50 Shades of J

Based on J-ottings *Vector* articles
1993—2017

by

Norman Thomson

Contents

Preamble ... 4
1. every, each, and a little bit of rank ... 8
2. A Composition on Composition ... 15
3. My J-oinery Workshop ... 25
4. Parallel Joins ... 30
5. Conjugacy and Rank ... 37
6. Punctuation and Rank ... 44
7. One Foot in the Grade ... 52
8. Transpositions, Perms and Combs ... 58
9. Power Steering extra ... 63
10. Bonding is Power – how Interesting ... 69
11. Time for amendment of data ... 73
12. Obverse to Adverse :.:: a trip along the Brailleway ... 78
13. If you think J is complex try j ... 85
14. j is complex? You bet! ... 97
15. Cancel, cancel little fraction ... 107
16. Thinking by numbers ... 111
17. Jaesthetics ... 119
18. The problem with J is… ... 126
19. Symphony in J minor, op.31 ... 131
20. How to Do Things with Words ... 136
21. Are you thinking what I'm thinking? ... 142
22. Index Lists, Grade Lists, and some simple joins ... 147
23. Some Numerical Problems Analysed in J ... 153
24. Here we go round… and round and round… ... 159
25. Two for the Price of One ... 164
26. Working in Groups ... 168
27. All but one ... 175
28. Have you a weight on your mind? ... 182
29. Just say it in J - ANOVA ... 186
30. Just what do they sell at C&A? ... 193
31. A rippling good yarn ... 198
32. So Easy a Child of 10… ... 202
33. Perming and Combing ... 210
34. Combination Lists ... 219
35. How many Obtuse angled triangles are there? ... 227
36. …the stylish part of Vector ... 232
37. Greed : patterns for the collapse of Western capitalism ... 238
38. Shortest Paths ... 245
39. The I-spy book of J ... 254
40. The I-spy book of J part 2 ... 260

41. Suffer the little children… to bring along their equations............264
42. Fifty ways to tell a fib 273
43. Seeds, Cones and Sunflowers...278
44. Catalan Numbers 287
45. A partial solution to a partial problem...293
46. Tables and Geometry 301
47. Musical J-ers..305
48. Heavens above!..318
49. Financial Maths and J – part 1, IRR and APR.....................................329
50. Financial Maths and J – part 2; Growth Rates 336

Preamble

Read this First!!

"Fifty Shades of J" is a collection of 50 essays which have appeared over a period of years as the J-ottings column in Vector. They have been updated and standardised so that all the code quoted runs on J602. Although many of the essays are about basic features of J, it should not be regarded as a language primer, but rather as providing some thoughts, insights and worked examples for the user who has already cleared the first hurdles of learning J. A 'Principal Topics' lists heads each essay, and most have a concluding Code Summary, which should prove helpful in giving a rapid view of the ground covered. The essays are not printed in their order of appearance in Vector, but rather have been arranged and in some cases merged, so that the first sixteen each emphasise a single primitive or feature, the next five are about the underlying philosophy of J, and the remainder have some specific problem or problems as a starting point, and show how readily J can be used to address it. In some cases the original titles and emphasis have been changed, although I have resisted too much over-polishing in the belief that is might be better to retain something of the original spirit of discovery. Many of the essays have a mathematical slant which reflects that one of the originals objectives of J, like APL before it, was to iron out some of the ambiguities and anomalies of conventional mathematical notation by creating an immediately executable bridge between mathematical theory and practice.

APL and its descendants have always been regarded as array processing languages. However, I consider that it is more helpful to view J as primarily a list processing language for reasons which will become clearer as the reader proceeds through the essays. These are numbered as e.g. E #42 (i.e Essay no. 42), for cross-reference, and also for the purpose of indexing. It is hoped that the index will help make the book a useful collection of phrases for the reader who is maturing into J. J has a formal vocabulary of English words which describe its symbol vocabulary. Where English words are used in this way they appear in italics thus : *signum*. The J symbol vocabulary can be divided into sets, first those symbols with which most users are very familiar either because of their general usage in life, e.g. the symbols of basic arithmetic, or because they arise so frequently in using J. Those in the former set do not generally appear in the Principal Topics sections at the head of each essay, unless there are some special aspects of them

which are illustrated in that particular essay. Typical members of this set are the arithmetic and logical verbs

+ - * % ^ ! (factorial)

= < > >: and <: (dyadic) ~: (*not equal)*

The only unfamiliar form is monadic * meaning *signum*. The symbols derived from the above for *double, halve, square* and *square root*, together with *increment* and *decrement* also fall into this category viz.

+: -: *: %: <: >:

as do the di-grams +/ and */ for sum and product of lists (>: and π in mathematics terms). The second set comprises the more specialised symbols which are often harder to pick out in the strings of characters which make up typical J expressions.

Series, that is lists, are an essential part of the discourse of mathematics, and it is assumed that the reader has grasped the important distinction between the joining verbs *ravel* (,) and *link* (;). *tally* (#) and *reverse* (|.) are also necessary elements of any programming language which admits strings. *head/behead* and *tail/curtail* are desirable additions to the basic set, although a degree of memorisation is required before it becomes ingrained that in J { means "take just one", } means "take all but one", dot means "from the left" and colon means "from the right".

Intelligent reading of J requires an appreciation of verb composition which can be implicit, that is *hook* and *fork*, or explicit as in the various forms of @ and & t, that is the conjunctions *at, atop, bond, compose, under, appose.* These are of such wide occurrence that they are dealt with in E #2 ("A Composition on Composition") and thereafter do not appear in the Principal Topics lists. Linked with these is the ~ (*passive* conjunction) which is often manifest in what I term the 'bridge hook' u~v. These symbols do not feature in Principal Topics, instead they are comprehensively described in E #2.

The items in lists may be numbers, of which there are six convertible representations (see E #14 "j is complex? You Bet!"), characters as given by a. (*alphabet*), or lists. Whenever lists consist of lists, it becomes essential to be aware of rank (see E #5 "Conjugancy and Rank" and E #6 "Punctuation and Rank"), that is for the user to be

consciously aware of depth within a hierarchy of lists. Rank is a consideration for all but the simplest operations, from which it follows that fluency in using the *rank* conjunction (") is assumed. This is used in some way in almost half of the essays, for which reason it does not always appear in Principal Topics lists.

There are two defined adverbs which are used repeatedly and are part of the J standard vocabulary, namely

```
every=.&>
```

and

```
each=.&.>
```

whose definitions are not repeated in individual essays. Also the *power* conjunction pervades many essays – roughly speaking it provides in a single digraph a style of looping activity which is characteristic of most computing languages.

By contrast with "At Play with J" where Gene McDonnell typically took a substantial problem, and solved it, sometimes at some length, my aim has been, particularly in the first set of essays, to work outwards from what is already an algorithmically rich base, and by looking at primitives in detail, show ways of making them even more useful and interesting. Gene and I frequently exchanged transatlantic e-mails, and I like to think that his top-down approach is complemented by my more bottom-up one of broadening the possibilities available through J's basic elements.

When J first appeared, its aficianados were carried away to some extent by enthusiasm for tacit programming, just as APL users were entranced by the ability to write one-liners. Tacit programming used uncritically can lead to long and incomprehensible lines which defeat Ken Iverson's initial objective of a computing language which reflected the styles and usage of natural languages, thereby making their meaning that much clearer than the statements of more conventional computing languages. As a broad rule, once 'pure' lines exceed around seven or eight characters, it is usually better to consider defining a new named verb or adverb, if necessary building up a chain of mixed new names and J primitives to achieve a final overall objective. Also even working within the 'seven or eight' line rule, over-enthusiasm for tacit programming can sometimes be less clear than explicit definition us-

ing x and y as left and right arguments. In the ensuing pages, I have tried to use whichever seemed to me the clearer style in each individual circumstance, so that as well as code quoted being executable by copying, it should also be self-describing.

For this purpose the Code Summaries contain the main verbs, adverbs and nouns in a hierarchical order indicated by indentation. This is in contrast to the 'bottom-up' fashion in which they have generally been developed in the text of their essay. Broadly speaking 'Principal Topics' describe what the essay sets out to do and the J features which required to help get there, the essay body decribes the process of getting there, and the Code Summary consolidates the final point of arrival. For sufficiently fluent J readers the last is all the information they need! (see E #21 "Are you thinking what I'm thinking?"). There is also a Topics Index and a Vocabulary Index which reference essay numbers rather than page numbers. Broadly speaking the former lists topic areas which have seemed to me to be amenable to J treatment, while the latter covers applications suggested by the J primitives themselves.

Many of the essays were triggered by other articles in Vector. All such references to these are embodied within the text. Words which would be emphasised in speech are underlined, an bold type is used occasionally where this helps definitions and specialised terms stand out.

1. every, each, and a little bit of rank

Principal topics : & (*compose)* &. (*under)* @ *(atop*). " (*rank* conjunction) , fill characters, ragged arrays, heterogenous arrays, inner product.

In APL2 there was just one primitive symbol (double dot) for 'each' – the 'pepper' effect is well imprinted on those with long memories, although that is not what APL looks like these days. In spite of J being considerably richer in primitives than APL, it does not include 'each' as a primitive. Instead J recognizes that there are two types of 'each', both defined as adverbs in the standard vocabulary and both using *open*

```
every=.&>         NB. uses compose
each=.&.>         NB. uses under
```

The definitions of the conjunctions u&v and u&.v in mathematical terms as (uv) and (v^{-1}uv) show that the rule

```
(f every) -: (>f each)
```

is completely general, and so only one of 'each' and 'every' is ever logically necessary. 'each' has the general merit of greater compactness through generating what APL2 users would recognise as ragged arrays. On the other hand 'every' has the merit of avoiding, or at least reducing, boxing on display, but often at the expense of requiring greater numbers of fill characters to achieve global homogeneity. In writing J it can be handy to have a 'rule of thumb' appreciation of how 'each' and 'every' work without having to consider the semantic details of *box* and *open* at every point of use. Thinking in terms of lists is helpful in using 'each' and 'every' effectively, as is sensitivity to the ways in which argument rank works. J has a wealth of algorithms pre-implemented for its users within its interpreters. Effective use of these requires an acknowledgemnt that Data = Content + Structure, and through 'each' and 'every' it makes demands on the accurate pinpointing of where algithms at structural levels of Data. For example, given

```
h=.i.2 3         NB. two 3-lists, rank = 2
```

compare the following

```
   h,every 6 7      NB. one 2-list, items=3 2-lists, rank=3
0 6
1 6
2 6

3 7
4 7
5 7

   h,each 6 7      NB. 2 3-lists, items=2-lists, rank=2
┌───┬───┬───┐
│0 6│1 6│2 6│
├───┼───┼───┤
│3 7│4 7│5 7│
└───┴───┴───┘
```

Applying the general rule above `h,every 6 7` is identical to `>h,each 6 7`, or to put it in another way, the two displays above show the same six 2-lists, in one case individually boxed, in the other integrated into a structure of higher rank.

Using either 'each' or 'every' with a dyadic verb often requires the user to *box* one of the arguments to achieve a desired result through bringing about correct argument matching. Often this consists of a simple boxing of one of the arguments as in

```
   x=.6 7;8;9 10 11 12
   (<h),each x     NB. left rank 0, right rank 1, result rank 1
┌─────┬─────┬───────────┐
│0 1 2│0 1 2│0  1  2  0 │
│3 4 5│3 4 5│3  4  5  0 │
│6 7 0│8 8 8│9 10 11 12 │
└─────┴─────┴───────────┘
```

The result is a 3-list of scalars (and so of rank 1) each made from the same scalar (`<h`) joined to each item of a 3-list, with everything *open*ed and *box*ed following the ($v^{-1}uv$) rule which defines &. (*under*). There is a local fill character in the first box, a local scalar expansion in the second, and the third containing two fill characters in the final column has a different *shape* ($) from the other two. Compare this with

```
   (<h),every x    NB. left rank 0, right rank 1, result rank 3
0   1   2   0
3   4   5   0
6   7   0   0

0   1   2   0
3   4   5   0
8   8   8   0

0   1   2   0
```

```
3   4   5   0
9  10  11  12
```

where there is no v^{-1} boxing. Again the result is a 3-list, but now not of scalars but of three filled lists, each of which is a 4-list of scalars after filling. APL2 talked about 'ragged' and 'heterogeneous' arrays – a more appropriate differentiation in J is between locally- and globally-filled lists, corresponding to each and every. Using "0 as an alternative gives a result rank between those of every and each :

```
  (<h),"0 x      NB. left rank 0, right rank 1, result rank 2
┌─────┬───────────┐
│0 1 2│6 7        │
│3 4 5│           │
├─────┼───────────┤
│0 1 2│8          │
│3 4 5│           │
├─────┼───────────┤
│0 1 2│9 10 11 12 │
│3 4 5│           │
└─────┴───────────┘
```

Here the scalar left argument gives rise to scalar replication, and the result is a 3-list of non-homogeneous 2-lists without fill characters, that is, it is only the display which is rectangular not the result object itself, as is made explicit by

```
   $each (<h),"0 x
┌───┬─┐
│2 3│2│
├───┼─┤
│2 3│ │
├───┼─┤
│2 3│4│
└───┴─┘
```

In order to avoid monotonous repetition, subsequent examples will use just one of 'each' or 'every', the other case being covered by the general rule (f every) -: (>f each) .Consideration of rank is almost always a preliminary to well-considered usage of `each`. There are no overall rank rules because these depend on the semantics of each individual f . In the next set of examples three figures in square brackets in a comment give the left, right and result ranks of the verb `|.` or `|.each` .

Consider first the basic case of the verb *shift* with a rank 1 argument on the left and a rank 2 argument on the right :

```
   0 1|.h        NB. 0-shift rows ,: 1-shift cols    [1 2 2]
```

```
1 2 0
4 5 3
```

Applying each with *box* on either right or left changes the ranks :

```
   0 1 |.each <h  NB. 0-shift on h ; 1-shift on h    [1 0 1]
┌─────┬─────┐
│0 1 2│3 4 5│
│3 4 5│0 1 2│
└─────┴─────┘
  (<0 1)|.each h NB. (<0 1)-shift on items of h     [0 2 2]
┌─┬─┬─┐
│0│1│2│
├─┼─┼─┤
│3│4│5│
└─┴─┴─┘
```

Between these extremes is the possibly more useful case of 'left rank 1, right rank 1':

```
  0 1|.each<"1 h   NB. 0-shift 1st row;1-shift 2nd     [1 1 1]
┌─────┬─────┐
│0 1 2│4 5 3│
└─────┴─────┘
```

Next, given the laminated rank 3 object :

```
   hlam=.(i.2 3),:10+i.2 3
   hlam
 0  1  2
 3  4  5

10 11 12
13 14 15
```

here are the effects of four rather similar expressions, along with annotated descriptions which attempt to explain the differences between them. What is the best medium for such descriptions? Why J, of course, hence the comments which follow each of the four executable lines supply matching equivalent statements

```
  (<0 1)|.each <"2 hlam NB. (0 1|.h);(0 1|.h+10)       [0 1 1]
┌─────┬────────┐
│1 2 0│11 12 10│
│4 5 3│14 15 13│
└─────┴────────┘
  0 1|. each <"2 hlam    NB. (0|.h);(1|.h+10)           [1 1
1]
┌─────┬────────┐
```

```
│0 1 2│13 14 15│
│3 4 5│10 11 12│
└─────┴────────┘
```

```
   0 1|. each <"1 hlam    NB. (0|. each H<"1 h),:(1|. each <"1 
h+10)
┌────────┬────────┐
│0 1 2   │3 4 5   │
├────────┼────────┤
│11 12 10│14 15 13│
└────────┴────────┘
```

This example is also equivalent to `2 2$0 0 1 1|.` each `H,<"1 hlam,` since `,<"1 hlam` is a 4-list whose items are 3-lists such as 0 1 2. The next example illustrates how "0 means apply a shift at the scalar level which is necessarily a 'do nothing' operation.

```
   (0 1)|."0<"2 hlam     NB. equivalent to <"2 hlam     [1 1 1]
┌─────┬────────┐
│0 1 2│10 11 12│
│3 4 5│13 14 15│
└─────┴────────┘
```

A further set of examples illustrates what happens with characters. g is a pair of 2-lists and thus has the same shape as h, only its items are characters lists rather numbers.

```
   ]g=.2 3$'ant';'bee';'cat';'dog';'elk';'frog'
┌───┬───┬────┐
│ant│bee│cat │
├───┼───┼────┤
│dog│elk│frog│
└───┴───┴────┘
```

Where items are characters lists, there are necessarily possible ambiguities between genuine space characters and space fill characters, something which is clarified in

```
      1 |. each g             NB. 1-shift items   ranks=[0 2 2]
┌───┬───┬────┐
│nta│eeb│atc │
├───┼───┼────┤
│ogd│lke│rogf│
└───┴───┴────┘
```

The next example shows simultaneous opening of both arguments followed by v^{-1} boxing :

```
   (<0 1)|. each <g           NB. <0 1|.g,     ranks=[0 0 0]
┌──────────────┐
│┌───┬────┬───┐
```

```
│bee│cat │ant│
├───┼────┼───┤
│elk│frog│dog│
└───┴────┴───┘
```

The next example is one in which the result rank is less than an argument rank :

```
   0 1|. each <g              NB. (<0|.g),(<1|.g),ranks=[1 0 0]
┌──────────────┬──────────────┐
│┌───┬───┬────┐│┌───┬───┬────┐│
││ant│bee│cat ││││dog│elk│frog││
│├───┼───┼────┤││├───┼───┼────┤│
││dog│elk│frog││││ant│bee│cat ││
│└───┴───┴────┘│└───┴───┴────┘│
└──────────────┴──────────────┘
```

the final example in this set the shapes of the items within the two outer boxes are not identical, thus resolving an ambiguity between fill and genuine space characters which each would not have shown :

```
   0 1|. each <"1 g         NB.(<0|.0{g),(<1|.1{g),ranks=[1 1
1]
┌─────────────┬──────────────┐
│┌───┬───┬───┐│┌───┬────┬───┐│
││ant│bee│cat││││elk│frog│dog││
│└───┴───┴───┘│└───┴────┴───┘│
└─────────────┴──────────────┘
```

However, given that each by definition reduces one level of boxing in every case, it often gives less fussy looking results than each where character lists are involved.

The examples so far have been chosen to illustrate the principles of each and every. An example of a practical problem involving each is that of evaluating linear expressions such as x + 2y - z over separate ranges of values of their variables. The results in this simple example are easy to check.

```
   t3=.1 2;3 4 5;6            NB. x={1,2} y={3 4 5} z={6}
   {t3                        NB. catalog gives all combinations
┌─────┬─────┬─────┐
│1 3 6│1 4 6│1 5 6│
├─────┼─────┼─────┤
│2 3 6│2 4 6│2 5 6│
└─────┴─────┴─────┘
   ip=.+/ .* every            NB. 6 inner products with 1 2 _1
   ({t3)ip <1 2 _1
│1│3│5│
```

```
│ │ │ │
├─┼─┼─┤
│2│4│6│
└─┴─┴─┘
```

The advent of 'each' and ragged arrays in APL2 breathed new life into that language. J deals with such matters more subtly by forcing choices between the two possibilities 'each' or 'every'. Inm broad terms 'each' allows raggedness and reduces homogeneity to a local level, 'every' reduces boxing at the cost of imposing greater homogeneity through globally applied fill characters.

Code Summary

```
every=.&>
each =.&.>
ip=.+/ .*
```

2. A Composition on Composition

Principal Topics : @ (*atop*) @: (*at*) & (*bond/compose*) &. (*under*) &: (*appose*) -: (*match*) ^. (*log*) -. (*less*) [(*left*)] (*right*) %. (*matrix inverse/divide)*
~ (*passive* conjunction), b. (*basic characteristics*) *hook*, *fork*, conjunctions, bridge hook, rank, rank inheritance, mean, geometric mean, harmonic mean, moments, normalization, transformations, valence.

Composition lies at the heart of the J language, and is based on the use of conjunctions, understanding which involves appreciation of the concept of *rank*. 'Compose' in a general sense means "make a composite verb from two others", but 'composition' also extends to the broader range of forming composite objects using explicit conjunctions, and also implicit ones as in the case of *hook* and *fork*.

Joining nouns and verbs

The simplest use of conjunctions is to join a noun to a verb in either order to form a new verb, something which is a familiar part of the fabric of natural language. For example, 'daydream' is a verb whose meaning cannot be fully conveyed by using 'day' or 'dream' separately, and the same applies to compound verbs such as 'headhunt', 'facelift', 'datestamp', etc.

```
   (-&2)6 7
4 5
   (2&-)6 7
_4 _5
```

where the composite verbs in parentheses could be thought of as 'take-two-from' and 'take-from-two'.

Compositions of two verbs are possible as in 'sleepwalk', 'tumbledry', 'stirfry', and again in each case the composition has an extra layer of meaning which exceeds the sum of its two constituents. For example the familiar technique of multiplying numbers by adding their logarithms, and then using anti-logarithms to obtain the final result. The monadic verb ^ means "raise e to the power", and its inverse ^. means "get the natural logarithm" . (Powers and logarithms to other bases are obtained by applying these verbs in their dyadic forms with the base as left argument) To *compose* (&) logs with addition gives the logarithm of a product, for example

```
   2+&^.3
1.79176
```

gives the sum of the natural logarithms of 2 and 3 to give the logarithm of 6, and the multiplication process can be completed by using *under* (&.) :

```
   2+&.^.3
6
```

Both 'verb-noun' and 'noun-verb' cases are examples of *bonding*. In Function terms *under* gives g^{-1} f g rather than just f g . Thus `( <:&*: )n` means n^2-1 whereas `( <:&.%: )n` means $\sqrt{(n^2-1)}$. If the verbs are switched `( *:&<: )n` means $(n-1)^2$ while `( *:&.<: )n` means $(n-1)^2$ +1 .

