APL THINKING
FINDING ARRAY-ORIENTED SOLUTIONS

Robert C. Metzger
Technical Manager
I.P. Sharp Associates
1200 First Federal Plaza
Rochester, NY USA 14614

ABSTRACT

APL is used for processing arrays. This is very different from traditional programming languages, which process single numbers and characters. This difference requires different programming methods. Such methods are array oriented. Several approaches to creating array oriented APL programs are explained here.

INTRODUCTION

Language expresses thought. Most people would agree with this. But fewer people recognize that language influences thought. Study of other cultures shows that they understand the universe differently than Westerners. More importantly, their languages provide different ways of describing the universe. While you can't determine which came first, the language or the culture, certainly one reinforces the other. "We see and hear and otherwise experience very largely as we do because the language habits of our community predispose certain choices of interpretation."[1]

The premise of this paper is that the computer language you use influences how you understand and solve problems. APL presents a very different way of looking at information processing than other languages. Those who use it well have learned to see in this 'different way'. Those who want to use it well need to learn how.

The difference in the APL world view is based upon how data is structured. A traditional programming language, like FORTRAN, presents one view. You see an array of data as a bunch of numbers or characters which can be forced to reside next to each other in memory by giving them a common name. In APL, you see an array of data as a unit. This unit is used as a whole, but can be broken into smaller pieces as the need arises.

The purpose of this paper is to explain some methods which can be used to derive array oriented solutions to data processing problems. The following methods will be explored.

1) Value First, Then Shape;
2) Shape First, Then Value;
3) Data Transformation;
4) Loop First;
5) Think Big;
6) Function Listing;
7) Synonym Search.

These methods are heuristics. A heuristic is the opposite of an algorithm. An algorithm is a set of precisely defined steps which always lead to a desired result. An algorithm is a recipe. A heuristic is a guideline which can help you find a solution to a problem. A heuristic is a trick, which experience has shown to often be useful. There is no guarantee that a heuristic will produce a useful result.

VALUE FIRST, THEN SHAPE

The concept behind this method is that every element in an array has two attributes-- value and position. We can extend this to say that an array has two attributes-- values and shape. So, to obtain a desired result, we have a method which has two steps.
1) Create an array which contains all the desired values. It may also contain some values you don't need.
2) Get the desired values in the proper shape. This may mean removing unwanted values.

An example of this method is an idiom for creating all the integers between two positive integers: $(LO-1)+\iota HI$. If $LO\leftarrow 3$ and $HI\leftarrow 7$, this would execute as shown.

```
(3-1)↓⍳7
2↓1 2 3 4 5 6 7
3 4 5 6 7
```

The Index Generator creates all the desired values, as well as some unwanted ones. The Drop function then removes the dross. This could also have been written in either of the following ways.

```
(LO≥HI)/HI←⍳HI
(1+LO-HI)↑⍳HI
```

As with the first expression, a structural selection function is applied to a set of generated data to obtain the desired result.

A second example of this approach is a solution to the problem of allowing vector arguments to the Index Generator. When we give our *INDEXGEN* function 2 5 3 4 as an argument, we want 1 2 1 2 3 4 5 1 2 3 1 2 3 4 as the result.

This solution creates all possible requested integers, and then determines which to compress out.

```
    ∇ INTEGERS←INDEXGEN1 LENGTH;MAX
[1]  MAX←⌈/LENGTH
[2]  INTEGERS←((⍴LENGTH),MAX)⍴⍳MAX
[3]  INTEGERS←(,LENGTH∘.≥⍳MAX)/,INTEGERS ∇
```

If the argument is 4 2 3, it will execute as shown.

```
[1] MAX←4
[2] INTEGERS←1 2 3 4
             1 2 3 4
             1 2 3 4
[3] INTEGERS←(,1 1 1 1)/,INTEGERS
               1 1 0 0
               1 1 1 0
INTEGERS←1 2 3 4 1 2 1 2 3
```

SHAPE FIRST, THEN VALUE

This method is related to the first. Here we create an array which has the right shape, and then adjust the values where needed. An example is a different approach to the problem discussed above. This expression will give all integers between any two integers: (*LO*-1)+⍳1+*HI*-*LO*. If *LO*←¯3 and *HI*←4, this would execute as shown.

```
(¯3-1)+⍳1+4-¯3
¯4+⍳1+7
¯4+1 2 3 4 5 6 7 8
¯3 ¯2 ¯1 0 1 2 3 4
```

With this method, you begin with a structural function which can force the desired shape. Then you use a mathematical function to alter the values as necessary. Here the structural function is the Index Generator and the mathematical function is Addition.

