. APL can also handle matrix manipulations.

The expression DρX yields a matrix of dimension D whose elements (in row-by-row order) are the elements of the vector X.

As in the case of vectors, we can operate upon matrices with any quantity of a lesser dimension.

Notice that when the righthand argument of the reshape operator (ρ) does not contain enough values to satisfy the left hand, or dimension, argument, the array values are used cyclically.

```
      M←3 4 ρ ι12
      M
1   2   3   4
5   6   7   8
9  10  11  12

      M+1
 2   3   4   5
 6   7   8   9
10  11  12  13

      M←3 4 ρ 1 2 3
      M
1  2  3  1
2  3  1  2
3  1  2  3

      M←3 5ρ'THREESHORTWORDS'
      M
THREE
SHORT
WORDS
```

```
      N←3 4 ⍴ ⍳12

      M+N
 2    4    6    8
10   12   14   16
18   20   22   24

      M[2;3]
7

      M[1 3;1 3 4]
1    3    4
9   11   12

      M[2;]
5   6   7   8
      M[;2 3]
 2    3
 6    7
10   11
```

The expression *M*[3;4] selects the element in the third row and fourth column of the matrix M. More generally, *M*[*I*;*J*] selects the row(s) determined by the elements of the vector I and the column(s) selected by the vector J.

If the index J is omitted, then the entire row (or rows) is (are) taken; if the index I is omitted, the entire columns are taken.

```
      M
1    2    3    4
5    6    7    8
9   10   11   12

      N←⍉M
      N
1    5    9
2    6   10
3    7   11
4    8   12

      ⌽M
 4    3    2    1
 8    7    6    5
12   11   10    9

      ⊖M
9    10    11    12
5     6     7     8
1     2     3     4
```

The expressions ⍉,⌽ *AND* ⊖ each transpose the argument about the axis indicated by the straight line in the symbol.

```
      M+.×N
 30    70    110
 70   174    278
110   278    446
```

The expression *M*+.×*N* denotes the ordinary matrix product of M and N. Matrix multiplication is a combination of addition and multiplication.

```
      M+.=N
4     0     0
0     4     0
0     0     4

      M⌈.⌊N
4     4     4
4     8     8
4     8    12
```

More generally, any pair of operators can replace the operators + and × in the foregoing expression. If $R←Mα.ωN$ (where α and ω represent any pair of operators), then $R[I;J]$ is equal to $α/M[I;]ωN[;J]$.

FUNCTION EDITING

(A listing of AVE:)

```
      ∇AVE[⎕]∇
[1]   'GOOD MORNING'
[2]   'ENTER STRING OF NUMBERS'
[3]   A←⎕
[4]   AVR A
[5]   ∇
```

One of the most important considerations of any programming language is the ability to make corrections/modifications to the original source programs. In some languages it is necessary to re-key the entire program (or at least the line to be changed.) With APL we can add or delete lines or change just one character in any line.

(Correcting AVE:)

```
      ∇AVE[2⎕10]
[2]   'ENTER STRING OF NUMBERS'
             //////5
[2]   'ENTER ARRAY OF NUMBERS'
[3]   [3.5]
[3.5]→(A=0)/0
[3.6][4.5]
[4.5]→3
[4.6]∇
```

Let's return to our first attempt at a program (AVE). You remember that we wrote another program (AV) to include the return and exit statements. Let's see how easy it is to add these, and also change the word 'string' to 'array'.

(New listing of AVE:)

```
      ∇AVE[⎕]∇
[1]   'GOOD MORNING'
[2]   'ENTER ARRAY OF NUMBERS'
[3]   A←⎕
[4]   →(A=0)/0
[5]   AVR A
[6]   →3
[7]   ∇
```

Notice in the new listing that all changes have been made, and that re-numbering has taken place! Luckily, we are not in trouble this time due to our branching to line numbers. It is a better practice to use line labels, such as:

```
[3]   RETURN: A←⎕
[6]   →RETURN
```

We have now examined each of the three modes of APL operation:

(1) Desk Calculator
(2) Function Definition
(3) Function Editing

In addition to the primitive operations discussed in the previous pages, Xerox UTS/APL includes a useful file input/output system.