The atop conjunction

Sometimes in programming it is useful to start a sequence of counting integers at 1 rather than 0 - this is a basic application of @

```
   int=.>:@i.        NB. integers from 1 to y
   int 4
1 2 3 4
   int 2 3
1 2 3
4 5 6
```

If the constituent verbs are reversed, incrementing comes first and the effect is to change the argument of `i.` and hence the shape of the result:

```
   (i.@>:)4
0 1 2 3 4
   (i.@>:)2 3
0 1  2  3
4 5  6  7
8 9 10 11
```

Atop and hook

The primitive verb | returns the absolute value or values of the items in a numeric list. Thus `>./@|` returns the maximum absolute value in the list.

```
   maxabs=.>./ @: |
   maxabs  2 7 _11 9
11
```

Now use this composed verb as the right argument of a *hook*, which is itself a form of composition without an explicit symbol. %maxabs divides all the list items by the maximum absolute value, so that

```
  normallise=.%maxabs
```

scales a list so that the largest absolute value is 1.

```
   normalise 2 7 _11 9
0.181818 0.636364 _1 0.818182
```

Atop and at

The essence of *atop* and *at* is sequencing, in the sense in which lines are sequenced in conventional programming languages. Indeed the presenc of many @s and @:s in a J expression is often an indication that the purpose would be more clearly achieved by rewriting the line in a simpler explicit multi-line style. Sequencing can take place in possible ways. Returning to the analogy of 'stirfry', two interpretations of this composition are possible; either every morsel is fried and immediately stirred, that is the verbs 'stir' and 'fry' are fused, or alternatively the frying is applied to everything, and the result is passed to the stirring process. Such distinctions necessarily involve the concept of verb rank.

Some verbs always enforce processing at the level of atomic elements (scalars), even if their arguments are of higher rank, in which case the result is obtained by an appropriate extension from the simple scalar case. The commonest verbs of this kind are the dyadic arithmetic and logic verbs + - * % ^ >. <. +. and *. . Such verbs are called rank-zero verbs. Verb rank may differ in monadic and dyadic cases, for example monadic + is not a rank-zero verb since with a numerical argument `+n` is just `n` and there is no requirement for atomic level processing. In general a verb possesses three verb ranks, one associated with the monadic case, and the other two with the left and right arguments of the dyadic case. Verb rank often underlines an aspect of a verb which is already part of its semantic description. For example the monadic verb `$` (*shape of*) always returns a shape list, that is a rank 1 object, and so the dyadic form (*reshape*) is constrained to have a shape list as its left argument. The left verb rank is thus 1, the maximum permitted value. A similar condition applies for the dyadic verb `|.` (*shift*). `%.` (*matrix inverse*) has a verb rank of 2. The three ranks of a verb are obtained as

```
   % b.0
0 0 0
```

```
   %. b.0
2 _ 2
```

The left rank of %. is infinite because matrix division places no rank restriction on its numerator, Apart from verbs such as `plus=.+` which are mere transliterations of primitive verbs, most defined verbs have infinite rank because it would require a dynamic algorithm to work out true verb rank on each application, and infinite verb rank is always a 'safe' assumption. The principal distinction in the present discussion is between rank-zero verbs and non-rank-zero verbs because this leads to an important variant form of composition.

In the phrase

```
   1 2 3(+/@-)2
_1 0 1
```

the leftmost of the two composed verbs *plus insert* is constained to operate at the same rank level as that of *minus*. Informally +/ fuses into - , which has verb rank zero, and since the +/ of a scalar is simply the scalar itself. The manner in which u binds closely with or 'tracks' v in u@v is called **rank inheritance**.

However with *at* (`@:`), in which the verb u does not inherit the rank of v. With `+/@:-` , the linkage between u and v is thus less tight, allowing the +/ to operate at a level which is dictated by the result rank of v rather than by its own verb rank, so that `1 2 3(+/@:-)2` results in -1 + 0 + 1= 0. More generally rank inheritance applies for @, but not for @:.

Rank inheritance distinctions are important in cases such as the *hook* `+/@-mean` where `mean=.+/%#` which is a *fork*. Compositions are resolved as soon as a verb is found to the right of the conjunction on a left-to-right scan, and so the above phrase is equivalent to `(+/@-)mean`, that is add the mean to eaxh item seperately. For the reasons just given +/ operates at rank zero, and so +/ does nothing. On the other hand the phrase `+/@(-mean)` has `-mean` as its rightmost verb which is of infinite rank, so that +/ is free to operate at its natural rank of 1, returning a scalar sum which, in the case of the sum of mean deviations, has the value 0 for all numeric lists. For the composed verb `+/@:(-mean)` the parentheses make no difference to the result, because @:: inhibits rank inheritance. The following sequence sums this up :

```
   (+/@-mean)i.5
```

```
_2 _1 0 1 2
   (+/@(-mean))i.5
0
   (+/@:-mean)i.5
0
   (+/@:(-mean))i.5
0
```

Left and right verbs

Two superficially trivial, but nevertheless important verbs are [(*left*) and] (*right*). The *fork* is another form of composition without explicit symbols. `[,]` transforms a pair of arguments into a two-item list, and the fork `],[` transforms them into the same list in reverse order. These verbs often have to be used in conjunction with @ (*atop*) when, for example, one of the transformations in a dyadic fork uses only one of the two arguments. For example, suppose that the xth. moment about the mean of a numeric list y is required, that is the average of the values of mean-adjusted y raised to the power x :

```
   moment=.mean@:((mdev@])^[)
   2 moment t=.4 5 2 1
2.5
   3 moment t
0
```

In such cases the *passive* conjunction which switches arguments gives a neater solution :

```
   mdev=.-mean
   moment=.mean@:^~mdev
   2 moment t=.4 5 2 1
2.5
```

In the verb `-.&i.` incorporating -. (*less*) is used to remove the items of `i.x` from `i.y` .

```
   12(-.&i.)3       NB. integers from 3 to 11
3 4 5 6 7 8 9 10 11
```

In making this into a defined verb :

```
   to=.-.&i.~,]
   3 to 12
3 4 5 6 7 8 9 10 11 12
```

the *passive* conjunction (~) again switches arguments and] makes the right argument inclusive. Applications of ~ often involve what I choose to call the 'bridge hook' of which an example is `$~$`. The ex-

pression `0($$)a` returns a 0-list whatever the value of a whereas `0($~$)a` switches the 0 to be right argument and thus returns an array of 0s in the shape of a, a much more useful idiom!

The appose conjunction

The *appose* conjunction (&:) provides for & and &. the same release from rank inheritance as @: does for @ . Compare

```
   mean&% 1 2 3 4   NB. mean inherits rank 0
1 0.5 0.333333 0.25
   mean&.% 1 2 3 4  NB. divn followed by inverse, ri =0
1 2 3 4
   mean&:%1 2 3 4   NB. no rank inheritance
0.520833
```

Using `&:` the harmonic mean (that is the reciprocal of the mean of reciprocals) and geometric mean are given by

```
   %mean&:% 1 2 3 4
1.92
   ^mean&:^. 1 2 3 4
2.21336
```

Here is another example of presence/absence of rank inheritance

```
   ]a=.2$<i.2 3           NB. a is a 2-list
┌─────┬─────┐
│0 1 2│1 2 3│
│3 4 5│4 5 6│
└─────┴─────┘
   ,&>a
0 1 2 3 4 5
0 1 2 3 4 5
   ,&:>a
0 1 2 3 4 5 0 1 2 3 4 5
```

When dyadic verbs are concerned, define

```
   ]b=.2$<10+i.2 3  NB. b is another 2-list
┌────────┬────────┐
│10 11 12│10 11 12│
│13 14 15│13 14 15│
└────────┴────────┘
```

`a,&> b` is `a,every b`, that is join item by item and then open the result. However `(a ,&:> b)` means open both items and then join, that is `(a ,&:> b)-:(>a),(>b) :`

```
   (a,&>b);(a,&:>b)
┌──────────┬──────────┐
│ 0  1  2  │ 0  1  2  │
│ 3  4  5  │ 3  4  5  │
│10 11 12  │          │
│13 14 15  │ 0  1  2  │
│          │ 3  4  5  │
│ 0  1  2  │          │
│ 3  4  5  │10 11 12  │
│10 11 12  │13 14 15  │
│13 14 15  │          │
│          │10 11 12  │
│          │13 14 15  │
└──────────┴──────────┘
```

Conjunctions compared

The conjunction *bond* joins a noun and a verb (or a verb and a noun) as in `-&2`

```
   (-&2)4 5 6
2 3 4
```

In the above example the bonded verb is monadic. However bonded verbs can also be dyadic in which case the following rule applies :

```
x m&n y  ↔  m&v*:x y
```

- see E #10 ("Bonding is Power – how interesting") for details.

Five explicit verb-verb compositions have been discussed so far. In each case the leftmost verb u can be thought of as the prime verb, and v as a transformation. The primary decision is whether the required composition is, in mathematical terms, of the form u(v y), in which case use *at* or *atop*, or if it is of the form (v x)u(v y) use *compose* or *appose*, with the additional possibility of *under* to incorporate an inverse transformation. Be careful to distinctions between **valences**, that is monadicity or dyadicity, since the valences of the component verbs are a separate issue from that of their compounds.

at and *appose* are the most fundamental of the composition verbs and follow the rules

	Compound = Monadic	Compound = Dyadic
at	(u@:v)y performs u(v y)	x(u@:v)y performs u(x v y)
appose	(u&:v)y performs u(v y)	x(u&:v)y performs (v x)u(v y)

atop and *compose* obey the rules as *at* and *appose* but invoke rank inheritance. This is discussed in more detail in E #9 ("Pulling Rank")

atop	(u@v)y is (u@:v)"(m v)y	x(u@v)y is (u@:v)"(lr v)y
compose	(u&v)y is (u&:v)"(m v)y	x(u&v)y is x(u@:v)"(mm v)y

where m is the monadic rank of v, lr is a list consisting of its left and right rank vectors, and mm is the list m,m.

under requires that v is monadic

(u&.v)y is v^{-1}(u v y)   x(u&.v)y is v^{-1}(u v y)

@ and & enforce rank inheritance, @: and &: do not. For monadic composed verbs the pairs of conjunctions (@,&) and (@:,&:) are interchangeable. The monadic composed verbs u@v and u&v are thus equivalent, and their effect is also the same as that of issuing u and v directly on a command line, for example

```
   (!i.2 2) ; ((!@i.)2 2) ; ((!&i.)2 2)
┌───┬───┬───┐
│1 1│1 1│1 1│
│2 6│2 6│2 6│
└───┴───┴───┘
```

In terms of transformations :

- u@v Transformation v uses x and y, then u is applied to the result with rank inherited from v.
- u@:v Transformation v uses x and y, then monadic u is applied to the result without rank inheritance.
- u&v u is applied at the same rank level as v following a transformation v of all arguments.
- u&.v monadic u is applied after a transformation v of all arguments, and then the inverse transformation of v is applied.
- u&:v u is applied without rank inheritance following a transformation v involving all arguments.

A *hook* is also a composition of two verbs, for which the two possible results are:

x u (v y) and
y u (v y), so that v is always applied monadically.

Because an absent left argument is given the default value y, the composed verb arising from a hook is always inherently dyadic.

A *fork* is a composition of three verbs, for which the two possible results are:

(x u y) w (x v y) and (u y) w (v y)

For monadic arguments the composed verbs resulting from *atop* and *compose* are identical. The dyadic cases can be thought of as

@	u is applied to a transformation v which uses both arguments;
&	u is applied after a transformation v has been made to both arguments separately;
hook	u is applied following a transformation v to the right argument only.

Subtle Differences

Varying or omitting conjunctions can give rise to substantial differences between superficially similar phrases. This is demonstrated by the following table of algebraic equivalents obtained by applying the above rules:

		Monadic	Dyadic
%^(*hook* }	means	y exp(-y)	xexp(-y)
%@^ and %@:^	mean	exp(-y)	x^{-y}
%&^ and %&:^	mean	exp(-y)	exp(x-y)
%&.^	means	-y	x-y

Final Summary

atop, *at* and *compose*, *under*, *appose* along with 'space *dot*' (inner product – see E #9 "Power Steering extra" for examples) are the most common conjunctions in J, whose rules, along with those of *hook*, *fork* and *cap* can be summarized (r.i. = rank inheritance, sp = space which must precede *dot*) :

		Monadic	Dyadic	
u&v	:	u (v y)	(v x) u (v y)	(v monadic) with r.i.
u&.v	:	v^{-1} u (v y)	v^{-1} (v x) u (v y)	(v monadic)
u&:v	:	u (v y)	(v x) u (v y)	(v monadic) without r.i.
u@v	:	u (v y)	u (x v y)	(u monadic) with r.i.
u@:v	:	u (v y)	u (x v y)	(u monadic) without r.i.

hook	:	y u (v y)	x u (v y)	(v monadic)
fork	:	(u y)w(v y)	(x u y)w(x v y)	(w dyadic)
sp *dot*	:	u y	u (x v y)	(u monadic) without r.i.
[: u v	:	u (v y)	u (x v y)	(u monadic) without r.i.

When *cap* is used to make a hook into a pseudo-fork , `[: g f` is equivalent to `g@:f` rather than `g@f`.

Code Summary

```
int=.>:@i.              NB. integers from 1 to y
maxabs=.>./ @: |        NB. maximum absolute value
normalise=.%maxabs      NB. max abs value in list = 1
mean=.+/%#.             NB. mean of a list
mdev=.-mean             NB. mean deviation
moment=.mean@:^~mdev    NB. xth moment of a list
```

3. My J-oinery Workshop

> Principal Topics : , (*ravel/append*) ,.(*ravel item* , *stitch*) ,: (*laminate*) -. (*less*), " (*rank* conjunction), autostereograms

It is a happy accident that 'join' is not the name of a primitive verb in J because this makes it appropriate to use the word as a generic name for the three primitive verbs *append*, *stitch* and *laminate*. Some analogies with what goes on in the wood-yard seem appropriate.

Suppose I have a pile of planks and I want to stack a second pile alongside. My *append*er (,) takes good care of me so that I do not have to worry about imbalance due to non-matching widths, although it does keep appropriate space clear in order to keep everything in tidy overall order.

```
   ]App=.(2 3$'abc'),(3 4$'defg')
abc
abc
defg
defg
defg
```

Next I use my *stitch*er (,.) when I want to stack two planks side by side, and then join plank to plank in matching pairs. Naturally the stitcher only works when the number of planks in the two piles are equal.

```
   ]St=.(2 3$'abc'),.(2 4$'defg')
abcdefg
abcdefg
```

Next my *laminator* (,:) starts a new pile from two existing piles. Like the appender I do not have to worry about imbalance due to non-matching widths.

```
   ]Lam=.(2 3$'abc'),:(3 4$'defg')
abc
abc


defg
defg
defg
```

The *shape*r ($) starts its work as a laminator and continues as an appender. In this sense the laminator is one of the lowest level (most

primitive) operators in the overall J tool kit, and thus one of the most pervasive verbs even although its explicit usage is relatively small.

Another valuable little tool is my gouge (-.) which helps me dig out unwanted bits like knots

```
   'abcdef'-.'bd'
acef
```

When it comes to finding and rearranging things in the workshop, I reach for one of what I call my monadics. First the *ravel*ler (,) which lays everything out in a line, using my gouge to close up the spaces which may have been put in for overall tidiness :

```
   Ravel=.-.&' '@,
   (Ravel App);(Ravel St);(Ravel Lam)
┌──────────────────┬──────────────┬──────────────────┐
│abcabcdefgdefgdefg│abcdefgabcdefg│abcabcdefgdefgdefg│
└──────────────────┴──────────────┴──────────────────┘
```

Next my spacesaver tool which I affectionately call my 'ri' (short for Ravel Items). This comes into play in two circumstances, one when I want to stack a plank vertically instead of horizontally

```
   ,.'abcd'
a
b
c
d
```

and the other when I want to put my two laminated stacks one on top of the other which I can do in either of two ways

```
   (,.Lam);(,/Lam)
┌───────────┬────┐
│abc abc    │abc │
│defgdefgdefg│abc │
│           │    │
│           │defg│
│           │defg│
│           │defg│
└───────────┴────┘
```

each of which has the effect of reducing the number of piles (rank in J terminology).

```
   ($Lam);($,.Lam);($,/Lam)
┌─────┬────┬───┐
│2 3 4│2 12│6 4│
└─────┴────┴───┘
```

My accountant tells me that ,. (*ravel items)* is also very handy for making lists into columns (easier to tot up, I suppose)

```
   ,.i.5
0
1
2
3
4
```

To complete the picture, my *itemize*r puts a band around all of my timber arrangements. You won't see any visible difference but the shaper shows it clearly :

```
   ($App);($St);($Lam)
┌───┬───┬─────┐
│5 4│2 7│2 3 4│
└───┴───┴─────┘
   ($,:App);($,:St);($,:Lam)
┌─────┬─────┬───────┐
│1 5 4│1 2 7│1 2 3 4│
└─────┴─────┴───────┘
```

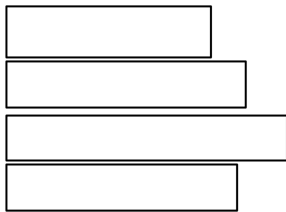
What I haven't told you about is my packing machine known as the '*box*er' which opens up many possibilities for joining at the package level which will be exploited elsewhere.

```
   <"1 App
┌────┬────┬────┬────┬────┐
│abc │abc │defg│defg│defg│
└────┴────┴────┴────┴────┘
   <"1 St
┌───────┬───────┐
│abcdefg│abcdefg│
└───────┴───────┘
   <"1 Lam
┌────┬────┬────┐
│abc │abc │    │
├────┼────┼────┤
│defg│defg│defg│
└────┴────┴────┘
   <"2 Lam
┌───┬────┐
│abc│defg│
│abc│defg│
│   │defg│
└───┴────┘
```

The last step is a demonstration of the tools in use. The editor was not sympathetic to my idea of distributing a set of planks with Vector, and so I have to fall back on a diagram from the book “How the Mind Works” by the American psychologist Stephen Pinker, He uses this diagram to explain how autostereograms are constructed. The basis of the diagram is that in the short lines two 4s have been removed; in the long lines two Xs havd been inserted between 3 and 4. The diagram is itself an autostereogram - look at these two areas and use stereogram viewing techniques and you should seeve a raised and a recessed rectangular block respectively.

```
123456789012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890
1234567890123456789012356789012356789012345678901234567890
1234567890123456789012356789012356789012345678901234567890
1234567890123456789012356789012356789012345678901234567890
1234567890123456789012356789012356789012345678901234567890
1234567890123456789012356789012356789012345678901234567890
1234567890123456789012356789012356789012345678901234567890
123456789012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890
12345678901234567890123567890123567890123512345678901234567890
12345678901234567890678901235678901235678912345678901234567890
12345678901234567890012356789012356789012312345678901234567890
12345678901234567890567890123567890123567812345678901234567890
12345678901234567890901235678901235678901212345678901234567890
12345678901234567890356789012356789012356712345678901234567890
123456789012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890
```

A blueprint for the diagram is



Here is how the appender, stitcher and gouge realise this blueprint

```
t=.'123X4567890'
u=.t-.'X'
v=.u-.'4'
a=.21 20$u
```

```
b=.3 40$u
c=.(6 18$v),.6 20$u
d=.(6 22$v),.6 20$u

a,.b,c,b,d,b
```

Another possibility is to use 'joinunequal' with the following blue-print

'joinunequal' laminates and then undoes the extra dimension, and follows with the addition of a routine to remove blank rows :

```
A=.3 70$u
B=.(6 30$u),.6 38$u-.'4'
C=.(6 30$u),.6 42$t

rbr=.#~ -.@:(*./"1@:=&' ')
juneq=.rbr@,/@,:          NB. join unequal
```

The Pinker diagram is then obtained by

```
A juneq B juneq A juneq C juneq A
```

Ah well, time to close up the workshop for another day.

Code Summary

```
juneq=.rbr@,/@,:                 NB.join char matrices of diff
shapes
  rbr=.#~ -.@:(*./"1@:=&' ')     NB.remove blank rows
```

4. Parallel Joins

Principal Topics : , (*ravel/append*) ,.(*stitch*) ,: (*laminate*) " (*rank comjunction*), ` (*tie*) @. (*agenda*) ;(*raze*) scalarisation, rectangularity, fill characters, bit to integer conversion, gerund.

By 'programming for parallelism' I mean the process of, having worked something out for simple data, contriving that the relevant verbs also work on data with more complicated list structures, and thus carry out their actions on different sets of data in parallel. In the case of scalar verbs there is often nothing to do, thus 1 2 3 + 4 5 6 carries out additions in parallel, and 2 + 3 4 5 extends 2 to 2 2 2 prior to doing the parallel additions. It is often desirable to upgrade more complicated verbs so that they behave like scalar verbs. There are several mechanisms available for achieving parallelism in J, and it helps to have clearly in mind the various levers which are available to be pulled. Some readers may relate to the experience of 'going parallel' by pulling hopefully at some of these levers, until eventually the desired result emerges (or in some cases, not!) If you have never had this heuristic experience, and always do the right thing first time, then stop reading at this point - J-ottings can be regarded as a confessional for imperfect J programmers!

The first mechanism for parallelism is rank control, one of the outstanding features of J. As an example, start with *append* which operates at a minimum rank of 1 :

```
   1 2,3 4 5
1 2 3 4 5
```

Provided that the *append*ing is to take place at equal levels of the arguments, it does not matter what argument is given to the rank conjunction provided that it is a strictly positive integer :

```
   1 2,"(17)3 4 5
1 2 3 4 5
```

Rank control is about operating at different levels of the left and right arguments, for example 'scalar to list' or 'scalar to list of lists' :

```
   1 2,"(0 1)3 4 5  NB. join scalars to list
1 3 4 5
2 3 4 5
   1 2,"(1 0)3 4 5  NB. join list to scalars
1 2 3
1 2 4
1 2 5
```

Sometimes an operation cannot be performed without length equalisation using a fill character as in

```
   4 5 6 7,"(1 2)i.2 2      NB. join list to list of lists
4 5 6 7
0 1 0 0
2 3 0 0
```

Length equalisation and fill characters are provisions rather than controls, and are present because the language designers wisely judged that they are often greatly preferable to length errors.