A second example of this approach is another solution to the Index Generator vector argument problem. This solution creates an array of numbers of the right length. Then it does arithmetic to obtain the correct values.

```
    ∇ INTEGERS←INDEXGEN2 LENGTH
[1]  INTEGERS←(+/LENGTH)⍴1
[2]  INTEGERS[1++\¯1↓LENGTH]←1-¯1↓LENGTH
[3]  INTEGERS←+\INTEGERS ∇
```

If the argument is 4 2 3, it will execute as shown.

```
[1] INTEGERS← 1 1 1 1 1 1 1 1 1 1
[2] INTEGERS←1 1 1 1 ¯3 1 ¯1 1 1
[3] INTEGERS←1 2 3 4 1 2 1 2 3
```

A final example is the idiom which creates a row oriented matrix out of a scalar, vector, or matrix: (1⌈¯2↑⍴*ARRAY*)⍴*ARRAY*. The important part of this idiom is inside the parentheses. If the array is a scalar, we want the calculated shape to be 1 1. If it is a vector of N elements, we want a shape of 1,N. If it is a matrix, we want the same shape that it currently has. First we use the structural function Take to force the shape to have 2 elements. Then we use the mathematical function Maximum to give the shape meaningful values.

DATA TRANSFORMATIONS

The data transformations method begins with the same assumption as the 'Black Box' philosophy. That is, a function may be viewed as an electronic 'black box'. Information is fed into it. The information is transformed in some useful manner, and transmitted back out. You can't see how it works, however. Its walls are opaque.

If your black box breaks down, how would you go about replacing it? You would have to invent a new one. You might do that in the following way.

1) Pick an array which is a representative input. It should contain the important possible variations. Write it down at the top of your paper.

2) Determine what the output of the black box would be if it was given your input. Write the output array at the bottom of the paper.

3) You may work down the page, up the page, or both. When working down, write down an array which is a possible transformation of the one above. Also write down, in natural language, what the transformation is. Put it between the two arrays. DON'T try to come up with an APL expression yet. When working up the page, write down an array which is a possible source of the array below it, if it was appropriately transformed. Write down in natural language, between the arrays, what the transformation is.

4) When you have a series of transformations which take you from the top to the bottom of the page, you are ready to start writing in APL. You should have a natural language statement between each array which describes the transformation. Write an APL expression which will do each transformation. Combine them as necessary. Your data transformer is now ready.

An example of this method is the derivation of an idiom for generating expansion vectors from boolean location vectors. By location vector, I mean a boolean vector, in which 1's indicate the position of some value or sub-array to be processed. In this case, the process is to expand a fill element after the identified element or sub-array.

A representative input would be 0 1 0 0 1 0 0 0 1 1. The corresponding output array would be 1 1 0 1 1 1 0 1 1 1 1 0 1 0. The overall process could be described as transforming 0's into 1's, and 1's into 1 0 pairs.

Working from the bottom, I could say that one transformation would be to group 1 0 pairs together, and separate single 1's from them. If I group things by putting them on separate rows of my paper, I get the following.

```
1
1 0
1
1
1
1 0
1
1
1
1 0
1 0
```

Unfortunately, this isn't a rectangular array. So it won't work. But it does let me see the problem in a different light. How could I make it a rectangular array? I could fill in the holes, with 0's or 1's, on the right or the left. These possibilities are listed below.

```
ONE   ONE   ZERO  ZERO
LEFT  RIGHT LEFT  RIGHT
1 1   1 1   0 1   1 0
1 0   1 0   1 0   1 0
1 1   1 1   0 1   1 0
1 1   1 1   0 1   1 0
1 0   1 0   1 0   1 0
1 1   1 1   0 1   1 0
1 1   1 1   0 1   1 0
1 1   1 1   0 1   1 0
1 0   1 0   1 0   1 0
1 0   1 0   1 0   1 0
```

The first two give the same result. The last one loses information, since all rows are the same. So, we will work with the third. Our choice is reinforced by the fact that column 1 of the third table is identical to our input array.