FILE I/O

UTS/APL provides a set of locked, library functions which permit the user to operate on more data than may be contained in the current workspace. These functions allow any APL value, with its type and dimensional attributes, to be written to a file and subsequently retrieved. The following examples illustrate the use of this facility.

```
      'TESTFILE' FCREATE 1
```

This causes an empty file, with the specified name, to be created in users account and associated with the number '1' for use in subsequent file operations.

```
      FNUMS
1
```

A function is available which will interrogate the status of current file ties. This function returns a vector containing the numbers of all current open files.

```
      A← 3 3 ρ ι 9
      A FAPPEND 1
```

The function FAPPEND adds a new component (record) to a specified file. In the example the matrix 'A' is written to file '1'.

```
      FLIM 1
1  2
```

Now that we have written something in our file, let's exercise a function which will return the current range of component numbers. This function returns a two element vector. The first element is the first existing component number in the file and the second element is one greater than the last component number.

```
      B←FREAD 1 1
      B
 1  2  3
 4  5  6
 7  8  9
      'RECORD TWO' FAPPEND 1
      FLIM 1
1  3
      FREAD 1 2
RECORD TWO
```

Information may be retrieved from a file with the FREAD function. FREAD reads a copy of the specified component from the specified file into the current workspace. Another FREAD operation without a component number specified would cause the next sequential component to be read.

```
      FUNTIE 1
```

When a file is no longer needed by a program, it may be closed and saved with the "FUNTIE" function.

```
      FUNTIE FNUMS
```

All currently tied files may be untied by making the argument for FUNTIE the vector returned by FNUMS.

```
      'TESTFILE' FTIE 9
```

An existing file is opened and associated with a reference number by the FTIE function.

```
      'RECORD THREE' FAPPEND 9
      FLIM 9
1 4
```

Another record is appended to the file.

```
((FREAD 9 3),' UPDATED') FREPLACE 9 3

      FREAD 9 3
RECORD THREE UPDATED
```

This expression reads component 3 of file 9 (TEST FILE), catenates the string 'UPDATED' and replaces component 3 with the result. FREPLACE then allows us to rewrite specific components of a file.

```
      FDROP 9 2
```

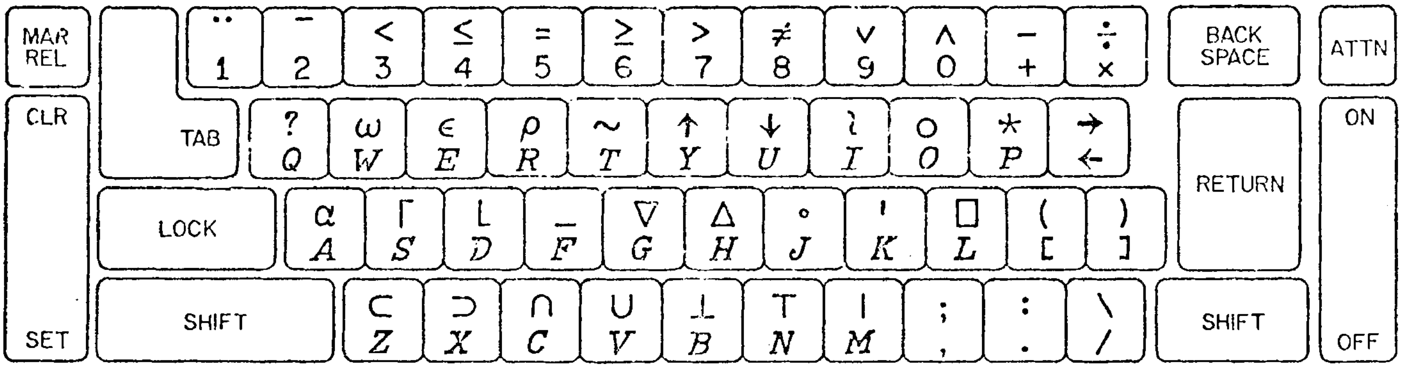
Existing components in a file may be deleted with a function which operates in a manner analogous to the drop primitive. This example causes the first two components of the file to be deleted. If the argument had been 9 ¯2, then two components would have been deleted from the end of the file.

```
      FLIM 9
3 4
```

Component number 3 is now the only component in the file.

```
      'TESTFILE' FERASE 9
```

An unwanted file may be deleted with the function, FERASE.



APL/2741 Keyboard

SAMPLE APL PROGRAM

Matrix Inversion by Gauss-Jordan Elimination
With Pivoting

```
      ∇ B←REC A;P;K;I;J;S
[1]     →3×⍳(2=⍴⍴A)∧=/⍴A
[2]     →0=⍴⎕←'NO INVERSE FOUND'
[3]     P←⍳K←S←1⍴⍴A
[4]     A←((S⍴1),0)\A
[5]     A[;S+1]←S⍺1
[6]     I←J⍳⌈/J←|A[⍳K;1]
[7]     P[1,I]←P[I,1]
[8]     A[1,I;⍳S]←A[I,1;⍳S]
[9]     →2×⍳1E¯30>|A[1;1]÷⌈/|,A
[10]    A[1;]←A[1;]÷A[1;1]
[11]    A←A-((~S⍺1)×A[;1])∘.×A[1;]
[12]    A←1⌽[1]1⌽A
[13]    P←1⌽P
[14]    →5×⍳0<K←K-1
[15]    B←A[;P⍳S]
      ∇
```