Parallelism for *append* means creating an upgraded verb which

(a) joins several lists to the same list of scalars;
(b) joins the same list to several lists of scalars; and
(c) join matched pairs of lists and scalar lists.

This requires **scalarisation**, which is the second major control. It is tempting to equate scalarisation with *box*, but this is not quite accurate since *link* (;) also performs scalarisation. The results of scalarisation are non-simple scalars, which in general can only take part in structural operations such as appending, shifting, rotating and transposing, which do not involve looking at their content. Any operations on contents require an *open*, and, in the words of the help file, "opened atoms are brought to a common shape". To put this in another way, open always forces rectangularity which is sometimes what is wanted, at other times not.

To return to the requirements for extended *append*, consider requirement (a). `1 2; 1 3` is a list of two boxed scalars which must be opened before the concatenation of each to a simple list such as 7 8 9 . Following the remarks in the previous paragraph, no concatenation can take place until the two boxed lists are opened, whereupon the result is 'list joined to list', that is *append* at rank 1 :

```
   (>1 2;1 3),"(1)7 8 9
1 2 7 8 9
1 3 7 8 9
```

There is a snag, though, namely that if the two lists on the left are of unequal length the rectangularity rules will enforce an unwanted fill character between the first pair of joined lists :

```
    (>1 2;4 5 6),"(1)7 8 9
```

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

The following workaround

```
   (1 2;4 5 6),every<7 8 9
1 2 7 8 9 0
4 5 6 7 8 9
```

still forces a fill character, and demonstrates further that explicit opens always expose the user to this possibility. If this is not acceptable then lists must be boxed by using *under*(&.) rather than *compose* :

```
   (1 2;4 5 6),each<7 8 9
┌─────────┬───────────┐
│1 2 7 8 9│4 5 6 7 8 9│
└─────────┴───────────┘
```

This example underlines the fact that u&.v is more subtle than 'do v, apply u, undo v' (in this instance '*open*, *append*, *box*'), because if this were so, there would be an embedded fill character present in the result.

For requirement (b), suppose the list 1 2 has to be joined to the separate lists 4 5 and 7 8 9. The foregoing discussion suggests

```
   (<1 2),each 4 5;7 8 9
┌───────┬─────────┐
│1 2 4 5│1 2 7 8 9│
└───────┴─────────┘
```

Finally requirement (c), which, perhaps surprisingly, turns out to be the simplest of the three because there is no need for scalar expansion to be invoked by an explicit *box* :

```
   (1 2;2 4 6),each 4 5;7 8 9
┌───────┬───────────┐
│1 2 4 5│2 4 6 7 8 9│
└───────┴───────────┘
```

Using the verb `je=.,each` standing for 'join each' the three requirements (a), (b) and (c) are dealt with by

```
  (1 2;4 5 6) je< 7 8 9
  (<1 2) je 4 5;7 8 9
  (1 2;2 4 6) je 4 5;7 8 9
```

respectively, and the basic case, that is using `je` to join two unscalarised lists, is :

```
     1 2 (je&<) 3 4 5
┌─────────┐
│1 2 3 4 5│
└─────────┘
```

Thus `je` can deal with all possible cases, but only if responsibility for boxing unboxed arguments falls on the user. It would be nice if this decision could be automated. The first of the verbs below tests whether an item is boxed, and the second uses a gerund to *box* it if it is unboxed, otherwise leaves it alone :

```
   unboxed=.-:>               NB. 0=boxed, 1=unboxed
   box=.]`< @.unboxed         NB. box if unboxed, else do nothing
```

so, given that all result lists are required to be boxed, the verb

```
   JE=.(,each)&box  NB. je having boxed any unboxed arguments
```

addresses the simple case and cases (a) to (c) :

```
  1 2 JE 3 4 5                      NB. equivalent to append then
box
  (1 2;4 5 6) JE 7 8 9              NB. cf. 1 2 + 3
  1 2 JE 4 5;7 8 9                  NB. cf. 1 + 2 3
  (1 2;2 4 6) JE 4 5;7 8 9          NB. cf. 1 2 + 3 4
```

Other possible requirements can be addressed by combining `JE` with rank control. For example to join each scalar element in the right argument to a list as left argument

```
    1 2 3 JE"(1 0) 4 5
┌───────┬───────┐
│1 2 3 4│1 2 3 5│
└───────┴───────┘
```

*box*es individual scalars in the right argument so that they behave as two one-item lists, which also happens in the case of multiple lists :

```
     1 2 JE"(0 1) 3;4 5 6;7 8
┌───┬───────┬─────┐
│1 3│1 4 5 6│1 7 8│
├───┼───────┼─────┤
│2 3│2 4 5 6│2 7 8│
└───┴───────┴─────┘
```

In the next two cases using *ravel* on the left gives lists of all possible joins, only in a different order..

```
   ,(1 2;2 4 6)JE"(1 0) 4 5;7 8 9
┌───────┬─────────┬─────────┬───────────┐
│1 2 4 5│2 4 6 4 5│1 2 7 8 9│2 4 6 7 8 9│
└───────┴─────────┴─────────┴───────────┘
   ,(1 2;2 4 6)JE"(0 1) 4 5;7 8 9
┌───────┬─────────┬─────────┬───────────┐
│1 2 4 5│1 2 7 8 9│2 4 6 4 5│2 4 6 7 8 9│
└───────┴─────────┴─────────┴───────────┘
```

The same principle can be applied to a verb such as `btoi` :

```
   btoi=.# i.@#           NB. converts bits to integer list
   btoi&> 0 1;1 0 0 1 0   NB. accept fill characters
1 0
0 3
   btoi each 0 1;1 0 0 1 0 NB. returns boxed lists
┌─┬───┐
│1│0 3│
└─┴───┘
```

With character strings it is natural to think in terms of suffixing and prefixing in the obvious literary sense. It is instructive to see why some plausible ideas for adding different inflections to the same verb root fail. First

```
   (<'post') ,.'s';'ing';'ed'
┌────┬───┐
│post│s  │
├────┼───┤
│post│ing│
├────┼───┤
│post│ed │
└────┴───┘
```

This goes part of the way but there appear to be fill characters present even in the absence of an explicit *open*. However, the ravel of the above expression

```
   ,(<'post') ,.'s';'ing';'ed'
┌────┬─┬────┬───┬────┬──┐
│post│s│post│ing│post│ed│
└────┴─┴────┴───┴────┴──┘
```

shows that the fill characters are purely for the necessities of display, but loses the division into three separate words. Replacing *ravel* with *raze* is even worse :

```
   ;(<'post'),.'s';'ing';'ed'
```

```
postspostingposted
```

An attempt to remove boxes preserves the words as entities but at the expense of introducing fill characters :

```
   ;"1(<'post') ,"1 0 's';'ing';'ed'
posts
posting
posted
```

But do not worry – help is at hand through JE :

```
   'post' JE 'ing'
┌───────┐
│posting│
└───────┘
   'post' JE 's';'ing';'ed'
┌─────┬───────┬──────┐
│posts│posting│posted│
└─────┴───────┴──────┘
   ('post';'bow')JE 'ing'
┌───────┬──────┐
│posting│bowing│
└───────┴──────┘
   ('post';'bow')JE 's';'ing'
┌─────┬──────┐
│posts│bowing│
└─────┴──────┘
```

The third mechanism for parallelism is the join alternatives, namely *stitch* and *laminate*. *stitch* (which used to have the more descriptive name of 'append items') provides an alternative for rank control, but has more restrictions than with append due to the constraints of length compatability.

```
   1 2,"(0)3 4
1 3
2 4
```

```
   1 2,.3 4
1 3
2 4
```

laminate does its join by introducing a new 2-list which means that it is less easy to find simple direct alternatives to *append*. Also, unlike *stitch*, length equalisation and fill characters can be tolerated. Both of these points are illustrated in the example below :

```
       >1 2 ,&box 3 4 5
1 2 0
3 4 5
```

```
   1 2 ,: 3 4 5
1 2 0
3 4 5
```

The verbs `JES` and `JEL` are identical to `JE` except that *append* has been replaced by *stitch* and *laminate* respectively. It is helpful to see the differences by comparing results side by side. Two informal observa-

tions are first, that the boxes now contain lists of lists rather than lists. In loose terminology, stitch adds things on the right, laminate adds them on the bottom.

```
   JES=.,.each&box                  JEL=.,:each&box

   1 2 JES 3 4                      1 2 JEL 3 4
┌───┐                           ┌───┐
│1 3│                           │1 2│
│2 4│                           │3 4│
└───┘                           └───┘

   1 2 JES 3 4;5 6              1 2 JEL 3 4;5 6
┌───┬───┐                       ┌───┬───┐
│1 3│1 5│                       │1 2│1 2│
│2 4│2 6│                       │3 4│5 6│
└───┴───┘                       └───┴───┘

   (1 2;3 4)JES 5 6             (1 2;3 4)JEL 5 6
┌───┬───┐                       ┌───┬───┐
│1 5│3 5│                       │1 2│3 4│
│2 6│4 6│                       │5 6│5 6│
└───┴───┘                       └───┴───┘

   (1 2;3 4)JES 5 6;7 8            (1 2;3 4)JEL 5 6;7 8
┌───┬───┐                       ┌───┬───┐
│1 5│3 7│                       │1 2│3 4│
│2 6│4 8│                       │5 6│7 8│
└───┴───┘                       └───┴───┘
```

In summary, JE uses scalarisation via *box* and *open* to achieve fundamental parallelism such as available in primitive scalar verbs. By adding rank control and introducing the variants JES and JEL an even greater variety of parallelism opportunities is possible.

Code Summary

```
je=.,each                 NB. join each
JE=.(,each)&box           NB. append each
JES=.,.each&box           NB. stitch each
JEL=.,:each&box           NB. laminate each
btoi=.# i.@#        NB. converts bits to integer list
unboxed=.-:>        NB. 0=boxed, 1=unboxed
box=.]`< @.unboxed  NB. box if unboxed, else do nothing
```

5. Conjugacy and Rank

Principal Topics : b. (*basic characteristics*), verb ranks, conjugations

Rank is of enormous importance in J. There are two types of rank, noun rank and verb rank. The former can be applied to any object in the J universe, and is the 'tally-of-the-shape', or in J :

```
   nrank=.#@$
```

Verb rank is somewhat more subtle, and is of even greater importance in understanding J language constructs. Henry Rich put things with admirable conciseness in 'J for C Programmers':

"If you don't know the rank of a verb, you don't know the verb!"

It is not just primitive verbs which possess rank, user defined verbs also possess it, and so do compound verbs, that is verbs formed by combining or adapting primitive verbs using appropriate conjunctions and adverbs. Moreover, there is no excuse for J programmers not knowing any relevant verb ranks, since these are immediately obtainable for any verb by using the b. adverb with right argument 0 :

```
   %. b. 0                    NB. primitive
2 _ 2
   +/ b. 0                    NB. compound (adverbial)
_ _ _
   +.@*: b. 0                 NB. compound (using conjunction)
0 0 0
   mean=.+/ % #
   mean b. 0                  NB. user defined
_ _ _
```

Within the J literature I am not aware of any explicit categorisation of verbs by rank, which is why much of this article consists of an appendix which does just this. I hope that some readers may find this appendix helpful, perhaps long after this preamble has been forgotten.

As postings on the J conference testify, the distinction between @ and @: is a stumbling block in the path of almost every new learner; what I hope to show here is how this block can be rapidly and effectively smoothed by a conscious endeavour to understand verb rank.
In general verbs have three unforced ranks, which are, in the order given in the dictionary definitions:

monadic rank, left rank, right rank

The word 'unforced' acknowledges that verbs can have their basic ranks changed to an explicit value by using the *rank* conjunction " . Classification by rank has some affinity with verb conjugations in the classical languages, and although this analogy should not be pushed too far, it seems reasonable to think of verbs as being grouped by conjugations along lines such as the following :

1. Pure scalar verbs, all three rank vector items = 0
2. Monadic list verbs, all rank vector items = 1
3. 'Irregular' verbs with hybrid ranks, mostly specialised
4. Pure structural plus' verbs, ranks are _ 1 _
5. Verbs with all ranks infinite.

A verb rank of, say, 1 should not be read as meaning that only objects of noun rank 1 are acceptable as arguments, but rather that all arguments will be processed as assemblages of rank 1 objects. There is an analogy with the operation of a clinic which has a large and motley assembly in the waiting room, from which patients may either be called one by one (processing at rank 0), or by families (that is lists, rank = 1) or the whole lot may be taken together as a single block of humanity (rank infinite). The analogy goes further than this, because the method of calling is independent of what happens once the surgery is entered, in other words, rank comes first, semantics later. Immediately following the above quotation in Henry's book is an explanation in detail of how rank is applied in particular cases, that is how rank and semantics are married together in terms of cell and frame selection. However, focussing on ranks shows that, apart from the 'special algorithm' verbs, J verbs can be grouped into quite a small number of categories.

A general problem in categorising verbs by rank is that of deciding when the monadic and dyadic forms are sufficiently related to each other to justify retaining this association even if they possess different rank vectors. At one extreme monadic > (*open*) and dyadic > (*greater than*) are completely unrelated in meaning, even although all their rank vector items are zero. On the other hand verbs such as monadic #: (*antibase2*) and dyadic #: (*antibase*) have a strong semantic association although formally they belong to different conjugations.

Scalar verbs which form the first conjugation all have rank 0 and are the 'most penetrating', meaning that unless otherwise modified by the rank conjunction they operate at the lowest cell levels. The ordinary

arithmetic verbs are thus all of rank 0. Second conjugation verbs are all monadic and operate at the level of lists, such as i. (*integers*) and #: (*base 2*). The conjugations range from the most penetrating in the first conjugation through to those of the 4th and 5th conjugations which handle objects at a macro level. In between at the third conjugation are a set of verbs which are the counterpart of irregular verbs in natural language grammars

Conjugation 2 contains monadic verbs only which make no sense other than when applied to lists, for example integers i. (*integers*) and ;: (*word formation*).

Conjugation 3 contains irregular verbs some of which are verbs such as %. (*matrix inverse / divide*), which are of great value to a minority of J users, and of little or no interest to the rest.

Conjugation 4 consists of the structural verbs, all of whose right ranks are infinite, meaning that their arguments are processed in their entirety as single objects.

Conjugation 5 consists of verbs whose ranks are all infinite.

Compound Verbs and Rank

Compound verbs formed with conjunctions and adverbs possess rank vectors in the same way as primitive verbs, and Henry's maxim applies as much to compound verbs as to primitives. The rules which govern the rank vectors of compound verbs are somewhat complex, and it is generally best from a pragmatic point of view to use the `b.0` test to confirm them. It is worth noting though that compounds arising from *atop*, *appose* and *bond* have rank vectors _ _ _ which is a consequence of their not forcing rank inheritance (see E #2 "A Composition on Composition") in which the following table appears).

at and *appose* are the most basic composition verbs and follow the rules

	Compound Monadic	Compound Dyadic
at	u@:v)y means u(v y)	x(u@:v)y means u(x v y)
appose	(u&:v)y means u(v y)	x(u&:v)y means (v x)u(v y)

atop and *compose* invoke rank inheritance and obey the rules

atop (u@v)y is (u@:v)"(m v)y   x(u@v)y is (u@:v)"(lr v)y
compose (u&v)y is (u&:v)"(m v)y   x(u&v)y is x(u@:v)"(mm v)y

where m is the monadic rank of v, lr is a list of the left and right rank vectors, and mm is the list m,m.

under requires that v is monadic

(u&.v)y is v^{-1}(u v y)   x(u&.v)y is v^{-1}((v x)u(v y))

Compare

```
   2 3(+/@%)4 5            NB. (+/@%)b.0 is 0 0 0
0.5 0.6
   2 3(+/@:%)4 5           NB. (+/@:%)b.0 is _ _ _
1.1
```

Using the clinic analogy, in the first case the 'patients' are called in matched pairs, and each is divided to give result 0.5 0.6. the patients are called in two blocks (left and right), which are divided as blocks to give 0.5 0.6, and then *plus-insert* is applied to this single block to give 1.1. Informally the difference between *atop* and *at* is that u and v are more closely bound in the former. The name *atop* is apt as it gives a picture of two creatures, one piggy-backing on the other, and thereby fusing to make a tight compound before any considerations of data come into play. *at* conveys less well the way in which infinite rank produces verb sequencing, that is "u following v", is a perhaps a more pictorial way of thinking about it.

Another example is

```
   2 (#.@^.)2 5 6          NB. (#.@^.)b.0 = 0 0 0
1 2.32193 2.58496
   2^.2 5 6
1 2.32193 2.58496
```

that is the #. has no effect. However

```
   2 (#.@:^.)2 5 6         NB. (#.@:^.)b.0 = _ _ _
11.2288
   2 #.1 2.32193 2.58496
11.2288
```

that is the left argument applies to both u and v

Here are yet more examples illustrating *atop* and *at* :

Compare `({:@*:)i.2 3` and `({:@:*:)i.2 3` . As above, the only rank which matters in the first case is that of *: (*square*) which is 0. The semantic rule which extends *tail* when applied to scalars (i.e. at rank 0) is 'no change' and so the final result in the first case is

```
0   1   4
9  16  25
```

In the second case, *tail* with infinite rank means the *tail* of the list of lists arising from squaring, which gives the result list 9 16 25.

Next compare

`({:@#.)2 3$1 0 1 1 1 0` and `({:@:#.)2 3$1 0 1 1 1 0`

Here the rank of the rightmost verb is 1, and so in the first case *tail* is applied to each of the *antibase2*s of the two three-lists which make up the right argument, giving a final result 5 6. In the second case, *tail* applies to the entire result of #. which is a two-list, so that the final result is 6.

Compare `2 3 4(+/@%.)m=.?2 3 3$10` and `2 3 4(+/@:%.)m`
This is a dyadic example, in which the rank of the rightmost verb is 2. m represents two sets of 3 by 3 linear equations with the same right hand side 2 3 4. In the first case the result is the sums of each of the three solution sets (x1+ y1+ z1, x2+ y2+ z2) . In the second case summation applies to the two-list of solution values, so that the final result is (x1+ x2, y1+ y2, z1+z2) .

... and so I could continue.

Adverbs
Again, pragmatically it is best to use the `b.0` test. J adverbs are relatively few in number and are all monadic in the sense that each qualifies a single verb, although the compounds arising may be either monadic or dyadic. The following is a table of them :

	Monadic	Dyadic
/	*Insert*	*Table*
/.	*Oblique*	*Key*
\	*Prefix*	*Infix*
\.	*Suffix*	*Outfix*
~	*Reflexive*	*Passive*
{.	*Item Amend*	*Amend*

b. *Basic Characteristics* *Boolean*

Regardless of the verb which they qualify *infix*ed and *suffix*ed verbs always have left rank 0 reflecting the fact that the semantics require an integer as left argument for the compound verb. On the other hand the rank of *reflexive* is infinite because rank depends on arguments for the compound the ranks of *passive* depend on those of the qualifying verb as in

```
   (+~) b.0
_ 0 0
   (#.~) b.0
_ 1 1
   (%.~) b.0
_ 2 _
   (i.~) b.0
_ _ _
```

Appendix : Verb Ranks

1st Conjugation, rank vector = 0 0 0

Logicals : (monadic) -. (*not*)
(dyadic) = ~: (*not equals)* < <: (*less than or equals)*
> >: (*greater than or equals)* +: (*not or)* *: (*not and)*
Arithmetics : (monadic) -. (*1 minus)*
(monadic and dyadic) + - * % ^ (*power)* ^. (*logarithm)*
<. (*floor/lesser of)* >. (*ceiling/greater of)*
| (*modulus/residue)* ! (*factorial/out of)*
%: (*square root/root)*
+. (*real/imaginary /GCD)* *. (*length/angle /LCM)*
Algorithmics : (monadic) p: (*ith. prime*)
(dyadic) ? (*roll/deal)* j. (*imaginary/complex)*
o. (*pi times/circle function)* r. (*angle/polar)*
q: (*prime factors/prime exponents)*

2nd Conjugation, monadic, list oriented

i. (*integers)* { (*catalog)* ;: (*words)*
#. (*base 2)* ". (*do)* #: (*anti-base 2)*
p. (*roots*) A. (*Anagram Index*) C. (*Cycle-Direct*)

3rd Conjugation, irregular (with ranks)

(monadic) %.(*matrix inverse :* 2)
(dyadic) #: (*antibase :* 1 0) p. (*polynomial :* 1 0)
{ (*from :* 0 _) A.(*Anagram :* 0 _) %. (*matrix divide :* _ 2)

C. (*permute :* 1 _)

4th Conjugation, dyadics with left rank=1, right rank=infinite

$ (*reshape)* |. (*shift)* |: (*transpose)* # (*copy)*
{. (*take)* }. (*drop)* ": (*format)* {:: (*fetch*)

5th Conjugation, all ranks infinite

(monadic) = < (*box)* ~. (*nub)* ~: (*nub sieve)*
{: (*tail)* }: (*curtail)* #: (*base)* $ (*shape)*
|. (*rotate)* |: (*transpose)* # (*tally)*
{. (*head)* }. (*behead)* ": (*default format)* L. (*level of)*
(dyadic) -. (*less)* -: (*match)* i. (*index of)*
". (*numbers)* E. (*member of interval)*
(monadic and dyadic) , (*ravel, append)* ,. (*ravel items/stitch)*
,: (*itemize/laminate)* /: (*grade up/sort)* \: (*grade down/sort)*
; (*raze/link)* e. (*raze in/member in)*
$. (*sparse)* $: (*self-reference)* [(*same/left)*] (*same/right)*
s: (*symbol*) u: (*unicode*) x: (*extended precision*)
constant functions, that is _9:, _8:,..., 0:, 1:, 2:, ... 9:, also _: (*infinity*)

6. Punctuation and Rank

Principal Topics : |. (*shift*) /: (*grade up*) [(*left*)] (*right*) " (*rank* conjunction) , (*append*) ; (*link*) ,. (*stitch*) [: (*cap*) rank inheritance, statement separator, mood, transitivity, commutativity, trains, binding

In Vector Vol. 24 nos. 2 &3 pp. 114-121, Neville Holmes gives useful tables which categorise the structural primitive verbs of J according to their function – 'what they do' as opposed to the 'how they do it'. Accurate use of such verbs requires strict adherence to rules regarding two intimately related quantities, namely punctuation and rank, which leads to a further categorisation of a different kind given here as Appendix 1. What follows in this article is the rationale underlying this classification.