So our transformation from the bottom is the the following. Group 1 0 pairs together, each in a separate row. Put each single 1 on a separate row, and put a 0 to its left. But this is backwards. What we want is a transform which will get us from the table of 2 columns to the list. This transform starts by removing the first element in a row if it is a 0. Otherwise, the entire row is selected. After selecting the data, the remnants are raveled.

Since we noticed that this table contains our input, let's see if we can transform our input into it. If you look at it carefully, you will see that the first column is our input. The second column is the negation of our input. So now we have a transform between steps 1 and 2, and steps 2 and 3.

Now we can write APL expressions. The first transformation is *TABLE*←*BOOL*,[1.5] ~*BOOL*. The second one is (,*TABLE*[;1],1)/,*TABLE*. We can combine them as (,*BOOL*,[1.5] 1)/,*BOOL*,[1.5]~*BOOL* and we have solved our problem.

The Data Transformation method is a very effective technique for creating array oriented solutions to common problems. In addition, it has the welcome side effect of making writing inverse functions much easier, because you have done much of the preparation.

LOOP FIRST

Good APL programmers avoid looping. This is necessary because almost all current APL implementations are interpreters. Interpreters analyze a statement every time it is executed. This overhead makes interpretive looping expensive. It is possible to largely avoid looping in APL. This is because much of the explicit looping required with other languages is done implicitly by APL primitives.

Given these facts, it may seem odd that I suggest that you use looping as a means of building array oriented solutions. But the method is quite effective. It has four steps.

1) Build a looping solution to your problem.
2) Determine what is the essential work being done inside the loop.
3) Find an algorithm which does in parallel to all parts of the array what the loop was doing serially to one part of the array at a time.
4) Build a non-looping solution to your problem.

An example of this method is given below. Let's say you want to build a function which translates the elements of an array. The left argument will contain the values to be

changed and their replacements. The right argument will be the array to be changed. The result will be the new array. A use of such a function would be to convert all the special characters (Backspace, Linefeed, etc.) in a text into printable characters, without altering the regular characters.

A looping version of this function is shown below.

```
    ∇ ARRAY←TABLE TRANSLATE1 ARRAY
;OLD;NEW;SHAPE;CTR;LIMIT
[1]  OLD←TABLE[;1]
[2]  NEW←TABLE[;2]
[3]  SHAPE←⍴ARRAY
[4]  ARRAY←,ARRAY
[5]  CTR←0
[6]  LIMIT←⍴OLD
[7]  LOOP:→(LIMIT<CTR←CTR+1)⍴END
[8]  ARRAY[(ARRAY=OLD[CTR])/⍳⍴ARRAY]←
      NEW[CTR]
[9]  →LOOP
[10] END:ARRAY←SHAPE⍴ARRAY ∇
```

We recognize that we can take the search for elements to be changed out of the loop. That gives us version 2.

```
    ∇ ARRAY←TABLE TRANSLATE2 ARRAY
;OLD;NEW;SHAPE;CTR;LIMIT;SELECT;CHANGE
[1]  OLD←TABLE[;1]
[2]  NEW←TABLE[;2]
[3]  SHAPE←⍴ARRAY
[4]  ARRAY←,ARRAY
[5]  SELECT←ARRAY∈OLD
[6]  CHANGE←SELECT/ARRAY
[7]  CTR←0
[8]  LIMIT←⍴OLD
[9]  LOOP:→(LIMIT<CTR←CTR+1)⍴END
[10]  CHANGE[(CHANGE=OLD[CTR])/⍳⍴ARRAY]←
       NEW[CTR]
[11]  →LOOP
[12] END:ARRAY[SELECT/⍳⍴SELECT]←CHANGE
[13]  ARRAY←SHAPE⍴ARRAY ∇
```