Punctuation

Lynne Trusse created a best-selling book on the art of punctuation, famously drawing its title from the story of the panda which, after visiting a restaurant "eats shoots, and leaves", a plausible option allowing for a touch of anthropomorphism, and one which might not cause too great a disturbance to other diners. If on the other hand the panda "eats, shoots, and leaves" the effect is likely to be very different. Much has been made of the manner in which the constructs of J were inspired by and derived from the parts of speech of ordinary language grammar, nouns, verbs and so on, despite which little reference is made to punctuation. While analogies should not be pushed too far – e.g. unlike 'shoots', there can never be any verb/noun ambiguity for primitive J objects – the explicit consideration of punctuation by parentheses and space, as well as implicitly through *hook* and *fork*, can be helpful in writing and understanding expressions. Compare :

```
   ((i.#)t);(i.@#)t=.'abcde'
┌─┬─────────┐
│5│0 1 2 3 4│
└─┴─────────┘
```

can be tempting to think of conjunctions such as @ (*atop*) as punctuators. However the role of @ in the above is that of a neologiser, that is, it constructs a new compound verb, call it 'index-tally' (or perhaps even 'indally'), operating in scalar fashion on the items of the object to its right. The operational details of such verbs leads naturally to consideration of one of the most subtle of all J concepts, namely *rank*.

Consider three compound verbs which differ only in punctuation, that is the placing of the parentheses :

```
      v1=.>:@i.@#
      v2=.>:@(i.@#)
      v3=.(>:@i.)@#
```

The effect of all three verbs is the same, that is they are semantically equivalent as demonstrated by

```
   (v1 t);(v2 t);(v3 t=.'abcde')
┌─────────┬─────────┬─────────┐
│1 2 3 4 5│1 2 3 4 5│1 2 3 4 5│
└─────────┴─────────┴─────────┘
```

This raises general questions such as : (1) for any verbs a b c to which (if either) of the two forms a@(b@c)and (a@b)@c is a@b@c necessarily equivalent, and (2) is a@(b@c) equivalent to (a@b)@c, - in mathematical terms, is the conjunction @ **associative**. Intuitively it should not be, for the same sort of reason that "eats, shoots and leaves" has a different meaning from "eats shoots, and leaves". The scope rule for conjunctions is that they bind closely on the right, which means that it is (a@b)@c which is equivalent to a@b@c, which is of course is guaranteed by the J interpreter. Using conjunctions is analogous to coining new verb-names in English, so that the meaning of v2 is 'increment-(index-tally)' as opposed to v3 which is '(increment-index)-tally' ('incrindally' as opposed to 'increxally'!).

In learning J it takes a degree of mental adaptation to grasp the idea of a compound verb such as 'index-tally' let alone triple compounds like v2 and v3, and also to appreciate that the meaning of 'index-tally' is not "index then tally". This difference is demonstrated by :

```
   (|.@*:)t=.1 2 2 3       NB. rotate-square
1 4 4 9

   (|.@:*:)t         NB. rotate-following-square
9 4 4 1
```

which suggests that the J terminologies "a atop b" and "a at b" are rendered more comprehensibly in pseudo-English as 'a-b' and 'a-following-b'. The compound verb 'rotate-square' ('roquare'?) has rank zero because it takes on the rank of its rightmost component, a property which, for obvious reasons, is called rank inheritance (see E #2 "A Composition on Composition"). However, in 'rotate-following-square' the colon in @: can be thought of as signalling a pause in which rank is readjusted before the left hand verb is executed.

The nature of the arguments which can be presented to a conjunctionally compounded verb depend on the arguments presented to its rightmost verb, and so returning to v1, v2 and v3, all of these require an argument acceptable to *tally*. This can be any J object since all J objects are fundamentally lists and so can be tallied at their topmost level. Also since the three verbs are to be executed in right to left succession it would seem, superficially at least, to make no difference how they are parenthesised as the transformed data is 'passed down the line' from right to left. However, as 'rotate-square' shows, rank inheritance has to be taken into account in the general case.

Rank Lists

Verbs can be categorised according to their rank properties in a manner comparable to conjugation in classical language grammar (see Appendix to E #5 "Conjugacy and Rank"). Every verb has a rank list, viz.

monadic rank   left rank   right rank

which can always be explicitly obtained by applying the basic characteristics adverb b. and using 0 as right argument of the resulting verb. Rank can be infinite, and most verbs of infinite rank are structural, meaning that, like *box*, they are designed to operate on their argument or arguments as a whole, that is, they do not 'penetrate' the outer shells of objects. *Grade-up* is a useful illustration of the notion of **infinite rank** because however large the rank of its argument, it orders objects at the next lowest rank level, thus

```
   /:i.2 10 20 30 40             NB. grade up two 4 dimnl ob-
jects
0 1
   /:i.5 10 20 30 40      NB. grade up five 4 dimnl objects
0 1 2 3 4
```

Infinite is the default verb rank, which is also the rank of all but the simplest user-defined verbs, since the interpreter could potentially be forced to perform exhaustive and unproductive effort to work out the de facto rank, and so it makes the 'safe' assumption of infinite. However this can be over-ridden by explicit use of the rank conjunction as in

```
   mean=.+/ % #
   mean0=.(+/%#)"0        NB. scalarised mean
   (mean i.5);(mean0 i.5)
┌┬─────────┐
```

```
│2│0 1 2 3 4│
└─┴─────────┘
   (mean b.0);(mean0 b.0)
┌─────┬─────┐
│_ _ _│0 0 0│
└─────┴─────┘
```

Rank Inheritance

Returning to 'rotate-square' whose rank list is 0 0 0, although rotate is a rank 1 verb, rank inheritance forces rotation at rank 0 (that is, equivalent to an explicit "0) and so it inherits a list of rank 0 objects (scalars) each of which has to be treated as a list, with the result that is does nothing. However, if rank is not inherited as with `|.@:*:` , then rotation applies to the list of squares as a single entity of rank 1.

The equivalence of a@(b@c)and (a@b)@c (i.e. associativity) depends on the rank of the inheriting verb being no greater than that of the giving verb, something which will certainly take place if a, b and c are all rank 0 verbs, but which has to be examined in terms of verb rank properties when this is not the case. Rank inheritance from higher to equal or lower creates no problems as in

```
   (>:@i.)6            NB. rank 0 inherits rank 1
1 2 3 4 5 6
```

However, compare

```
   (/:@>:)7 3 5        NB. rank infinite inherits rank 0
0
0
0
```

in which each of the three incremented values is upgraded separately, with

```
   (>:@/:)7 3 5     NB. rank 0 inherits rank infinite
2 3 1
   (/:@|:)i.2 3     NB. both ranks infinite ..
0 1 2
   (|:@/:)i.2 3     NB. .. but the result is different
0 1
```

The next two examples involve verbs of equal ranks, again there is no inheritance issue, although changing the order of the verbs gives a different result because the *grade up* of a transposed matrix is not the same as the *transpose* of a *grade up* of the original matrix.

```
   (/:@|:)i.2 3     NB. both ranks infinite ..
```

```
0 1 2
   (|:@/:)i.2 3     NB. .. but the result is different
0 1
```

Mood, Transitivity and Commutativity

J verbs are restricted to the imperative mood apart from the verb 'to be' (copula). Mood is independent of transitivity, meaning that a verb is either monadic (intransitive) or dyadic (transitive). For transitive verbs the arithmetic **commutativity** of say + means that 2 + 3 is in every respect equal to 3 + 2. However when a computer does addition it is impossible for both arguments to be fetched simultaneously, and so, analogously with transitive verbs in English for which the subject is in some sense 'stronger' than the object, the left argument of dyadic verbs binds more strongly than the right. This becomes apparent when repetition is invoked by the *power* conjunction. Thus 2+^:(2)3 means add 2 twice to 3 (answer 7), as opposed to add 3 twice to 2 (answer 8).

Pseudo-punctuation

Returning to punctuation, there are three verbs, all of infinite rank, which can be thought of as providing a 'pseudo-punctuator' role for verbs. These are , (*append*) ; (*link*) and ,. (*stitch*) In each example below the pseudo-punctuator is the middle tine of a fork, and the sum and difference of a list can be 'pseudo-punctuated' in the following ways :

```
   5 4(+,-)2 0      NB. sums joined to differences
7 4 3 4

   5 4(+;-)2 0      NB. boxed sums, boxed differences
┌───┬───┐
│7 4│3 4│
└───┴───┘
    5 4 (+,.-)2 0   NB. sums, diffs as lists of lists
7 3
4 4
```

The verb [(*left*), also of infinite rank, provides pseudo-punctuation in the form of a **statement separator** :

```
   a=.2 [ b=.3
   a,c
2 3
```

which works because the above line is essentially

```
   a=.2[3
7
```

with the assignment to b taking place 'in passing'.

Square brackets can sometimes give rise to what looks orthographically as 'verb parentheses' when a transformation has to be applied to one argument only :

```
   2(*:@[+])3
7
   2([+*:@])3
11
```

Arguably these phrases would have been written more clearly as

```
   2((*:@[) + ])3  and   2([ + (*:@]))3
```

Redundant punctuation

As the above example shows, redundant parentheses can be invaluable in clarifying the meanings of tacit definitions, although, like all good things, it can be overdone, and too many parentheses can sometimes be just as confusing as too few. Once a string of J symbols exceeds about seven characters even an expert reader's eyes begin to glaze over, as with for example :

```
   lengths=.<"1 @:,.3&":@:i.&' '"1
```

An example of using lengths might help to clear the fog :

```
   lengths >'Florida';'California';'Alaska'
┌──────────┬─────────────┬──────────┐
│Florida  7│California 10│Alaska   6│
└──────────┴─────────────┴──────────┘
```

Things become a little clearer if lengths is rewritten with some parentheses and an explicit space within the hook:

```
   lengths1=.<"1 @: (,. ((3&":)@:(i.&' ')"1))
```

But following the seven character rule it would have been even better to articulate some of the bits by giving meaningful names to verbs along the following lines :

```
   boxrows=.<"1
   format=.3&":            NB. width = 3 characters
   length=.(i.&' ')"1      NB. gives length of string
   lengths2=.boxrows@(,.(format@length))
```

Interestingly, although the above four lines appear at first sight to have only a few primitive symbols, all such symbols in lengths are faithfully reproduced. Arguably it would have been better to write `lengths` this way in the first place as this helps to contrast the @:s which define compound verbs, with the implicit punctuation in the *hook* `,.(format@length)`.

Thoughtful punctuation can often help documentation. As a further example, most readers would find that on first sight the following verb definition conveys little of its purpose:

```
verb=.(+/@:*:@:-+/%#)%<:@#
```

With some redundant parenthesising and renaming, and use of space to emphasise the *fork*, things become a little clearer :

```
sdest=.(+/@:(*:@:(-+/%#))) % (<:@#)
```

and with a little more renaming of the parts

```
sum=.+/
mean=.+/%#
mdev=.-mean                NB. mean deviation
nminus1=.<:@#              NB. n minus 1
sdest1=.sum@:(*:@mdev)%nminus1
```

the objective of providing the usual form of standard deviation estimate from a sample should become reasonably apparent.

Cap

When *cap* ([:) was introduced it was argued that it allowed indefinitely long *trains* of verbs to be written without parentheses, thereby implying that parentheses were inherently undesirable. The analogy in English is to favour long strings of words without punctuation, which may not be to everyone's reading taste. Forks and hooks work well enough because the human mind assimilates readily twosomes and threesomes, but thereafter the reverse is true, that is a b c d e wrongly suggests "a then b then c then ..." whereas a b(c d e) gives a natural visual picture of the correct meaning. Continuing in the vein of the previous example `*:-+/%#` does not at first sight reveal its meaning whereas `*:-(+/%#)` says with reasonable clarity "subtract the mean from the squares".

Space

A first step in the parser of most compilers and interpreters is to remove redundant spaces which are often highly desirable at the orthographic level, for example to underline the fact that three primitive verbs form a fork. Successive digraphs can lead the reader through an unnecessary initial step of disentanglement as, for example, in verb above which, even without the suggested parenthesising and breaking down into smaller verbs, would be easier to interpret if written

```
verb=.(+/@:*:  @:  - +/%#) %  <:@#
```

that is, (add-following-square)-following-(mean-adjust) divide by decrement-tally. On the other hand spaces are probably best omitted between verbs and their objects, e.g. `i. 5` is probably less clear than `i.5`, although it is best not to be too dogmatic.

Code Summary

```
mean=.+/ % #
mean0=.(+/%#)"0          NB. scalarised mean
verb=.(+/@:*:@:-+/%#)%<:@#
sdest=.(+/@:(*:@:(-+/%#))) % (<:@#)
sum=.+/
mdev=.-mean              NB. mean deviation
nminus1=.<:@#            NB. n minus 1
sdest1=.sum@:(*:@mdev)%nminus1
lengths=.<"1 @:,.3&":@:i.&' '"1
lengths2=.boxrows@(,.(format@length))
    boxrows=.<"1
    format=.3&":         NB. width = 3 characters
    length=.(i.&' ')"1  NB. gives length of string
```

7. One Foot in the Grade

Principal Topics : /: (*grade up*) \: (*grade down*) |: (*transpose*) a. (*alphabet*) ? (*deal*), " (*rank* conjunction) sorting, ranking, collating sequence, tied ranks, schoolmasters rank, reflections, rotations, mean, median

A Vector obituary preview column? Well not quite, or maybe the answer should be, of a sort, since that is what grade (used generically to refer to both the *grade up* and *grade down* verbs) is all about. Sorting a list (the equivalent of APL **V**[Δ**V**]) has a direct J equivalent in the following bridge hook:

```
    ({~/:)4 2 7 1
1 2 4 7
```

The algorithm applies equally to character lists:

```
   ({~/:)>'Fred';'Joe';'Egbert'
Egbert
Fred
Joe
```

For convenience give names to the verbs which sort lists up and down:

```
  sortu=:{~/:
  sortd=:{~\:
  sortd>'Fred';'Joe';'Egbert'
Joe
Fred
Egbert
```

Sort by rows and sort within rows are differentiated by using the *rank* conjunction:

```
   ]u=.?3 4$10
4 0 1 2
7 9 3 2
1 2 9 5
   (sortu u);sortu"1 u
┌───────┬───────┐
│1 2 9 5│0 1 2 4│
│4 0 1 2│2 3 7 9│
│7 9 3 2│1 2 5 9│
└───────┴───────┘
```

For sort by columns do a row sort *under* (&.) the transformation |: (*transpose*):

```
  sortu&.|: u
```

```
0 4 5 6
9 2 7 7
7 6 0 3
```

Flexibility in the choice of collating sequence (say all the odd numbers are to be prior to any of the evens) is achieved by using dyadic `i.` :

```
   oe=.1 3 5 7 0 2 4 6 8
   (/:oe i.v){v=.4 2 7 1
1 7 2 4
```

Consolidate this in a verb :

```
  csortu=./:@i.{]                NB. x is collating seq.
  oe csortu 4 2 7 1
1 7 2 4
```

Here is a collating sequence which blurs the distinction between upper and lower case characters :

```
   ]cs=.(,|:65 97+every <i.26){a.
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz

   m=.>'bless';'This ';'house'
   (];sortu;cs&csortu)m
┌─────┬─────┬─────┐
│bless│This │bless│
│This │bless│house│
│house│house│This │
└─────┴─────┴─────┘
```

Dyadic Grade

Whereas dyadic grade in APL provides a means of changing the collating sequence, this is not the case in J where the following rule for dyadic *grade up* applies :

`x/:y` is equivalent to `(/:y){x`

so that x must be at least as long as y . A special case satisfying this constraint is when x and y are equal, in which case y/:y (or equivalently /:~y) is simply another definition of `sortu`. The definitions of `sortu` and `sortd` can therefore be shortened by one character to

```
  sortu=:/:~
  sortd=:\:~
```

Sorting by columns of a matrix other than the first requires dyadic grade. The following display demonstrates sorting on the 2nd and 4th columns.

```
 colsort=.] /: {"1
 (];1&colsort;3&colsort)u
┌───────┬───────┬───────┐
│4 0 1 2│4 0 1 2│4 0 1 2│
│7 9 3 2│1 2 9 5│7 9 3 2│
│1 2 9 5│7 9 3 2│1 2 9 5│
└───────┴───────┴───────┘
```

It is worth while pausing to consider how this fork works. The verb *from* ({) at rank 1 on the right of the fork selects the chosen column as y,] selects the entire table as x . Now apply the dyadic grade rule above. The indices for ordering the chosen column are obtained as /:y and are used to select from all the columns in the table.

When items within matrices have complex structures as in data base tables the principle remains unchanged but some minor variations may be necessary involving the use of *open* (>) and @: to take care of rank inheritance :

```
  selectc=:>@{"1          NB. select xth column of y
  sortc=:/:@:selectc { ]  NB. order y by column x

  dbase=.'York';'England';100000
  dbase=.|:('Edinburgh';'Scotland';500000),.dbase
  dbase=.('Aberdeen';'Scotland';200000),dbase
  ]dbase=.('Bristol';'England';400000),dbase
┌─────────┬────────┬──────┐
│Bristol  │England │400000│
├─────────┼────────┼──────┤
│Aberdeen │Scotland│200000│
├─────────┼────────┼──────┤
│Edinburgh│Scotland│500000│
├─────────┼────────┼──────┤
│York     │England │100000│
└─────────┴────────┴──────┘

  1&sortc dbase           NB. sort by country
┌─────────┬────────┬──────┐
│Bristol  │England │400000│
├─────────┼────────┼──────┤
│York     │England │100000│
├─────────┼────────┼──────┤
│Aberdeen │Scotland│200000│
├─────────┼────────┼──────┤
│Edinburgh│Scotland│500000│
└─────────┴────────┴──────┘

  2&sortc dbase                NB. sort by population
┌─────────┬────────┬──────┐
```

York	England	100000
Aberdeen	Scotland	200000
Bristol	England	400000
Edinburgh	Scotland	500000

Ranking

The potential confusion between rank in the J sense (as in *rank* conjunction) and rank in the sense of collating sequence ordering can be avoided by referring to the latter as **ranking**.

It might be expected that, by analogy with APL, /:/:v delivers the upward ranking of the elements of v.

```
    /:/:4 2 7 1
7 2 1 4
```

Why does APL intuition fail? The reason is that /:/: is a hook. The rightmost /: obtains the upward ordering index vector of v which is 3 1 0 2, which, in the absence of an explicit left argument is both left and right argument to the leftmost dyadic /: . Following the rule above, a second *grade up* is performed resulting in the intermediate generation of the rank list 2 1 3 0 which is then applied as a selection list (*from*) on the original right argument leading to the result 7 2 1 4. To block this last step it is necessary to use a different composition mechanism for the two /:'s, viz.

```
   (rku=:/:@/:)4 2 7 1
2 1 3 0
```

Downward ranking uses both *grade up* and *grade down* :

```
     (rkd=:/:@\:)4 2 7 1
1 2 0 3
```

Upward and downward **tied ranks** require slightly unwieldy verb combinations :

```
   rktu=.-:@(rku + \:@/:@|.)
   rktd=.-:@(rkd + \:@\:@|.)
   rktu v1=.5 3 3 5 2 5 8
4 1.5 1.5 4 0 4 6
   rktd v1
2 4.5 4.5 2 6 2 0
```

A variation of `rktd` is **schoolmaster's rank** in which students with equal scores are each rated as highly as possible, a property inherent in dyadic i.. The work for the schoolmaster's ranking verb has largely been done and it can be achieved by a single fork :

```
    sch=.i.~{rkd
    >:sch v1
2 5 5 2 7 2 1
```

The grade verbs have applications outside the realm of strict sorting. In E #30 ("Just what do they sell at C. and A.?") *grade up* is used to perform inverse permutations, that to reverse the effects of a shuffle. In technical terms, when the argument y is a permutation, that is an arrangement of all the items in `i.n` where n is a positive integer, then /: is a self-inverse verb, and so in these circumstances /:/:y is identical to y .

Then in E #26 ("Working in Groups") it transpires that the application of grade verbs to permutations matched exactly the transformations of rotations and reflections about axes in the plane applied to matrices which provide alternative representations of the permutation in the sense that, for example, 1 0 2 3 is represented by

```
. 1 . .
1 . . .
. . 1 .
. . . 1
```

Specifically /: is a reflection in the line x+y=0 but \: is a clockwise rotation through 90 degrees. The reflection/rotation distinction underlines the non-symmetrical behaviour of /: and \: in the ranking algorithms given above. /:@\: and /:@\: represent reflections in the x and y-axes, \:@\:@\: represents an anti-clockwise rotation through 90 degrees and \:@\: represents a half-turn. Finally /:@\:@\: represents a reflection in the line x=y.