Now the only work being done inside the loop is working with elements which will be changed. We can go one step further. We can use the translation idiom *NEW[OLD⍳DATA]* and do all of the substitutions simultaneously.

```
    ∇ ARRAY←TABLE TRANSLATE3 ARRAY
;SHAPE;SELECT
[1]  SHAPE←⍴ARRAY
[2]  ARRAY←,ARRAY
[3]  SELECT←ARRAY∈TABLE[;1]
[4]  ARRAY[SELECT/⍳⍴SELECT]←
     TABLE[TABLE[;1]⍳SELECT/ARRAY;2]
[5]  ARRAY←SHAPE⍴ARRAY ∇
```

Here is a second example of the Loop First approach. We will derive idioms for finding unique elements in a vector. A looping solution is listed below.

```
    ∇ NUB←UNIQUE1 SET;CTR;LMT
[1]  NUB←⍳0
[2]  CTR←0
[3]  LMT←⍴SET
[4]  LOOP:→(LMT<CTR←CTR+1)/END
[5]  →(SET[CTR]∈NUB)/LOOP
[6]  NUB←NUB,SET[CTR]
[7]  →LOOP
[8]  END: ∇
```

This function is very similar to solutions in other languages. It would be virtually identical if the use of ∈ were changed to a loop also.

We inspect the inside of the loop for the essence of the problem. The search for an element in the result really is a search for the element in the original set. After all, the result comes from the original set. But if we try *X∈X* directly, we don't get a useful result. What other ways can we search an array for its elements? We can say *X⍳X*. If we remember that searching is comparison, *X∘.=X* also fills the bill. And since sorting requires the comparison of the elements in an array against each other, *X[⍋X]* may also help.

We choose to ignore the line after the search statement. Since this is an iterative process, it must have a way of building up a result. So the catenation is really just part of the bookkeeping caused by using a loop. The same rule applies when an array is initialized, and then written over piece by piece in a loop.

Let's choose a sample array and try out our expressions.

Let *X←'THE CAT IN THE HAT'*.

X⍳X gives

```
1 2 3 4 5 6 1 4 9 10 4 1 2 3 4 2 6 1.
```

X∘.=X gives

```
 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0
 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0
 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0
 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0
 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
```

X[⍋⌷⎕AV⍳X] gives ' *AACEEHHHINTTT*'

In the first case, we see that duplicates all have the same index position. In the second, we notice that if we could set to zero all but the first 1 in each row, the main diagonal would identify the unique elements. In the third case, we observe

that if we could identify the first element in each group, we would have the unique elements.

These observations lead us to three different ways to select unique elements. Questions of efficiency depend on the implementation you are using.

```
    ∇ NUB←UNIQUE2 SET
[1]  NUB←((SET⍳SET)=⍳⍴SET)/SET ∇

    ∇ NUB←UNIQUE3 SET
[1]  NUB←(1 1 ⍉<\SET∘.=SET)/SET ∇

    ∇ NUB←UNIQUE4 SET
[1]  SET←SET[⍋SET] ⍝ SET[⎕AV⍳SET] FOR CHARS
[2]  NUB←(SET≠¯1⌽SET)/SET ∇
```

We have seen that starting with a looping solution provides insight into the problem. It also gives you a valuable prototype which can be used in testing the final version. You end up with an array oriented solution to your problem.

THINK BIG

Thinking big means looking beyond your limited horizons. For most of us, that means moving beyond matrices to arrays of 3 or more dimensions. Even if the target array you want to produce may be a vector or matrix, this approach could help you. There are 4 steps.

1) You must be aware of which APL primitives can produce an array whose rank is greater than the rank of either argument. These are Reshape, Indexing, Lamination, Encode, and Outer Product.
2) Try to picture your input data arranged in a higher order array. What does each dimension represent? Write it down.
3) Choose one of the functions listed above to build your array. Work out the transformation.
4) Work out the transformation which will take the higher order array and transform it into what you want.