Order statistics, such as median, require the use of grade. A median algorithm can be built up using three auxiliary verbs:

```
   mindex=.-:@<:@#  NB. 0.5*(n-1), n=length of vector y
   minmax=:<. , >.  NB. floor,ceiling of real number y
   mean=.+/%#       NB. arithmetic mean
   median=:mean @ ((minmax@mindex) { sortu)
   median 4 2 7 1   NB.mean of two middle values
3
```

Now introduce `sortu` as the right prong of a fork, and use `mean` to average the values of the two (possibly identical) middle values :

```
   median=:mean @ ((minmax@mindex) { sortu)
   median 4 2 7 1   NB.mean of two middle values
3
```

Verbs defining other partition values, e.g. percentiles, use the same principle, although inevitably are more complex in detail.

In short grade is a very versatile verb for which you should rightly have grade expectations. Have a grade day!

Code Summary

```
   sortu=:/:~                        NB. sort upwards
   sortd=:\:~                        NB. sort downwards
   cs=.(,|:65 97+every <i.26){a.        NB. alphabet AaBbCc…
   colsort=.] /: {"1                    NB. sort matrix by col-
umns
   selectc=:>@{"1                    NB. select xth column of y
   sortc=:/:@:selectc { ]            NB. order y by column x
   rku=:/:@/:                        NB. upward ranking
   rkd=:/:@\:)                       NB. downward ranking
   rktu=.-:@(rku + \:@/:@|.)         NB. upward ranking with ties
   rktd=.-:@(rkd + \:@\:@|.)         NB. downward ranking with ties
   sch=.i.~{rkd                      NB. schoolmater’s ranking
   mindex=.-:@<:@#                   NB. 0.5*(n-1) where n is
length of vector y
   minmax=:<. , >.                   NB. floor,ceiling of real num-
ber y
   mean=.+/%#                        NB. arithmetic mean
   median=.mean @ ((minmax@mindex) { sortu)
```

8. Transpositions, Perms and Combs

Principal Topics : |: (*transpose*) "(*rank* conjunction), #(*tally*) SQL, diagonals of arrays, mappings, transformations, merged axes, symmetry test, inverse permutations.

Those with a mathematical training will naturally association the verb *transpose* (|:) with matrices and 'row and column switches'. This is perfectly natural, but I would suggest that, as with many aspects of J, it is insightful to think also in terms of lists, which also helps to bring thinking closer to what is going on behind the scenes. Consider

```
   i.3 4
0 1  2  3
4 5  6  7
8 9 10 11
```

This is a 3-list of lists, each of which is 4-list. Where secondary lists are of equal length (or equivlently the data is rectangular) a second list of lists automatically exists, viz. a 4-list of 3-lists, these being obtained by making a list of all the first items, then another of all the second items and so on (in simple terms, working down columns!). If the list hierarchy of `i.3 4` is say 0 1, then the effect of transposition is that of turning the secondary list into the primary one, so that the hierarchy becomes 1 0. With every further imbedding of equal-length lists such as `i.2 3 4` (a 2-list of 3-lists of 4-lists), there comes a new layer in the list hierarchy, and with it new possibilities for reordering that hierarchy. For example with `i.2 3 4` and a hiererchy 0 1 2, promoting 2 to the top level is equivalent to starting with all the first items in the 4-lists. These form a 2-list of 3-lists and lead to another choice, 0 1 or 1 0. Following this argument the number of possible transpositions is equal to the number of permutations amongst these hierarchies. It is very reasonable then that the left argument of `|:` should be a permutation of the *tally* (#) of the rank of the right argument. The manner in which such permutations is defined is that the hierarchly levels denoted by the left argument are pushed to the right leaving the remaining levels unchanged, whereas without any argument at all, that is the monadic case, the hierarchy is completely reversed :

```
   $0 2|:i.2 3 4
3 2 4
   $|:i.2 3 4
4 3 2
```

Matrices can be modelled as lists, and higher order rectangular arrays as multi-dimensional cuboids, so that in geometrical terms, transposition is the process of switching the order of axes. Arrays can also be described algebraically in terms of coordinates, and the result of transposition is a mapping (that is, transformation) of an array a onto itself in which each item such as a[i;j;k;l] has an image with the same coordinates in a different order as determined by the permutation list which is the left argument. Here are a few cases in which a=.i.2 3 4 5 :

left argument	new $a	image of a[i;j;k;l]
0 1	4 5 2 3	a[k;l;i;j]
0 2 3	3 2 4 5	a[j;i;k;l]
3 1 0	4 5 3 2	a[k;l;j;i]
2 3 1 0	4 5 3 2	a[k;l;j;i]

With a rank 4 array such as a there are 43 = 64 possible non-empty left arguments (4 with a single integer, 12 with two integers, and 24 with three and four integers in each case). Some of the corresponding transpositions must overlap since there are only 24 distinct permutations of $a. Consider the arguments in the last two rows in the table above. These are necessarily identical since pushing 3 1 0 to the right is equivalent to placing the remaining index first. Similarly a left argument 2 3 0 1 produces the same result as 0 1. A singleton left argument demotes that axis to the lowest hierarchy level. The opportunities afforded by arguments of less than full length therefore introduces redundancy. Further redundancy arises from applying monadic transpose at different levels of rank using the rank conjunction. Specifically for a rank 4 object the following monadic transpositions and dyadic transpositions are equivalent:

```
      |:"1         |:"2        |:"3         |:"4
 0 1 2 3&|:    0 1 3 2&|:  0 3 2 1&|:   3 2 1 0&|:
```

Boxed arguments

Boxed arguments allow the possibility of restructuring by merging axes. If the argument is boxed, the indices contained in the box are merged, and the corresponding axis-length in the box is the minimum of these axis-lengths. Since this results in general in the non-selection of some items, the overall effect is that of reducing the overall bulk of data. In the shape vector of the result array, the unmerged axes all appear before the merged one. In terms of mappings, transpose effects

a many-to-one mapping, for which the image of each point in the result array r is given in the final column below. The co-ordinate names in this column match the values in column two, and can be used to establish systematically from source the value of any item in the result.

argument	new $a	condition on index for an item of a to be selected	image of a[i;j;k;l]
<0 1	4 5 2	i=j	r[k;l;i]
<0 2	3 5 2	i=k	r[j;l;i]
<1 2	2 5 3 0	j=k	r[i;l;j]
<0 1 3	4 2	i=j=l	r[k;i]

Two properties of the above table deserve emphasis:

1. unlike the non-boxed case, the order of items in the boxed argument is immaterial, that is the contents of a box is a combination of m items from n rather than a permutation;
2. the third column echoes the first column with i,j,k,l replacing 0,1,2,3, and =s inserted, and the fourth column echoes the second column as explained above.

There is a parallel between the third column above and the SQL (Structured Query Language) command

SELECT ... WHERE ...

It may help to think of boxed arguments in these terms, subject to the qualification that it is indices which are selected and not values as would be the case with SQL.

Consider another specific example, say (<0 2)|:a . There are two steps in evaluating this. The first and most straightforward step is to identify the new $a . (<0 2) means that it is axes 0 and 2 which are to be merged. The length of the merged axis-length is 2<.4, that is, 2. The remaining axis lengths are 3 5, and so $a is 3 5 2, which means that the result is a 3-list, each list of which is a 5-list, each item within which is a 2-list. Using the SQL analogy (<0 2)|:a can be read as

SELECT a[i;j;k;l] WHERE i=k

The next step is to identify the original index of the data which occupies each cell in the transposed form. This is where the third column comes in. The constraint i=k means that the only items which are to be selected are those whose indices are of the form 0 x 0 y and 1 x 1 y

where x is chosen from i.3 and y is chosen independently from i.5. Specifically the indices of the items in the first 3-list are :

```
0 0 0 0       1 0 1 0
0 0 0 1       1 0 1 1
0 0 0 2       1 0 1 2
0 0 0 3       1 0 1 3
0 0 0 4 1 0 1 4
```

and those of the remaining two 3-lists are obtained by replacing the 0s in the second column in turn with 1s and 2s. The order of the most rapidly changing indices is l, j, k=i .

To transform a 3-list of 5-lists of 2-lists into a 5-list of 3-lists of 2-lists apply the non-boxed transposition 1 0 2|:(<0 2)|:a . The pattern of indices for the first list is now

```
0 0 0 0 1 0 1 0
0 1 0 0       1 1 1 0
0 2 0 0       1 2 1 0
```

and the order of the most rapidly changing indices is now j, k=i, l .

Geometrically speaking, the result of a merge transpose is a diagonal cross-section of rectangular data whose rank is determined by the number of axes merged. If the length of the boxed left argument is n then the rank of the result is less than the rank of the original by n-1 .

One of the most frequently occurring practical applications of *transpose*, which also demonstrates a noun-verb *bond*, is

```
   diag=.(<0 1)&|:     NB. leading diagonal of matrix
```

Its meaning should be readily apparent, as should that of

```
   ndiag=.diag@:(|.”1)   NB. non-leading diagonal
```

The following hook provides a symmetry test :

```
   sym=.-:|:
   (sym i.3 3),sym 3 3$1   NB. 1 if symmetrical
0 1
```

Transpose can be used as a transformation when either an operational verb is most readily coded for a different arrangement of data axes, or

reuse of an existing verb is easier than writing a new one to accommodate a different axis ordering. This principle can be simply illustrated using a rank three object `d=.i.2 3 4` thought of as a structure of planes, rows and columns.

1 2+d adds 1 to the first plane and 2 to the second. Corresponding additions to rows are achieved by applying the rank conjunction, 1 2 3 +"2 d, and to columns by 1 2 3 4+"1 d. The rank conjunction can be circumvented by promoting say columns to become the major dimension using transpose, adding, and then using the inverse transposition to restore the original shape.

```
1 2 0 |: 1 2 3 4 + 2 0 1 |:d
```

or, to put it more succinctly

```
(1 2 3 4&+&.(2 0 1&|:))d
```

An inverse transposition is obtained by using the corresponding inverse permutation. Inverse permutations are obtained directly by *grade up* provided that the permutation is spelt out in its full form. In the above example 1 2 0 and 2 0 1 are inverse permutations, that is 1 2 0 -:/: 2 0 1 so that a -: 1 2 0|:2 0 1|:a for any rank three object a.

The single item permutation cases are sometimes useful. For a rank three object, 1|: exchanges rows and columns within planes. For an object of any rank 0|: promotes the second dimension to become the major dimension, and (n-1)|: swaps the two lowest-order dimensions.

It is worth observing that data content has taken little or no part in this discussion, that is transposition is all about changing the shape of data in the unboxed case, and in determining how to make selections from it in the boxed case.

Code Summary

```
diag=.(<0 1)&|:         NB. leading diagonal of matrix
ndiag=.diag@:(|."1)     NB. non-leading diagonal of matrix
sym=.-:|:               NB. symmetry test
```

9. Power Steering extra

Principal Topics ^: (*power conjunction*) _ (*infinity*) = (*self-classify)* matrix product, eigenanalysis, infinity, left binding, normalisation, convergence, transition matrices, population forecasts.

The *power* conjunction is one of the most versatile computational tools in J. At the same time there are traps for the unwary which must be avoided. Consider the following statement :

> 2*^:(4)1 and 1*^:(4)2 mean "multiply 1 by 2 four times" and "multiply 2 by 1 four times" respectively, which are the same thing because multiplication is commutative, that is, 1 times 2 is the same as 2 times 1.

Obviously true? No – the answer is that the left operand is glued more closely to the 'times' so that the first of these expressions is properly interpreted as 'multiply-by-2' applied four times to 1 giving 16, and the second is 'multiply-by-1' applied four times to 2 which gives 2. J is in this sense 'left-handed', or, as the more technically aware would say, left binding is stronger than right.

Next, here is the standard matrix product verb to allow some work with matrices:

```
mp=.+/ .*            NB. matrix product
```

To self-test your understanding of the power conjunction, cover up all but the next two or three lines and answer the questions : what do the following do to a matrix

```
(a)  mp~^:2   and    (b) mp~^:3 ?
```

If your answer was that it raises it to the fourth and eighth powers respectively – well done! In general `mp~^:n` raises a matrix to the power 2n on account of the way in which each result is 'fed back' to the next exponentiation step.

Suppose you didn't want to go up in quite such big leaps, how about

```
(a) m&mp^:2 m and (b) m&mp^:3 m ?
```

This time the answers are the third and fourth powers of m. In general m&mp^:n raises m to the power n+1, which is irritatingly 'one out' if you are looking for a tidy verb for m^n .

To find such a verb, go back to the scalar case 2*^:(4)1 which raised 2 to the power 4. The key ingredient is the 1, that is the identity of multiplication. A verb which gives the identity matrix of the same shape as any square matrix is `id=.=@i.@#` , following which a verb which raises a matrix x to the power y is

```
   power=.dyad def 'x mp^:y id x'
```

in which y may be a list, so that several powers of a matrix can be obtained simultaneously. For example

```
   ]m2=.2 2$4 1 _2 1
 4 1
_2 1
   m2 power 10 9
 117074  58025
_116050 _57001

  38854  19171
 _38342 _18659
```

Now compute the ratios of two successive powers:

```
   ratio=.%/@power
   m2 ratio 10 9
3.01318 3.02671
3.02671 3.05488
```

This demonstrates that the items in the above matrix all tend to a value which is in fact the value of the largest **eigenvalue** of the matrix, which for m2 happens to be 3. Eigenanalysis at this early stage? - heady stuff! Press on and use this value 3 to obtain a value for the smallest eigenvalue :

```
     3+(m2-3*id m2)ratio 20 19
2 2
2 2
```

You don't have to be exact with the multiplier 3, but the closer you are to the true eigenvalue the faster is the convergence:

```
   5+(m2-5*id m2)ratio 20 19
1.9994  1.9997
1.9997 1.99985
```

As an aside – in the verb definition `id=.=@i.@#` you might think that the = is equals, but bearing in mind that the left conjunct of @ must be monadic, = in this definition is *self-classify*, which returns an equals table for the *nub* of a list y versus itself.

The above experiments need not be confined to two by two matrices, for example :

```
   ]m3=.3 3$1 1 1 1 2 1 1 1 3
1 1 1
1 2 1
1 1 3
   m3 ratio 10 9
4.21425 4.21415 4.21442
4.21415 4.21392 4.21456
4.21442 4.21456 4.21418
   4+(m3-4*id m3)ratio 20 19
0.3249388 0.3244097 0.3258422
0.3244097 0.3278868 0.3184167
0.3258422 0.3184167 0.3383134
```

showing that the largest and smallest eigenvalues of m3 are about 4.214 and 0.325. Given that the sum of eigenvalues is the sum of the leading diagonal, the third eigenvalue is readily calculated as 1.461. As for the eigenvectors, it is convenient first to define a verb based on the hook %>./ which normalises a list so that its largest item is 1, or in the case of a matrix, it does this for each row separately, hence the *transpose* and the *rank* conjunction :

```
   norm=.(%>./)"1%|:
   norm m3
        1         1   1
      0.5         1 0.5
0.3333333 0.3333333   1
```

Now observe the behaviour of the successive (normalised) powers of m3 as they converge :

```
    norm m3 power 10
0.5254314 0.6889001 1
0.5254368 0.6889113 1
0.5254221 0.6888809 1
```

Each row when fully converged is simply an eigenvector corresponding to the largest eigenvalue, and repeating the previous 'smallest eigenvalue' wrinkle :

```
    norm 4+(m3-4*id m3) power 20
_2.07862          1  0.4032745
       1 _0.4818991  _0.193451
```

```
        1 _0.4797006 _0.1949656
```

results in rows each of which converge to the eigenvector corresponding to the smallest eigenvalue.

It is not of course suggested that this is the basis of a systematic way of calculating eigenvalues and eigenvectors – for one thing there are numerous special cases which have to be looked out for, and the choice of adjustment factor for the smallest eigenvector must in general be made with some care.

The value of the *power* conjunction for obtaining, say, quick population estimates is readily apparent. Suppose that `transm` is a transition matrix giving the proportions of juveniles, workers and elderly who change from one status to the next within any one year, so that, for example, 94.71% of juveniles remains so at the end of the year, the birth rate for the workers group is 5.52% and the death rate within the elderly group is 5.7%.

```
    ]transm=.3 3$0.9471 0.0552 0 0.0379 0.9682 0 0 0.0207 0.943
0.9471 0.0552     0
0.0379 0.9682     0
     0 0.0207 0.943
```

If the current population (in millions) is 11 38 and 9 for the three sectors, the predicted values in 20 years time are

```
   trans mp^:20 11 38 9
26.7554 31.9539 11.1334
```

As before the ratio between successive powers shows convergence to the largest eigenvalue :

```
    transm ratio 100 99
1.00458  1.0046     0
 1.0046 1.00459     0
 1.0049 1.00457 0.943
```

This is approximately 1.005, which is the instantaneous overall rate of population increase.

For a matrix which has a column with only non-one zero item, that item is necessarily one of the eigenvalues, and the others are the eigenvalues of the matrix obtained by removing its row and column. Thus 0.943 is an eigenvalue of transm and the remaining eigenvalue is the smallest eigenvalue of

```
0.9471 0.0552
0.0379 0.9682
```

Power and Infinity

Returning to the *power* conjunction, there is another smart feature of J, namely 'function raised to the power infinity'. As observed earlier, the function values are repeatedly fed back as input, and if this process ultimately leads to convergence, then ^:_ will deliver it. For example given

```
   cos=.2&o.
```

successive values of cos y, cos(cos y), cos(cos(cos y))... starting from an initial value of 0 are given by

```
   cos^:(i.6)0
0 1 0.5403023 0.8575532 0.6542898 0.7934804
```

and where these converge, as they do for cos, the converged value is the solution of the equation cos(y)=y :

```
   cos^:_(0)
0.7390851
```

This feature is useful in the sort of population study which concerns, for example, the spread of AIDS within a third world population. Suppose that the current population is 60 million and an 8% birth rate is assumed, a percentage which is subject to a downward adjustment of 10^{-7} per unit of population to account for pressures of space, food shortage and so on. Assume also that there is a net annual emigration of 10,000. Forecasting future population levels is a matter of repeated exercising a polynomial defined by

```
   p=.#.&_0.000000001 1.08 _10000
```

so that the projected population in 20 years time is :

```
   p^:20(60000000)
7.50634e7
```

The infinity option gives the ultimate equilibrium population when the forces of environmental pressure and migration exactly balance the birth rate :

```
   p^:_ (60000000)
7.98748e7
```

that is equilibrium is attained at about 80 million.

Clearly there's a lot of power in J!

Code Summary

```
 power=.dyad : 'x mp^:y id x'     NB. matrix x to power y
   mp=.+/ .*                      NB. matrix product
   id=.=@i.@#                     NB. identity matrix
 ratio=.%/@power           NB. ratio of successive powers
 norm=.(%>./)"1%|:         NB. normalises each row of a matrix
```

10. Bonding is Power – how Interesting

Principal Topics : |. (*shift*) ^: (*power* conjunction) currying, compound interest, savings schemes, annuities, net present value.

"All verbs are monadic" may sound a heretical statement, after all what about 2+3? At the surface level the left and arguments have equal force, but delve a little more deeply, and at interpretation time in the computer one argument must be fetched before the other, in which sense argument marshalling is not symmetric, that is 2+3 is not the same as 3+2. In the case of 2+3 an intermediate unarticulated verb, either 2&+ or +&3 is created to which the other argument is presented monadically. The *power* conjunction helps to determine which, that is does 2+^:(4)3 mean 2 added 4 times to 3, or 3 added 4 times to 2? - the answer is determined by

```
   ((2&+)^:4)3
11
   ((+&3)^:4)2
14
```

that is it is the left argument which is more strongly bound. More generally in computer science the idea of argument bonding is known as **currying** after the American logician H. Curry, and the J documentation explicitly offers 'curry' as alternative name for *bond* (&).

Already the link between *bond* and the *power* conjunction is becoming clear, and this is made explicit in the J help file by the formal definition of bond

x m&n y ↔ m&v*:x y

To see how this works with lists look at the following line in which 4 and 40 are added progressively to start values 100 and 1000 several times :

```
   1 2 3(+&4 40)100 1000
104 1040
108 1080
112 1120
```

The above describes the process of accumulating simple interest at 4%. The difference with compound interest is that it is a process of progressive multiplication

```
   1 2 3(*&1.04 1.05)100 1000     NB. compound interest
```

```
    104    1050
 108.16  1102.5
112.486 1157.63
```

Replace multiplication with division, and the result is net present values

```
    1 2 3(%&1.04 1.05)100 1000   NB. net present value
96.1538 952.381
92.4556 907.029
88.8996 863.838
```

which can also be obtained by using negative power values with multiplication

```
   _1 _2 _3(*&1.04 1.05)100 1000
96.1538 952.381
92.4556 907.029
88.8996 863.838
```

Savings schemes

A scheme whereby regular sums are added periodically to a capital sum which attracts compound interest is a mixture of both these processes, and the verb base can be brought into the picture. 100 #.1.05 4 equals 109 and so

```
   1 2 3 4 5(#.&1.05 4)100
109 118.45 128.373 138.791 149.731
```

shows the progress of such a scheme. With a little help from the *rank* conjunction the calculation can be replicated for several initial capitals:

```
    1 2 3 4 5(#.&1.05 4)"(1)0,.,100 1000 2000 10000
  109  118.45 128.373 138.791 149.731
 1054  1110.7 1170.24 1232.75 1298.38
 2104  2213.2 2327.86 2448.25 2574.67
10504 11033.2 11588.9 12172.3 12784.9
```

An **annuity** is just a savings scheme with a negative addition in each period and thus in general a reducing value:

```
   1 2 3 4 5(#.&1.05 _6)100
99 97.95 96.8475 95.6899 94.4744
```

2&v and v&2 are different for non-commutative verbs. The difference is most readily seen by comparing the boxes

```
   (1 2 3 4 5(%&2 3)6);1 2 3 4 5(2 3&%)6
```

```
┌────────────────────┬────────────┐
│     3            2 │0.333333 0.5│
│   1.5     0.666667 │       6   6│
│  0.75     0.222222 │0.333333 0.5│
│ 0.375    0.0740741 │       6   6│
│0.1875    0.0246914 │0.333333 0.5│
└────────────────────┴────────────┘
```

In the left hand box, 2 and 3 are progressively divided by 6; in the right hand box 6 is progressively divided by 2 and 3, and dividing by the previous quotient is a null operation.

Progressive series

The straightforward evaluation of a series such as 3n+2 would routinely be carried out by say

```
   (3*i.10)+2
2 5 8 11 14 17 20 23 26 29
```

"Progressive evaluation" means feeding back each evaluation as the next n. The result of the first five values of such a procedure for 3n+2 is

```
   1 2 3 4 ((3&+)&2)1
5 17 53 161
```

the last of which values could more mundanely be obtained as

```
   (3*(3*(3*((3*1)+2))+2)+2)+2
161
```

The simple evaluation of the series 3n+2 is obtained with a left argument of 1 :

```
   1 ((3&+)&2)i.10
2 5 8 11 14 17 20 23 26 29
```

Adding &(^&2) squares the left argument :

```
   2 (((3&+)&2)&(^&2)) 1
161
```

and adding square to the left gives progressive values of $(3n+2)^2$

```
   2 3 ((*:@(3&+)&2)) 0
196 348100
```

which are the squares of 14 and 590 respectively.

Non-numeric applications

Progessive left shifts of a list :

```
   0 1 2(1&|.)'conundrum'
conundrum
onundrumc
nundrumco

   0 1 2(3&|.)'conundrum'
conundrum
undrumcon
rumconund
```

an effect which could also be achieved using the *rank* conjunction

```
   0 3 6 |."0 1 'conundrum'
conundrum
undrumcon
rumconund
```

Progressive reduction of a list :

```
   0 1 2(2&}.)'conundrum'
conundrum
nundrum
ndrum
```

Progressive repetition of a list :

```
   1 2 3 ('abc'&,)''
abc
abcabc
abcabcabc
```

as opposed to repetitions of every item separately :

```
   1 2 3 #every <'abc'
abc
aabbcc
aaabbbccc
```

Together the *bond* conjunction and the *power* conjunction are enormously versatile weapons in the J armoury.

11. Time for amendment of data

Principal Topics : } (*amend*, *item amend*), ? (*deal*), " (*rank* conjunction) gerund, 'cleaning' small numbers to zero, multiple choice tests.

Updating - the amend adverb

Updating part of a list disguises the fact that there are really two processes present which are telescoped into one. The first process involves a data transformation which selects those parts which are to be changed, while the second process does the actual replacement with a second set of data. Syntactically *amend* (}) is an adverb qualifying a selector (noun) to its left to produce a verb whose left and right arguments are the new and old data respectively, so 'new selector } old' should be read as 'new (selector}) old' .

```
   a=.i.2 3 4
   ((9) 1}a);((9) 1}"1 a);(9) 1}"2 a
┌──────────┬───────────┬───────────┐
│0 1  2  3 │ 0 9  2  3 │ 0  1  2  3│
│4 5  6  7 │ 4 9  6  7 │ 9  9  9  9│
│8 9 10 11 │ 8 9 10 11 │ 8  9 10 11│
│          │           │           │
│9 9  9  9 │12 9 14 15 │12 13 14 15│
│9 9  9  9 │16 9 18 19 │ 9  9  9  9│
│9 9  9  9 │20 9 22 23 │20 21 22 23│
└──────────┴───────────┴───────────┘
```

The phrase (9) 1} a demonstrates that an adverb, unlike an adverb in ordinary grammar, may qualify either a noun or a verb. By default, selection takes place at the level of items within lists, in this case at rank 3, but the rank conjunction allows indexing to apply at lower levels. In particular (9)0}"0 a replaces all atoms with 9.

} also provides a quick way of generating coarse plots of data presented in the form of co-ordinate pairs which act as scatter index coordinates, e.g. :

```
   z=.0 0;1 1;2 4;3 10
   '*'  z}4 11$' '
*
 *
    *
          *
```

An updating problem : a choice of methods

One way to change the initial letter of a set of words is

```
   words=.'blood';'blight';'bear'
   words
┌─────┬──────┬────┐
│blood│blight│bear│
└─────┴──────┴────┘
   (<'B'),each }.each words
┌─────┬──────┬────┐
│Blood│Blight│Bear│
└─────┴──────┴────┘
```

which involves two essential operations, *drop* and *append*. *Amend* allows these to be telescoped into one :

```
   'B' 0}each words
┌─────┬──────┬────┐
│Blood│Blight│Bear│
└─────┴──────┴────┘
```

Selectors do not have to be explicit, they can be returned by verbs as in the next example. If *open* (>) is applied to words the result is a set of homogenous (equal length) lists, and so in order to change the last characters, it is necessary to compute the 'coordinates' of the final non-blank characters :

Changing last characters

Suppose you want to replace the last characters in each of the list of words :

```
   llc=.<:@i.&' '                  NB. locate last character
   ilc=.<"1@(i.@# ,. llc"1)        NB. indexes of last characters
   ilc >words
┌───┬───┬───┐
│0 4│1 5│2 3│
└───┴───┴───┘
   'xsk' (ilc words)}>words
bloox
blighs
beak
```

Repeating the data-name `words` in the above phrase is not inherently pleasing. However, the specification of amend allows this to be tidied up by using a gerund, which at the same time allows the replacement characters to appear as the left argument :

```
   replasts=.[`(ilc@])`]}
   'xsk'replasts >words
```

```
bloox
blighs
beak
```

Without the gerund option, it is hard to accommodate amend in explicit definitions. A nontrivial transformation of the old data might be to convert lowercase characters to upper case :

```
   lctouc=.monad :'(t-32*96<t=.a.i.y){a.'
   lctouc EVERY words
┌─────┬──────┬────┐
│BLOOD│BLIGHT│BEAR│
└─────┴──────┴────┘
```

Using a boxed list presents a difficulty because of the need for an *open* between successive indexing activities. To get round this an amend based verb can be defined to work at the item level and applied to each of the items in the object :

```
   RepLASTS=.[`(<:@#@])`] }
   'xsk'RepLASTS each words
bloox
blighs
beak
```

Item amend

So far, the adverbially qualified verb 'selector}' has been used dyadically. It can also be monadic in which case } is called *item amend*. The result has the structure of a single item of the right argument y, and its value is determined by selecting indices for y's of those items which are to be amended. Take for example a simulation of answers to a multiple choice test. The data is five items, each comprising twenty repetitions of the same character; the result of each execution is a further item, each of whose items comes from just one of the original five.

```
    ]mch=.|:20 5$'ABCDE'   NB. Construct char matrix mch
AAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCC
DDDDDDDDDDDDDDDDDDDD
EEEEEEEEEEEEEEEEEEEE
```

The following is a random selection of responses :

```
    (?20$5) } mch          NB. (?20$5)} is a noun
ADDEBCAAECDABACDCEEC       NB. result may be different!
```

... or if the responses are required in strict sequence :

```
   rint=.({. | i.@{:)@$    NB. Repeat row indices to length of
   rint mch                NB. number of columns
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
   rint} mch
ABCDEABCDEABCDEABCDE
```

Selection by criterion

Amend along with a selector generating verb allows updating using arbitrary indexes, and hence updating by criterion. For example, suppose an array contains some 0s

```
   ]p=.3 5$i.3
0 1 2 0 1
2 0 1 2 0
1 2 0 1 2
```

The 0s can be identified by

```
   (=&0)p
1 0 0 1 0
0 1 0 0 1
0 0 1 0 0
```

This criterion can be transformed into an array of indices satisfying the criterion by

```
   (i.$p)*(=&0)p
0 0  0 3 0
0 6  0 0 9
0 0 12 0 0
```

To change all the 0s into 99s say

```
   99((i.@$*=&0)@])}p
99  1  2 99  1
 2 99  1  2 99
 1  2 99  1  2
```

This amendment has been applied for one specific criterion, namely ‘equals zero’, whereas the technique is clearly generalisable, suggesting an adverb which transforms a criterion verb into the verb which gives the matching indices in y :

```
   ind=.adverb : '(i.@$*x)@]'
   99(=&0 ind)}p
99  1  2 99  1
 2 99  1  2 99
```

```
 1  2 99  1  2
```

A good use of this technique is to ‘clean’ numeric arrays of very small near-zero numbers which typically arise from floating point calculations

```
   clean=.0&((<&1e_6@| ind)})
   clean %1000000*_5+i.10
0 0 0 0 _1e_6 _ 1e_6 0 0 0
```

To round numbers which are very close to integers use `swingu` :

```
   cleani=.]`swingu@.(<&1e_6@(|@- swingu))
   swingu=.<.@+&0.5 NB. move to nearest integer
   >cleani each 5.9999999 5.999999 6.000001 6.0000009
6 5.99999 6 6.00001
```

Code Summary

```
   ilc=.<"1@(i.@# ,. llc"1)       NB. indexes of last chars
       llc=.<:@i.&' '             NB. locate last char
   lctouc=.monad :'(t-32*96<t=.a.i.y){a.' NB. change l/c to u/c
   replasts=.[`(ilc@])`]}         NB. replace last chars
   RepLASTS=.[`(<:@#@])`] }              NB. ditto for boxed
lists
   rint=.({. | i.@{:)@$           NB. Repeat row indices
   ind=.adverb : '(i.@$*x)@]'
   clean=.0&((<&1e_6@| ind)})     NB. clean small values to o
   cleani=.]`swingu@.(<&1e_6@(|@- swingu))
                                  NB. clean to integer
   swingu=.<.@+&0.5               NB. move to nearest integer
```

12. Obverse to Adverse :.:: a trip along the Brailleway

Principal Topics : :. (*obverse*) :: (*adverse*) /. (*key*) ^: (*power* conjunction) b. (*basic characteristics*), inverse, compound interest, frequency tables, currying, scans, alternative verbs, error control

Welcome to the Vector Value excursion from Obverse Central to Adverse! Don't forget you need an inverse for the return journey – personalised if you like – and note that the inverse is also mandatory (a) when the train gets *under* (&.) way and (b) when it *power*s (^:) up with a negative argument. For many primitive monadic verbs and a few dyadic ones an inverse function is well defined, and there's a list of these on a billboard further down the line. For example *logarithm* (^.) and *power* (^) are inverse operations, and this underlies the use of &. and ^: in the following examples :

```
    log=.^.
    2+&.log 3        NB.  e to the power ln2 + ln 3
6
    2 ^.^:(_1)3      NB.  2 inverse-log (that is, to the power)
3
8
    2 ^.^:(_2)3      NB.  2 inv-log inv-log 3
256
```

The first of these examples performs a transformation, 'log to base e', on both arguments prior to the primary operation plus, then performs the 'inverse log' transformation on the result. In the second and third examples parentheses are necessary to separate the arguments of the adverb ^: and the adverbially modified verb ^. .

Suppose that through some perversity you decided to conclude the log operation by squaring rather than taking anti-logs :

```
   log=.^. :. *:    NB. log with "square" as obverse
   2 +&.log 3       NB. square of (ln 2 + ln 3)
3.2104
```

It is hard to see the circumstances in which this would be a sensible thing to do; however when you define your own verbs, J may not be able to define an inverse, and so you are given the opportunity to define your own through *obverse*. The *obverse* does not need to have a direct connection with the primary verb, nor does it affect its operation, other than in the contexts of *under* and negative *power*. Here is

another example with an *obverse* in which a primary operation joins names which are to have trailing blanks deleted prior to the join, with a final comma to be added after the *append* :

```
   dtb=:({.~i.&' ') :. (,&',')    NB. delete trailing blanks
   (dtb 'mickey     '),'*'        NB. obverse does not affect
mickey*                           NB. normal opeartion
   'mickey     ' ,&.dtb 'rooney    '
mickeyrooney,
```

In ordinary usage the obverse side of a coin means the side commonly turned towards you, which in turn usually means 'heads' as opposed to 'tails'. This reflects quite well the idea of obverse in J, because it emphasises (a) that when you use it you are thinking of a verb as a two-sided object, and (b) as with a coin there is no reason why the designs on the two sides should be directly related, although often the relationship is inverse in nature. Another example involves a frequency table verb `frtab` :

```
    x=.0 0 1 1 2 2 3
    frtab=.+/"1@=
    frtab x
2 2 2 1
```

A natural inverse operation would be to 'unwrap' the frequencies back to the original data which is what the verb `convert=.# i.@#` does. So redefine

```
     frtab=.+/"1@= :.convert
     frtab ^:_1 frtab x
0 0 1 1 2 2 3
```

The conductor will now give a further demonstration of obverse by converting between percentages and multiples as in, e.g. handling compound interest calculations. Suppose the percentage growth rates of an investment are known for a number of years, and it is desired to compute the average overall growth rate throughout the entire period. The following two verbs convert percentages to and from multiples:

```
   ptok=:>:@(*&0.01)       NB. %age to multiple, 5 -> 1.05
   ktop=:%&0.01@<:         NB.  multiple to %age, 1.05 -> 5
   */ptok 5 6 13 _12 0
1.10677
```

Now suppose that this result should be given as percentage :

```
   ptok=.>:@(*&0.01) :. ktop     NB. ptok with obverse
```

```
   */&.ptok 5 6 13 _12 0
10.6767
```

To compute the average, as opposed to total, growth over the period introduce *under* :

```
   gm=.(+/%#)&.^."_         NB. geom mean(=arith mean under log)
   gm&.ptok 5 6 13 _12 0
2.04959
```

(The rightmost two characters of `gm` are necessary to make it into a scalar verb.) It is reasonable to retain the name `ptok` for the obverse-enriched verb since adding the 'flip side' verb `ktop` does not affect its previous function; it is only in the presence of *under* and negative power that the enrichment occurs. In this case the *obverse* is just the inverse, which J is smart enough to figure out for itself, since it has a rule that the inverse of a verb u@v is (v^:_1)@(u^:_1), assuming of course that u and v are among the list of primitive verbs for which inverses can reasonably be computed, as is the case with `ptok`. So why bother to write an obverse when all those bright J folks have already given you what you probably wanted anyway, and moreover have provided a conjunction b. which by joining a verb to _1 returns its *obverse* :

```
    ptok=:>:@(*&0.01)       NB. back to the original ptok
    ptok b. _1              NB. compute the obverse ..
%&0.01@<:                   NB. .. which is identical to ktop.
```

Well, let's suppose that on the flip side (converting multiples back to percentages) you decide to display an explicit percent sign. Easy, just do

```
   mut=.,&'%'@(":@ktop)    NB. enrich ktop with %
   mut 1.05
5%
   ptok=.>:@(*&0.01) :.mut NB. redefine obverse of ptok
   gm&.ptok 5 6 13 _12 0
2.04959%
```

The verb `ptok` provides a salutary lesson in the power of parentheses. The verb

```
      pet=.>:@*&0.01
```

means `(>:@*)&0.01` and is indistinguishable from `ptok` when used directly. However in inverse mode, and bearing in mind that conjunctions have long left scope, the rule for the inverse of `ptok` is

```
        (*&0.01 ^:_1)@(>: ^:_1)
```

that is `(%&0.01 ^:_1)@(<:)` , or in plain English subtract one and multiply by 100 (that is, divide by 0.01), which is exactly what `ktop` does. The computed obverse of `pet`, however, is a somewhat complicated verb, which in the case of a scalar argument reduces to division by 1.01. The programming lesson is that whereas parentheses may be redundant in the normal operation of a verb, they can sometimes have a profound effect on its *obverse*.

Kirk Iverson provided a dazzling example of the use of *obverse*. The problem is to segregate a list of integers, say 4 2 2 4 1 4 8, into boxes with empty boxes for those integers in sequence not present. Boxing like items is achieved by using the vector itself as a *key* (/.)

```
   k=.4 2 2 4 1 4 8
   k</.k
┌─────┬───┬─┬─┐
│4 4 4│2 2│1│8│
└─────┴───┴─┴─┘
```

Now augment the vector with say i.10 :

```
     aug=.(i.10)&,
     aug k
0 1 2 3 4 5 6 7 8 9 4 2 2 4 1 4 8
```

and finally add an *obverse* which *behead*s under box so that the artificially added `i.10` is wiped out item by item each from its own box:

```
   aug=.(i.10)&, :. (}.&.>)
```

Finally do the initial box by key under `aug` which achieves both numerical ordering and also empty boxes where appropriate :

```
   </.~&.aug k
┌┬─┬───┬┬─────┬┬┬┬─┬┐
││1│2 2││4 4 4││││8││
└┴─┴───┴┴─────┴┴┴┴─┴┘
```

Previous paragraphs raise the question of which verbs have 'reasonable' inverses, to which the answer is - quite a lot. First there are self-inverse monadic verbs for which an even number of applications amounts to 'do nothing'. These are

arithmetic `+` `-` `-.` (*not*) `%` `%.` (*matrix divide*)
structural `|.` (*reverse*) `|:` (*transpose*) `/:` (*grade up*) `[` (*left*) `]` (*right*) and `C.` (*cycle-direct*) (see E #30)

Next here is a set of monadic verbs which have inverse partners, that is for each of these verbs v^:_1 is equivalent to its partner :

arithmetic
<: and >: (*increment /decrement*)
+: or +~ and -: (*double/halve*)
*: or *~ and %: (*square/square root*)
^ and ^. (*power/log*arithm)
#. and #: (*base/anti-base* = decode/encode in APL)
o. and %&(o.1) (multiply by π/ divide by π)
j. and %&0j1 (multiply by srqt(-1)/divide by sqrt(-1))
+. and j./ (vector from/to complex no.)

structural
< (*box)* and > (*open*)
;~ (reflex-raze) and >@{. (*open* following *head*)
\: and /:@|. (*grade down* and *grade up* following *reverse*)
;: (*itemise*) and {. (*head*)
". (*do)* and ": (*format*)

There are also families of monadic verb pairs formed by 'currying' dyadic verbs with a noun:

+&n and -&n (add/subtract constant)
*&n and %&n (multiply by/divide by constant)
n&^ and n&^. (power of n/log to base n)
n&o. and _n&o. (for appropriate n, e.g. sin and arcsin)
n&|. and _n&|. (shift n left/ shift n right)
p&|: and (ip&|:)
(ip= inverse permutation applied to *transpose*)

The generic property of the inverse i of a monadic verb v is that (i@v) is equivalent to] .

The 'scans' of five arithmetic verbs, namely + * % = and ~: can be reversed by ^:_1, that is each of these verbs has the property that v/\^:_1 is equivalent to] .

Dyadic verbs have potentially two kinds of inverse, left and right. For example the dyadic + has a right inverse function - because a+b-b = a, that is subtraction on the right 'cancels out' addition. However dyadic + has no left inverse because there is no function i for which a i a+b = b for all a and b. By the same reasoning minus has a left inverse minus and a right inverse plus. Inverses for dyadic verbs are rarer than for monadics. The following is a list of them :

	+	-	*	%	^	^.	=	~:
Left inverse		-		%	^.	^	=	~:
Right inverse	-			*			=	~:

A left inverse i is characterised by the fact that the fork [iv is equivalent to] and a right inverse by iv] being equivalent to [.

Adverse, like *obverse*, is used to define a two-sided verb, but now it provides an alternative verb, the one you use 'in adversity' when the first one fails. It is unlikely to be an inverse, (although in principle it could be!) If the first verb fails, then the *adverse* takes over. Here, for example, is a protection device against 'index error'

```
   3 4{'ABCDE'
DE
   3 5{'ABCDE'
index error
   From=.{ :: ]
   3 5 From 'ABCDE'
ABCDE
```

One obvious circumstance in which you might think this could be applied is

```
    div=.% :: 1:
```

to prevent attempted division by zero but this doesn't work because division by 0 is not an error in J - the result is infinity (_). The only circumstance in which 0 would be delivered is on an attempt to divide characters, or two vectors of unequal lengths, both greater than 1.

Since they connect verbs, *obverse* and *adverse* are conjunctions, but unlike the more commonly used conjunctions they define 'flip-sides' rather than creating fundamentally new composite verbs. To summarise, an *obverse* is a user-defined inverse, *adverse* is the emergency exit to forestall error conditions.

Code Summary

```
    dtb=:({.~i.&' ') :. (,&',')   NB. delete trailing blanks
    frtab=.+/"1@=                 NB. frequency list
    convert=.# i.@#               NB. inverse of frtab
    ptok=:>:@(*&0.01)             NB. e.g. 5% to 1.05
    mut=.,&'%'@(":@ktop)          NB. enrich ktop with %
    ktop=:%&0.01@<:               NB. inverse of ptok
```

```
gm=.(+/%#)&.^."_                 NB. geometric mean
From=.{ :: ]                     NB. from with error trapping
aug=.(i.10)&, :. (}.&.>)         NB. boxed integers with blanks
```

13. If you think J is complex try j

Principal Topics : `j.` (imaginary/complex) `r.` (angle/polar) **`o.`** (pi times/circular function), `+.` (GCD/real+imaginery), `*.` (LCM/length+angle), Cartesian and polar coordinates, complex conjugates, groups, reflections, rotations, de Moivre's theorem. complex powers and logarithms, determinants, inner product, matrix multiplication, quaternions, hypercomplex numbers.

This article is about the facilities available in J for handling complex numbers, something which is greatly helped by a few simple diagrams. When 'talking maths', the representation i will be used to denote the square root of -1 , which translates into `0j1` in J.