An example of this aproach is a solution to the problem of computing variances. The most common variance computation is comparing Budget figures to Actuals. Usually, there are two matrices-- Budget and Actual. Each has columns related to time, (months, quarters, years), and rows related to accounts, products, locations, etc. The desired result has as many rows as the original matrices, but has 3 times as many columns as the originals. The columns are to be Period I Budget, Actual, Variance; Period II Budget, Actual, Variance, etc.

Since the Budget and Actual matrices are the same shape, we can use Lamination to build a cube. The new dimension represents type of data. To it, we cah catenate the difference between the two planes. Now we have all the data in the cube. We need to re-arrange the data so that related columns are together, and then convert it to a matrix. Dyadic Transpose can shuffle the columns, and Reshape finishes the task.

```
    ∇ R←B VARIANCE A
[1]  R←(B,[0.5] A),[1] B-A
[2]  R←((¯2↑⍴R),×/¯2↑⍴R)⍴ 3 1 2 ⍉R ∇
```

Another example of this method is the problem of element replacement. In this situation, we would like to replace every occurrence of a designated scalar found in a vector by a group of scalars. Traditional approaches to this problem expand the vector to the proper shape, and insert using indexing.

```
    ∇ VECTOR←VECTOR EREPLACE1 PARMS
;LOCATIONS;POSITIONS;EXPAND
[1]  LOCATIONS←VECTOR=1↑PARMS
[2]  POSITIONS←LOCATIONS/⍳⍴LOCATIONS
[3]  LENGTH←¯2+1⌈⍴,PARMS
[4]  EXPAND←
     ((⍴,VECTOR)+LENGTH×⍴POSITIONS)⍴0≠0
[5]  EXPAND[(POSITIONS+LENGTH×¯1+
     ⍳⍴POSITIONS)∘.+¯1+⍳LENGTH+1]←1
[6]  VECTOR←(~EXPAND)\(~LOCATIONS)/VECTOR
[7]  VECTOR[EXPAND/⍳⍴EXPAND]←
     (+/EXPAND)⍴1↓PARMS ∇
```

A different approach builds up an array which contains the original vector, the group of elements to replace it, and a lot of blanks for padding. Then it compresses out the element to be replaced, and the extra blanks.

```
    ∇ VECTOR←VECTOR EREPLACE2 PARMS
;LOCATIONS
[1]  LOCATIONS←VECTOR=1↑PARMS
[2]  VECTOR←,VECTOR,LOCATIONS\
     ((+/LOCATIONS),¯1+⍴PARMS)⍴1↓PARMS
[3]  VECTOR←(,LOCATIONS∘.≠(⍴PARMS)↑1)/VECTOR
∇
```

These functions are also examples of SHAPE FIRST, THEN VALUE and VALUE FIRST, THEN SHAPE, respectively.

So far, we have created higher rank arrays using structural functions. But Encode and Outer Product can also be used to create such arrays. In these cases, we make use of a common APL coding pattern-- Reduction applied to the result of Inner Product.

One example of this is the idiom which counts the number of times each element occurs in a vector: `+/VECTOR∘.=VECTOR`. If `X←'ABABCDABC'`, then `+/X∘.=X` gives 3 3 3 3 2 1 3 3 2. Another example is the idiom for classifying data within numeric ranges: `+/DATA∘.>0,RANGES`. If `PRIMES←2 3 5 7 11 13 17 19 23`, then `+/PRIMES∘.>0 5 10 15 20` gives 1 1 1 2 3 3 4 4 5. The reason these idioms are helpful is that, while they create new information to be processed, they use it all. This stands in contrast to the expression `1 2 2⍉MATRIX∘.+VECTOR`. This adds

the vector to each of the rows of the matrix. It unfortunately also creates a lot of data which is calculated only to be thrown away. The expression *MATRIX*+(⍴*MATRIX*)⍴*VECTOR* serves this need much better. So, if you are going to think big, be careful that you don't forget than one the reasons for APL Think is efficiency.

SYNONYM SEARCH

The method of synonym search is really an idea generating mechanism. If you have no idea where to begin writing an APL statement to solve a problem, it will help you with the first step.