1. The two complex number constructors

Although complex numbers are readily input using numeric constants, e.g. `12.5j_7.9` , in meaningful applications the components are more likely to be expressions from which complex numbers are constructed. J has two tools for constructing complex numbers, namely **`j.`** and **`r.`** These correspond to their two possible representations, namely as 2-lists of Cartesian (that is x-y) coordinates, and as 2-lists of polar coordinates (that is {length, angle}). The way in which **`j.`** and **`r.`** work is illustrated by

```
   (2 j.1),(2 r.1)  NB. the two complex no. constructors
2j1 1.0806j1.68294
```

The second of these results shows that 2 times the coordinates of the end point of a radius of the unit circle at an angle of 1 radian are approximately (1.08,1.68). For primary input in the form of 2-lists use *insert*:

```
   (j./2 1),(r./2 1)
2j1 1.0806j1.68294
```

Informally **`j.`** compresses x-y coordinates into complex numbers, and **`r.`** converts polar representation to complex number form. Monadic **`j.`** is dyadic **`j.`** with a default left argument of 0, while monadic **`r.`** is dyadic **`r.`** with a default left argument of 1. It is not a coincidence that these defaults are the identity elements of addition and multiplication. **`r.k`** where **`k`** is real returns the Cartesian coordinates of the point on the unit circle whose polar coordinates are (1,k), for example

```
   r.1              NB. coords of radius at 1 radian
```

```
0.540302j0.841471
```

r.k is represented by e^{ik} in maths and thus by **^j.k** in J, an operation also available through the circle verb as **_12 o.k**. The fact that **r.** and **^j.** are synonyms links the two complex number constructors. More generally the circle functions with arguments in the ranges {9,..12} and {_9,.._12} are directly relevant to complex number construction, and they too have synonyms which will emerge as the discussion continues. The equivalence of **r.** and **^j.** will come to the fore later when discussing complex powers and logarithms.

2. The complex number deconstructors

The construction process is reversed (that is complex numbers are converted back to 2-lists) by `+.` for Cartesian coordinates and `*.` for polar coordinates :

```
   +.2j1 1.0806j1.682941          NB. +. reverses j./
     2        1
 1.0806 1.68294
   *.2j1 1.0806j1.682941          NB. *. reverses r./
2.23607 0.463648
     2        1
```

The circle verb provides the opportunity to obtain the components of `+.` and `*.` one by one:

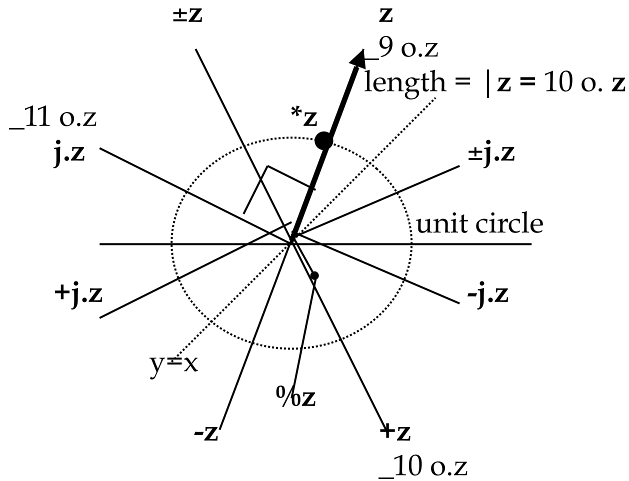
```
   9 11 o.2j1          NB. 9 o. is x , 11 o. is y
2 1
   10 12 o.2j1         NB. 10 o. is length, 12 o. is
angle
2.23607 0.463648
```

The following is a 'rule of thumb' table which summarises the meanings of the circle verbs and incorporates the above ideas :

n o.			**n o.**
_9	identity		
_10	conjugate		
	construct	deconstruct	
_11	**j.**	**+.**	9,11
_12	**r. (^j.)**	***.**	10,12

3. Monadic operations with complex numbers

J provides alternative routes for several common complex numbers operations. In the diagram below a complex number **z** is represented by the arrowed line, and other points represent represent the results of the fundamental monadic arithmetic operations of addition, subtraction, multiplication and division, separately and in combination with **j.** as well as those of the circle functions which are synonyms.



The symbol ± is used to denote either of the equivalent verbs **+@-@j.** or **-@+@j.** . The points **z ±z -z +z** represent a rectangle formed by reflections in the x and y axes with vertices visited anti-clockwise, while the points **j.z ±j.z -j.z +j.z** represent a rectangle formed by reflections in the diagonal axes with vertices visited clockwise. In addition to the three circle function synonyms shown for circle functions, **_12 o.** is a synonym for **r.** .as noted earlier.

The symmetries of rectangles can be represented by groups of verbs of order 4 in which **I** is the identity verb :

Rotations	**{I – j. -j.}**	**j. -j. =** anticlock/clockwise rotations of π/4
Reflections	**{I – + ±}**	**+ ± =** reflections in main axes,
	{I – ±@j. +@j.}	**±@j. +@j. =** reflections in diagonal axes

From these as starting points the full order 8 group table for the symmetries of the square can easily be obtained (see E #26 "Working in Groups").

4. Basic dyadic operations

The basic operations **+ - * %** behave as expected and rules such as the following are obeyed :

```
   |5j2*3j4                  NB. modulus of a product is ..
26.9258
   (|5j2)*(|3j4)             NB. .. the product of moduli
26.9258

   12 o. 5j2*3j4             NB. the angle of a product ..
1.3078
   +/12 o. 5j2 3j4           NB. .. is the sum of the angles
1.3078
```

Also

```
   +/5j2*3j4
7j26
```

is equivalent numerically to $\begin{pmatrix} 5 & -2 \\ 2 & 5 \end{pmatrix}\begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 7 \\ 26 \end{pmatrix}$ and $\begin{pmatrix} 5 \\ 2 \end{pmatrix}\begin{pmatrix} 3 & -4 \\ 4 & 3 \end{pmatrix} = \begin{pmatrix} 7 & 26 \end{pmatrix}$ showing that multiplication of complex numbers is equivalent to the inner product `+/ .*` for matrices of the form $\begin{pmatrix} a & -b \\ b & a \end{pmatrix}$. When multiplication takes the angle outside the range [-π,π], `*.` and **`12 o.`** automatically make a wraparound to bring the angle back into range, for example

```
   wrap=.monad :0
t=.(o.2)|y
if.t>o.1 do.t=.t-o.2 end.
)
    +/12 o._5j2 _3j4         NB. sum of angles exceeds pi
4.97538
   12 o._5j2*_3j4
_1.3078
   wrap 4.97538              NB. 4.975.. + 1.307.. = 2pi
_1.30781
```

Complex numbers raised to real powers are the subject of de Moivre's theorem. This depends on the fundamental relation

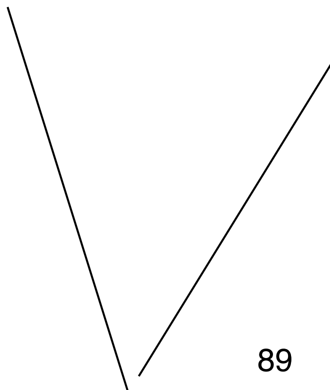
$e^{i\theta} = \cos\theta + i\sin\theta$ which in J expresses the equivalence of the verbs `^@j.` and `j./@(2 1&o.)`. De Moivre's theorem says that $(re^{i\theta})^n = r^n e^{in\theta} = r^n\{\cos n\theta + i\sin n\theta\}$ so to raise complex **z** to the power *n* its modulus should be raised to the power *n* and its angle multiplied by *n*. To see this in action raise `2j1` to the powers 1, 2 and 8 in first cartesian and then polar form :

```
   +.2j1^1 2 8
   2    1                   NB. modulus = sqrt(5)
   3    4                   NB. modulus = 5
_527 _336                   NB. modulus = 625 = 5^4

   *.2j1^1 2 8
2.23607 0.463648            NB. (length,angle) for 2j1
      5 0.927295            NB. (length,angle) for 2j1 ^2
    625   _2.574            NB. (length,angle) for 2j1 ^8
```

The angle in the last line above can be confirmed by

```
   wrap 8*0.463648
_2.574
```

It may be tempting to use the circle function **`_3 o.`** (arctan) to obtain angles, but this only works in simple cases because the range of arctan is [-π/2,π/2]. The range for complex numbers is double this because arctan makes no distinction between (-x)/y and x/(-y), whereas the difference between second and fourth quadrants is significant in dealing with complex numbers.

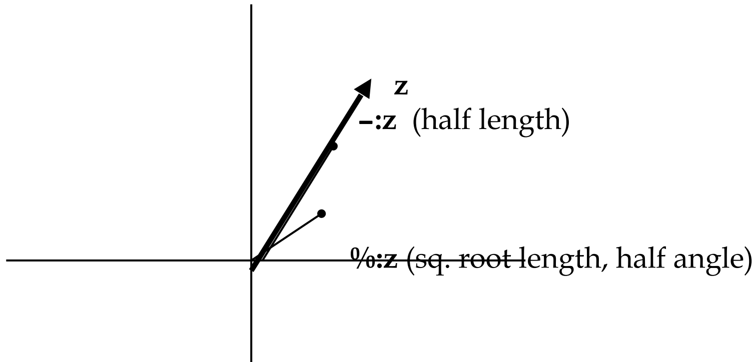


5. The enhanced arithmetic operations

The second diagram illustrates the actions of the J verbs which are obtained from the basic arithmetic operations by adding 'colon' to make a di-gram :

***:z** (squared length, double angle)

+:z (double length)



Only one of these four forms, namely **%:** extends to the dyadic case, for example **4%:z** means (4th root of length, 1/4 angle).

In the case of di-grams formed by adding full stop **`1-.z`** is the same as with real numbers and **%.** and **%** are exactly equivalent. If either **`*.`** or **`+.`** are applied to real scalar numbers the results are 2-lists made by joining zeros. This can be used as a method of stitching 0s as in

```
   +.3 4
3 0
4 0
```

With real numbers dyadic **`*.`** and **`+.`** are LCM and GCD respectively, but these should not be used with complex arguments in the expectation of obtaining the separate GCDs and LCMs of their components. For example

```
   (5j6 +. 10j3),(5j6 *. 10j3)
0j1 75j_32
```

Complex powers and logarithms

To understand complex powers start with the synonym relationship between **`r.`** and **`^j.`** (or **`_12 o.`**) which at first sight should lead to **`j.`** and **`^.r.`** also being synonyms. This is indeed true in some cases :

```
   (j.2j1),(^.r.2j1)
_1j2 _1j2
```

but not always :

```
   (j.12j10),(^.r.12j10)
_10j12 _10j_0.566371
```

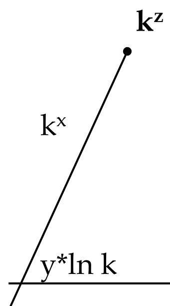
This is because, unlike in real arithmetic, the logarithm of a complex number is not a single valued function. In Cartesian coordinates x and y values stretch out indefinitely in both directions, but in polar coordinates angles wrap around in cycles of 2π in the manner defined by the verb `wrap` above. In mathematical notation

$$\ln(z) = \ln(r\exp(i\pi\theta)) = \ln(r) + i(\theta + 2k\pi) \text{ where k is an integer.}$$

As a matter of arbitrary (but natural!) choice J returns the unique angle which lies in the range $[-\pi,\pi]$. The same wrapping process applies when real numbers are raised to complex powers :

```
   *.2^4j4.5 4j4.6
16  3.11916          NB. unwrapped
16 _3.09471          NB. wrapped
```

More specifically, if k is real and z=x+iy, then k^z is illustrated by




```
   *.2^2j1 3j2
4 0.693147           NB. k to power x, y times ln(k)
8  1.38629           NB. with angle doubled
```

The cases 'complex raised to real' (de Moivre's theorem) and 'real raised to complex' have now been covered leaving only the case 'complex raised to complex' to be dealt with. An interesting starting point is the number i^i which at first sight should be about as complex as it gets :

```
   0j1^0j1          NB. i to the power i
0.20788
```

Not so! To explain this result, considering first $\ln(i^i)$ = i times ln(i). Using the formula $\ln(r\exp(i\pi\theta)) = \ln(r) + i(\theta + 2k\pi)$, and choosing k=0 (as J does) to make the logarithm single-valued, gives $\ln i = 0 + i\pi/2$ which when multipled by i gives $-\pi/2$. i^i must therefore be $e^{-\pi/2}$ which has the value 0.20788 to 5 decimal places. This sequence of calculations is confirmed by

```
   ^-o.0.5
0.20788
```

The verb `ln` fulfils the familiar 'reduce multiplication to addition' property of logarithms of real numbers, that is log(ab) = loga + logb, for example :

```
   ln 1j2*3j4       NB. ln ab
2.41416 2.03444
   +/ln 1j2 3j4     NB. ln a + ln b
2.41416 2.03444
```

For powers where both w and z are fully complex (that is, have non-zero imaginary parts) the following sequence of equivalences $w^z = (e^{\ln w})^z = (e^z)^{\ln w} = e^{z\ln w}$ leads to, for example

```
   2j1^1j3          NB. 2j1 to the power 1j3
_0.537177j0.145082

   ^1j3*j./ln 2j1   NB. e to the power ln w
_0.537177j0.145082
```

It is not easy to visualise the relationship of w^z to w and z diagrammatically, that is the link of `_0.537177j0.145082` with `2j1` and `1j3` is a numerical rather than a graphical one. The same is true for other functions which can accept complex arguments, for examples trig ratios and their inverses. Also the logarithms concerned must be to the base e. e is one of the five most fundamental numbers in the universe, namely 0, 1, e, π and i, which are connected by the equation $1 + \exp(\pi i) = 0$. It is reasonable to suppose that advanced intelligent communicators from outer space (if such there be) would certainly try

to convey this set of numbers to us an immediate 'lingua franca'. This equation can be expressed in J in the following three equivalent ways :

```
   (1+^o.j.1),  (1+_12 o.o.1),  (1+r.o.1)
0 0 0
```

The equation $1 + \exp(\pi i) = 0$ can be rewritten $\ln(-1)= \pi i$ which, since $\ln(-r) = \ln(r)+\ln(-1)$, means that the natural logarithms of negative real numbers are obtained by appending 'jπ' to the logarithm of the corresponding positive number. For example :

```
   ^.5.2 _5.2        NB. (ln r), (ln -r)
1.64866 1.64866j3.14159
```

Extension to Quaternions

Given a matrix of the form $M = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}$ where *a* and *b* are real numbers, e.g.

```
   M=.2 2$2 _3 3 2
```

and an inner product of `M` with a 2-list such as

```
   M +/ .* 2 _1
7 4
```

the same information could be obtained by multiplying two complex scalars :

```
   2j3*2j_1
7j4
```

Similarly finding the determinant of `M` is equivalent to a couple of operations on a complex scalar :

```
   (det=.-/ .*)M    NB. determinant of M
13
   *:10 o.2j3       NB. sum of squares of 2 and 3
13
```

Thus complex numbers can be used as a means of reducing the rank level of some operations. An obvious question is then : if complex data can reduce rank 2 operations to rank 1 can it correspondingly reduce rank 3 operations to rank 2? This speculation is not unique to J, in fact the question was answered in the mid-19th century by Sir

William Hamilton who discovered that this progression is multiplicative rather than additive, that is that the next step up is not from 2 to 3 but from 2 to 4.

First define a verb which transfoms a 2-list of real scalars into a matrix of the above form (call this a quaternion form) :

```
   r2ltom=.,._1 1&*@|.      NB. matrix from real 2-list
   r2ltom 2 3
2 _3
3  2
```

Next observe that is possible to have a matrix with complex coefficients which nevertheless has a real determinant, for example :

```
   C=.2 2$4j3 6j_2 _6j2 4j3
   det C
39
```

The matrix `C` has the form $\begin{pmatrix} PjQ & _Rj_S \\ RjS & PjQ \end{pmatrix}$ which is the form $\begin{pmatrix} a & -b \\ b & a \end{pmatrix}$ extended to complex elements. Straightforward arithmetic shows that the determinant of a matrix of the above from has a real part ($P^2 - Q^2 + R^2 - S^2$) and an imaginary part 2(PQ+RS)) and so if PQ = –RS as is the case with `C`, the determinant is real, otherwise not. In 'all real' case Q = S = 0 and det(M)=$P^2 - R^2$. If the form is now changed to $\begin{pmatrix} PjQ & _RjS \\ RjS & Pj_Q \end{pmatrix}$, that is $\begin{pmatrix} a & -\bar{b} \\ b & \bar{a} \end{pmatrix}$ where the overbars denote complex conjugates, then the determinant is arithmetically guaranteed to be real for all values of *P, Q, R* and *S*. If this is now taken as a standard form then four fundamental matrices obtainable by setting each of *P, Q, R* and *S* to 1 and the other three to zero correspond to unit points on the axes of a four dimensional geometrical space as denoted by (1,0), (0,1), (i,0) and (0,i). A verb analogous to `r2ltom` which constructs the above matrix from a 2-list of complex scalars is

```
   c2ltom=.,.+@|.@(*&1 _1) NB. matrix from complex 2-list
   c2ltom 2j3 4j5
2j3 _4j5
4j5 2j_3
```

Define variables to correspond to (1,0), (0,1), (i,0) and (0,i) :

```
   ]'I i j k'=.c2ltom &.> 1 0 ;0 1;0j1 0;0 0j1
┌───┬────┬─────────┬────────┐
```

```
│1 0│0 _1│0j1    0│   0 0j1│
│0 1│1  0│  0 0j_1│0j1    0│
└───┴────┴────────┴────────┘
```

and self-multiply each of these matrices :

```
   times=.+/ .* each              NB. matrix multiply
   times~ i;j;k                   NB. squares of i,j and k
┌─────┬─────┬─────┐
│_1  0│_1  0│_1  0│
│ 0 _1│ 0 _1│ 0 _1│
└─────┴─────┴─────┘
   times~^:2 i;j;k          NB. 4th powers of i,j and k
┌───┬───┬───┐
│1 0│1 0│1 0│
│0 1│0 1│0 1│
└───┴───┴───┘
```

which shows that $i^2 = j^2 = k^2 = -I$ and $i^4 = j^4 = k^4 = I$ where I is the identity matrix. Now multiply `i,j` and `k` by each other :

```
   (i;j;k)times(j;k;i)      NB. result is k;i;j
┌───────┬────┬────────┐
│  0 0j1│0 _1│0j1    0│
│0j1   0│1  0│  0 0j_1│
└───────┴────┴────────┘

   (j;k;i)times(i;j;k)      NB. result is (-k);(-i);(-j)
┌─────────┬────┬─────────┐
│   0 0j_1│ 0 1│0j_1    0│
│0j_1    0│_1 0│   0  0j1│
└─────────┴────┴─────────┘

   det&> i;j;k              NB. determinants equal 1
1 1 1
```

If `i,j` and `k` are raised to third powers a further three matrices not previously encountered arise :

```
    ]'ci cj ck'=.(i;j;k)times(i;j;k)times i;j;k
┌────┬────────┬─────────┐
│ 0 1│0j_1   0│   0 0j_1│
│_1 0│   0 0j1│0j_1    0│
└────┴────────┴─────────┘
```

but now, however much the set of eight matrices

```
        I,-I,i,j,k,cj,ck,ci
```

are intermultiplied, the result is always another member of the set, for example :

```
   (i;j;k)times cj;ck;ci          NB. result is ck;ci;cj
┌───────────┬─────┬────────┐
│    0 0j_1 │  0 1│0j_1   0│
│0j_1     0 │_1 0 │   0 0j1│
└───────────┴─────┴────────┘
```

The set of eight matrices possesses the properties of a **group**, more specifically the **quaternion group**. Although there are eight elements in the group these are all related to each other, and the whole set of seven excluding the identity matrix can be generated from any two. For example if `i` and `j` are chosen as generators `k` is `ij`, `ci` and `cj` are defined as powers of `i` and `j`, and `ck` is `cj` multiplied by `ci` The seventh matrix is the common value of i^2 and j^2.

If, analogous to `j`, J were to contain two further independent number constructors `k` and `m` such that `2j3k4m5` was a scalar, and the same rules $i^2 = j^2 = k^2 = -I$ and $i^4 = j^4 = k^4 = I$ applied where I= `1`, `i=0j1k0m0`, `j=0j0k1m0`, `k=0j0k0m1` then these scalars would be recognised mathematically as **hypercomplex numbers**. The four basic hypercomplex numbers for which the real elements are (1 0 0 0), (0 1 0 0), (0 0 1 0) and (0 0 0 1) would follow the same multiplication structure as the set of eight matrices, and form a group isomorphic with the quaternion group.

A scalar such as `0j3k4m5` whose first element is zero corresponds to a **pure quaternion** and so pure quaternions exactly match points in three dimensional space, or quantities such as E, B, H which define electro-magnetic fields as three dimensional entities. Such equivalences are of course only useful if the operations employed on them are guaranteed to produce further pure quaternions.

Code Summary

```
  det=.-/ .*                 NB. determinant
  r2ltom=.,._1 1&*@|.        NB. quaternion matrix from real 2-
list
  c2ltom=.,.+@|.@(*&1 _1)  NB. quat matrix from complex 2-list
  times=.+/ .* each          NB. matrix multiply
```

14. j is complex? You bet!

Principal Topics : ? (*roll*) \: (*grade down*), j. (*complex*), E. (*member of interval*) I. (*interval index*). simulation, odds, bookmaker's odds, true odds, overround, normalisation, fantasy betting, betting methods, betting systems, d'Alembert's system, Martingales, negative exponential, weighted random numbers, random races and race cards.

j doesn't necessarily mean complex!