There are 3 parts to this approach.
1) Write out your understanding of the process in natural language.
2) Go through your description and underline the verbs and adverbs.
3) Look up the verbs and adverbs in an alphabetized list of APL synonyms. (Such a list is included in the Appendix). Write down the associated APL primitives.

The premise of this method is that APL arrays, functions, and operators are analagous to natural language nouns, verbs, and adverbs. Functions act on arrays, and operators modify the action of functions. Therefore, the verbs and adverbs in the process description should point you to functions and operators which do what you want.

You will find that a few uses of this method will make it seem natural to you. Then you will be able to do it mentally without writing or using the list.

FUNCTION LISTING

This technique comes from the literature of creative problem solving.[2] It is a natural follow-on to the Synonym Search method. Once you have some potentially useful functions, how do you build a statement from them? You can use a 'Forced Relationship' technique like this to generate ideas.

Here's how it works.
1) List all of the functions you feel might be useful in this problem.
2) Pair each function with every other function. (If you have N functions, you will have 2!*N* pairs).
3) Go through the pairs and select those which are promising.

The first step might come from another heuristic, like the Synonym Search. The second is purely mechanical. But how do you decide which pairs are promising? You have to look for instances of common patterns and related functions.

The most obvious relationships between functions are inverse, dual, and negation. If a pair you have listed is found among the pairs shown below, it may be of use.

Inverses	Duals	Negations
+ -	⌈ ⌊	< ≥
× ÷	∨ ∧	≤ >
* ⍟		> ≤
⊥ ⊤		≥ <
⍋ ⍒		= ≠
		∧ ⍲
		∨ ⍱

Another type of relationship is when the range (output set) of the function executed first is the domain (input set) of the function executed second. Some of these relationships are listed below. When your pair includes one function from the left column, and one from the right column, you should explore it further.

Domain	Range
∨ ∧ ⍱ ⍲	< ≤ = ≥ > ≠ ∈
∨ ∧ ⍱ ⍲	∨ ∧ ⍱ ⍲
/ ⌿	< ≤ = ≥ > ≠ ∈
/ ⌿	∨ ∧ ⍱ ⍲
[]	⍳ ⍋ ⍒

There are several common patterns that you see in APL idioms. These patterns occur because the primitives used are complementary to each other. Several common patterns are listed below.

Left	Right
⍉	⊤
⌹	⊥
⍎	Scan
Reduction	Outer Product
Scan	Outer Product
Reduction	Inner Product
Scan	Inner Product

If your pair fits one of these patterns, it is a promising candidate.

Once you've chosen some potential function pairs, try them out together on some sample data. Look for data transformations which resemble the one you're looking for. Build your statement from them.

CONCLUSION

This paper has explored several methods for deriving array oriented APL expressions. Many of these techniques are just the explicit statement of the thought process that good APL programmers go through unconsciously all the time. These techniques are a practical and effective means of teaching novice APL users to become more adept in using APL. It is my hope that these techniques will take away some of the mystique of APL programming. APL faces stiff competition in the programming language marketplace. If it is going to become more widely used in the face of this competition, effective APL techniques must

be understood by the average user, not just an elite few.

REFERENCES

[1] Language, culture, and personality, essays in memory of Edward Sapir, quoted in Language, Thought, and Reality, Benjamin Lee Whorf, Ed. John B. Carroll (Cambridge, MA: The MIT Press, 1956), p. 134.

[2] Alex F. Osborn, Applied Imagination, 3rd ed., (New York: Charles Scribners Sons, 1963), p. 214

APPENDIX

All	∧/
Any	∨/
Biggest	⌈/
Connect	,
Delete	⍪
Expand	\
Join	,
Match	∧.=
Move	[]
Move	⌽
Order	[]
Pair	∘.
Product	×/
Repeat	⍴
Search	∈
Search	⍳
Select	⍴
Select	/
Select	↑
Select	[]
Shuffle	⍉
Slice	⍉
Smallest	⌊/
Sort	⍋
Sort	⍒
Sum	+/
Tip	⍉