Although j in J usually means 'complex', numbers of the form `7j2` can model other forms of duple, for example the left argument of *format* (`":`) where the elements denote field width and number of decimal places. Another possibility is odd ratios, for example `7j2` can model odds of 7 to 2 (that is 7 to 2 against), from which fractional odds (`fro`) are obtained as

```
   fro=.({: % +/)@:+."0    NB. +. transforms ajb to a b
   fro 3j1
0.25
```

`%fro` then gives what is returned (winnings and stake) after a successful unit bet. The accumulated fractional odds of a field of three in which the odds offered by a bookmaker are evens, 3-1 and 7-2 is

```
   (+/@:fro)1j1 3j1 7j2
0.9722
```

Of course (pun intended!) bookmakers and betting shops see to it that such a sum is never less than 1, the excess over 1 being what the bookmaker creams off as markup or **overround**. In practice, <u>real</u> probabilties, that is the <u>absolute</u> probabilties of the various horses winning, vary dynamically right up to the final minutes before a race. Real probabilties reflect the many technicalities of racing as a sport such as the assessment of horses, jockeys, trainers, weather, racetrack condition, even insider trading and corruption, all of which lends a certain naivety to the fact that some of the observations on which this article is based are derived from the single sets of static odds quoted in the racing pages of daily newspapers. However, broad conclusions can be drawn, for example that in UK horse racing overround seems to average between 25% to 40%. (It goes without saying that should any reader discover a race card for which the `+/@:fro` is less than 1, he or she should immediately raise every possible penny to place bets on all horses in multiples of `fro`, and even more importantly should as a matter of duty contact me urgently!)

Add a couple of horses to the above field to make matters more realistic :

```
   (+/&:fro) fld=.1j1 3j1 7j2 5j1 8j1
1.25
```

giving an overround is 25%. Thus if bets are placed in the proportions `fro fld` then the cost of betting on all horses will be 1.25 for an assured return of 1 and an assured gain to the bookmaker of 0.25.

Assuming that the bookmaker's quoted odds reflect his view of the relative probabilities of the horses winning the race, the underlying **true odds** are :

```
   to=.(%+/)&:fro          NB. true odds
   to fld
0.4 0.2 0.1778 0.1333 0.08889
```

(Technical note : the *hook* `%+/` normalises a list so that the total of its elements is 1.)

`to` also demonstrates the extent to which the bookmaker downgrades odds in order to achieve overround, e.g. the horse quoted at evens has in fact a probability of 0.4 of winning the race. Also the returns (that is, including the original stake) multiplied by the true odds remain the same whichever horse wins the race :

```
   (to * %@fro)fld
0.8 0.8 0.8 0.8 0.8
```

namely the reciprocal of the overround. The bookmaker accepts bets to create a **book**, on which he reckons to make the overround as profit whatever the outcome of the race.

Random real probabilities totalling 1 are generated by

```
   rnd=.?@#&0        NB. random uniforms in {0,1}
   rrp=.(%+/)@:rnd   NB. random real probabilities
   rrp 5
0.176 0.28 0.047 0.232 0.265
```

Using these and a book based on `fld`, the bookmaker's long term income and outgoings based on horses winning with random probabilities are given by

```
   book=.40 20 18 13 9
   (+/book),+/book*(rrp 5)*%fro fld
100 79.85
```

Significant risk to the bookmaker arises only if *both* his book and the real probabilities change. Bookmaker's arithmetic is a continuous process with input parameters : **current book, real probabilities, current actual odds** in which he strives to adjust his quoted odds in order to keep the book in balance, and thereby his profits secure. Incomings and outgoings can be formalised in a verb whose left argument is **book;real probabilities**, and whose right argument is **current actual odds**. A balanced book would be simply a multiple of the real probabilities. The example below shows how real probabilities make little difference to the bookmaker's expectations even if public assessment of the race shifts dramatically in favour of the outsider :

```
   inout=.dyad : '(+/>{.x),+/*/(>x),%fro y'
   book_rp=.40 20 18 13 9;0.2 0.1 0.1 0.1 0.5
   book_rp inout fld
100 80.4
```

However, suppose that the outsider attracts a large number of bets :

```
   book_rp=.40 22 18 13 50;0.2 0.1 0.1 0.1 0.5
   book_rp inout fld
143 265.7
```

This gives the bookmaker a projected loss of 123. His options are (1) to sustain his previous belief in the relative propabilities but reduce exposure to the new favourite by reducing its quoted odds, in the hope that future bets on the other horses may help to recoup his losses :

```
   book_rp=.40 22 18 13 0;0.2 0.1 0.1 0.1 0.5
   book_rp inout 1j1 3j1 7j2 5j1 1j2
93 40.7
```

or (2) to accept the new real probabilities and requote all his odds based on these. The primitive verb `j.` transforms `a b` to `ajb`, that is fractions back to odds :

```
   (j./@:(%/,-.))0.78
0.78j0.22
```

(The *hook* `,-.` returns a fraction joined to its 1s complement)

However, it is more satisfactory to have odds in the form `1j`**x** or **x**`j1` so define

```
   odds=.monad :0
```

```
t=.%/y,1-y
if.t>1 do.r=.>.1,t
else. r=.<.(%t),1 end.
r=.j./r
)
   odds"(0)0.5 0.25
1j1 3j1
```

(Note : Any rounding favours the bookmaker which seems quite reasonable given that odds can never be a scientifically precise measure.)

The bookmaker might choose to revise his true odds and apply an overround of about 25% to give

```
   odds"(0) 0.2 0.1 0.1 0.1 0.5*1.25
3j1 7j1 7j1 7j1 1j2
```

It is not suggested that bookmakers carry out any such arithmetic formally, although the above presumably models roughly the nimble calculations which they instinctively perform.

Beating the Bookie

Given the inherent bias in favour of the bookmaker, are there any ways by which the better can possibly turn the situation to his advantage? First assume that he has some technical knowledge of which he feels reasonably assured and believes to be superior to that of the bookmaker.

Since `%fro fld` gives the returns for a unit bet the returns for any list of bets are

```
    rets =.[*%@:fro@]       NB. left argument = bets
    1 1 1 1 1 rets fld
2 4 4.5 6 9
```

Suppose now that as a matter of judgement the better believes that the race will certainly be won by one of the two favourites with probabilities in proportion 3:2. His expected returns for a bet which reflects this are

```
   6 4 0 0 0 rets fld
12 16 0 0 0
```

that is, for a total outlay of 10 he will achieve a return of either 12 or 16 or 0. His expectation, using true odds, is (0.4× 6) + (0.2× 12) = 4.8 which would give the bookmaker an expected gain of 5.2. However the expectation based on his own judgement is (0.6×12) + (0.4×16) =

13.6, and so if he has complete confidence in his judgement and behaves rationally, it would be senseless for him not to bet, nor indeed would he be unhappy if one of the unbacked horses won, since he would still have achieved value for his money in the same sense that an insurance policy on which no claim is made has nevertheless provided valuable cover.

Alternatively the better might choose to use the judgement of others, e.g. newspaper tipsters. What are the net gains or losses resulting from a unit bet on every tipster recommendation for a given day? On a day in which 20 races were run and four winners were tipped at 4-1, 11-4, 7-2 and 4-1, the net gain achieved for unit bets placed by following a tipster was given by :

```
   tips=.-~+/@:(1&rets)@ ]
   20 tips 4j1 11j4 7j2 4j1
_1.75
```

that is an overall loss of -1.75. Empirical evidence using the racing correspondents of the *Times* and the *Daily Telegraph* shows that following tipsters' advice consistently is very rarely profitable, and even then only when a winner happens to be picked at unusually long odds.

Turn now to manipulating probabilities, are there any techniques based on probability alone which can swing the bias away from the bookmaker towards the better? Such a possibility is demonstrated by the so-called **Martingale** in which a stake is progressively doubled for a losing bet and betting stops on a winning one. In a fair game at evens, e.g. coin tossing with bets on a tail, a tail is bound to occur eventually, at which point there is a net gain of one original betting unit. The problem is that the certainty of winning requires unbounded available capital.

Fantasy betting

The safest way for the novice to take his first steps into the world of betting is to use his computer to estimate and simulate the forces he will encounter in the real world in which real money changes hands. First generate random uniform integers using *Interval Index* `I.` to transform each of the numbers in `rnd` into a serial number of one of the intervals defined by the left argument.

```
   wrnd=.(+/\)@[ I. rnd@]  NB. weightd random integers
   >:(to fld)wrnd 10
1 3 3 3 1 4 1 1 2 3
```

thus in 10 reruns the first and third horses each won 4 times, the second and fourth horses won once and the fifth horse not at all.   To count frequencies arising in such runs say

```
   +/"1 (i.#fld) =/ (to fld)wrnd 10
2 3 4 0 1
```

which can be consolidated in a verb where the right argument is the number of reruns :

```
   rerun=.dyad :'+/"1 (i.#x) =/ (to x)wrnd y'
   fld rerun 20
10 5 2 2 1
```

that is the favourite won exactly half of the time in the above simulated sequence.

A simulated race with between 5 and 17 runners each of which consists of drawings from a negative exponential distribution with mean 1.25 is given by

```
   sortd=.{~\:
   rne=.[ * ^.@%@rnd@] NB. random negative exponential
   odds"(0)0.0475>.sortd (%+/)1.25 rne 10
5j1 5j1 6j1 8j1 9j1 10j1 12j1 12j1 14j1 20j1
```

(Note : There is no special reason for using the negative exponential distribution other than that it appears empirically to give lists of odds which look tolerably similar to those actually printed daily in the sporting pages. `0.0475>` is to ensure that no odds are greater than `20j1`)

It is convenient to head each list with the sequence number of the randomly drawn winner (favourite = 1, etc.).

```
   rrace=.monad :0                NB. random race
r=.odds"(0)t=.0.0475>.sortd (%+/)1.25 rne 5+?13
r=.(>:(+/\ (%+/)t) I. rnd 1),r    NB. join random winner
)
   rrace 10
2 1j1 3j1 12j1 13j1 17j1 18j1 20j1 20j1 20j1 20j1 20j1 20j1
```

A random race card with 3 races is then given by

```
   rrcard=.monad : '>rrace every 5+?y#13'
   rrcard 3
3 2j1 2j1 6j1  6j1  7j1 19j1    0    0    0    0    0    0
1 2j1 3j1 9j1 11j1 14j1 15j1 16j1 17j1 19j1 19j1 19j1 19j1
```

```
1 1j1 5j1 5j1  7j1 10j1 18j1 19j1 19j1 19j1 19j1    0    0
```

Betting Methods

Various methods can be employed when a bet on a single race is placed, for example the favourite can be backed, or a pin stuck in the race card, or a horse chosen at random but with weights applied based on the quoted odds. These three possibilities are described respectively in

```
   method=.dyad :0
r=.i.0 [ i=.0
while.i<#x do. t=.i{x                    NB. loop through races

select. y
  case. 1 do. b=.1                       NB. bet on favourite
  case. 2 do. b=.>:?<:#t                 NB. stick a pin in race
card
  case. 3 do. b=.>:(fro }.t)wrnd 1       NB. random, wts=odds
end.
if.(b={.t)do. r=.r,(<:{.t){%fro }.t     NB. win
else. r=.r,0 end.                        NB. lose
i=.i+1 end. r
)
```

The experiments which follow are based on a hypothetical race meeting where between 5 and 17 horses ran in each of 1,000 races, with a simulated 25% overround.

```
   rc=.rrcard 1000
   +/"1>(<rc)method every 1 2 3
752.8 503.3 772
```

gives the total winnings on a unit stake in each race. Thus for each method 1,000 units of were staked, and apart from method 2, the totals in the above run converge towards a value of 800. Repeated reruns with further race cards show consistency in the case of methods 1 and 3 but considerable variability with method 2, which rarely comes even close to 800 – in other words, random selection is likely to be a worst case strategy in the long run! That said, the methods were applied to three real race meetings at Ripon, Carlisle and Newton Abbot with 7, 7 and 6 races respectively with results :

Ripon :	9 0 0
Carlisle :	7.375 16 0
Newton Abbot	11.1 0 3.75

showing that even pin-stickers can have their lucky day!

Betting systems

Simulated race cards provide the opportunity for testing out betting systems, that is betting sequences in which stakes change dynamically according to previous results. One such system is due to the 18th. century mathematician d'Alembert. Applying this system the size of the stake is increased by 1 in the case of a losing bet and decreased by 1 in the event of a winning bet. A zero stake is replaced by the original stake. For example, with an initial bet of 5 and a sole win of 5 on the fourth race out of six, the succession of bets was 5 6 7 8 7 8, a total of 41 for a return of 8×5=40 and an overall loss of 1. The following verb simulates the sequence of stakes :

```
   dalem=.dyad :0          NB. x is stake, y is returns list
r=.x [ i=.1
while.i<#y do.             NB. loop through returns
if.(0={:r)do.r=.(}:r),x    NB. if 0 restore initial stake
else.r=.r,({:r)+_1++:0=(<:i){y end.  NB. raise or lower
i=.i+1 end.r
)
   5 dalem 0 0 0 5 0 0
5 6 7 8 7 8
```

Long runs of losers lead to increasingly large stakes developing. Using the 'back the favourite' method on the simulated 1000-race card `rc`, the total returned is

```
   t1=.rc method 1
   +/(*5&dalem) t1
198252.2
```

for total stakes of

```
   +/5 dalem t1
256230
```

198,252/256,230 = 77.4% which is little different from straightforward constant bets. The corresponding figures for methods 2 and 3 are 211,035/445410 = 47.4% and 267065/331,278 = 80.6%, indicating again the weakness of 'selecting by pin'. In all cases the figures show how the better runs the risk of a heavy absolute loss using this system when wins are relatively infrequent.

Other systems could be based on patterns of wins and losses for which the primitive verb *Member of Interval* `E.` is helpful. For example if a constant bet of 5 is made only after observing a 'win-lose' sequence, define

```
    wl=.3 :'0 0,_2}.1 0 E.y~:0'
    wl 1 0 0 0 1 1
0 0 1 0 0 0
```

In this case bets would in the long run be placed only part of the time:

```
    +/wl 0~:t1=.rc method 1
185
    +/(wl t~:0)#t1
131.33
```

131.3/185 = 71.0% and the corresponding percentages for methods 2 and 3 were 55.5% and 90.0% respectively. The practical message is that neither of the above systems offers the better much hope of advantage in the long run. However, having so much experimental possibility available at home makes things significantly easier to organise than a day at Aintree or Goodwood, and a good deal cheaper too – have a great day in!

Code Summary

`ajb` is interpreted as 'odds of a to b against'.

```
fro=.({: % +/)@:+."0        NB. fractional odds from ajb
to=.(%+/)&:fro              NB. true odds (dec) from e.g. fld
   fld=.1j1 3j1 7j2 5j1 8j1

   odds=.monad :0           NB. converts fractnl odds to ajb
t=.%/y,1-y
if.t>1 do.r=.>.1,t
else. r=.<.(%t),1 end.
r=.j./r
)
```

Simulated Racing

```
rnd=.?@#&0                  NB. random uniform (0,1]
rne=.[*^.@ %@ rnd@]         NB. random negative exonential
wrnd=.(+/\)@[ I. rnd@]      NB. weighted random ints
rerun=.dyad :'+/"1 (i.#x) =/ (to x)wrnd y'

   rrace=.monad :0          NB. random race
r=.odds"(0)t=.0.0475>.sortd (%+/)1.25 rne 5+?13
r=.(>:(+/\ (%+/)t) I. rnd 1),r   NB. append random winner
)
rrcard=.monad : '>rrace every 5+?y#13' NB. rand race card
```

Betting Methods

```
   method=.dyad : 0         NB. betting methods, x=race card
r=.i.0 [ i=.0
while.i<#x do. t=.i{x
select. y
  case. 1 do. b=.1          NB. bet on favourite
  case. 2 do. b=.>:?<:#t    NB. stick a pin in race card
```

```
  case. 3 do. b=.>:(fro }.t)wrnd 1   NB. random, wts=odds
end.
if.(b={.t)do. r=.r,(<:{.t){%fro }.t     NB. win
else. r=.r,0 end.                       NB. lose
i=.i+1 end. r
)
```

Betting Systems

```
   dalem=.dyad :0          NB. x is stake, y is returns list
r=.x [ i=.1                NB. initialisations
while.i<#y do.             NB. loop through returns
if.(0={:r)do.r=.(}:r),x    NB. if 0 restore initial stake
else.r=.r,({:r)+_1++:0=(<:i){y end.  NB. raise or lower
i=.i+1 end.r
)
```

15. Cancel, cancel little fraction

Principal Topics : x: (*extended precision*) , numerical data types, conversion rules, rational approximations.

There are various forms of number representation in J, as well as rules and conditions for converting between them. To start with, in complex arithmetic `+.` transforms a complex number in to a list of its components

```
   +.2j3
2 3
```

An analogous problem is that of transforming a rational number into a list {numerator,denominator} following any possible cancellation. This is provided by the primitive verb `x:` with left argument 2 :

```
   2 x: 3r5 6r10 14r7
3 5
3 5
2 1
```

but notice

```
   %/2 x:8r6
4r3
```

does not convert `4%3` into decimal format as you might expect. Instead say either of the following :

```
   _1 x:8r6
1.33333

   x:^:_1 (8r6)
1.33333
```

There are conversion rules which apply in calculations in which the six different data types (Boolean, Integer, Extended Integer, Rational, Floating Point and Complex) are mixed. It is not usually necessary to be explicitly aware of the thirty possible conversions - the main ones are

(1) the presence of just one decimal notated number in a calculation is enough to force decimal notation for the result :

```
   1r1 2r1 3r1 + 3r5 3r5 14r7
8r5 13r5 5
   1r1 2r1 3r1 + 3r5 0.6 14r7
```

```
1.6 2.6 5
```

(2) mixing j and r results in conversion to floating point

```
   1r9j2r3        NB. (1/9) + (2/3)i
0.111111j0.666667
```

Also rational notation cannot be used to perform rational complex division

```
   1j9r2j3        NB. not (1+9i) ¥ (2+3i)
|ill-formed number
```

(3) where present, floating point and complex are dominant. Rational is also "strong" except as under (1),
Boolean and integer are "weak".

```
   2+3j2+7r2
8.5j2
```

The full conversion table is

	Bool	Integer	ExPrec	Ratnl	Float	Cplex
Bool	Bool	Integer	Exprec	Ratnl	Float	Cplex
Integer	Integer	Integer	Exprec	Ratnl	Float	Cplex
Exprec	Exprec	Exprec	Exprec	Ratnl	Float	Cplex
Ratnl	Ratnl	Ratnl	Ratnl	Ratnl	Float	Cplex
Float	Float	Float	Float	Float	Float	Cplex
Cplex	Cplex	Cplex	Cplex	Cplex	Cplex	Cplex

Pure rational calculations perform cancellation automatically

```
   ]t=.+/\1r3 1r4 1r5
1r3 7r12 47r60

   t*/t
   1r9    7r36    47r180
  7r36  49r144   329r720
47r180 329r720 2209r3600

   t +/ .*t
213r200
   1r9 + 49r144 + 2209r3600
213r200
```

Interestingly cancellation can also be done using `+.` in its dyadic role of greatest common denominator.

```
   dac=.%@(1&+.)
```

gives the denominator after cancelling, and multiplying this by the number itself gives the numerator, so `((*dac), dac)` does the job!

```
   ((*dac), dac)8r6
4 3
```

Again subsequent calculations do not convert to decimal format unless such a number appears.

```
   %/((*dac), dac)8r6
4r3
   1.6* %/((*dac), dac)8r6
2.13333
```

To extend to objects with rank, say

```
   ((*dac),"0 dac)8r6 12r2 34r51
4 3
6 1
2 3
```

Some ingenious alternatives have been discussed of which `(,&1)% (+.&1)` proposed by Roger Hui avoids the double calculation of `dac` and has a certain appealing symmetry, extending to `,"0&1 % +.&1` to deal with general rank:

```
  2 x:2 3$8r6 12r2 34r51
4 3
6 1
2 3

4 3
6 1
2 3
```

To obtain close rational approximations to irrational numbers such as π and e , say

```
   x: o.1
1285290289249r409120605684    NB. 1,285,290,289,249 /
409,120,605,684
   _1 x: x: o.1
3.14159
   _1 x: x: ^1
```

```
2.71828
```

Code Summary

```
  dac=.%@(1&+.)
```

16. Thinking by numbers

Principal Topics : #: (*anti-base*) b. (*Boolean*), |: (*transpose*), syllogism, predicate, Boolean verbs, tautology, contradiction.

Two over-riding goals of scientific method are to seek truth and to expand knowledge – this article shows how do both, admittedly in the restricted context of propositions and binary relations.

Propositions and definitions

First consider propositions such as

No Vector reader eats haggis.
All Americans are Vector readers.
No American ever rejects haggis.

Each of these has the form of a **subject** and a **predicate** connected by a **copula** ('is'), for example the subject of the first is a Vector reader and the predicate is a haggis eater. Such propositions include an implicit quantifier ("For all ..") or ("There exists ..."), and are assertions of which a clear binary judgement can be made, true or false. If three such propositions are such that the third (called the **conclusion**) is logically deducible from the first two (called the **premisses**), the resulting set is called a **syllogism**.
In the English language the word 'predicate' has two meanings. Examples of the second meaning are

A: He is a Vector reader
B: He is an American
C: He eats haggis

In this sense a predicate is an assertion containing a **variable** (in these examples 'he') and thus true/false judgements are variable dependent. A list of columns representing as binary digits the numbers from 0 .. 2^y is given by :

```
   binlist=.monad : '|:#:i.2^y'
   t3=.binlist 3
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
```

The columns of `t3` collectively represent all possible selections of 3 binary digits with repetition allowed. In another view `t3` is a list of

lists, each of which contains 2^y binary digits in equal proportions of 0s and 1s. Use these lists to model predicates and define verbs which select from the top level 3-list :

```
   A=.0&{
   B=.1&{
   C=.2&{
   A t3
0 0 0 0 1 1 1 1
```

The Boolean Verbs

`b.`is an unusual quantity. It is not an adverb but behaves somewhat like one, only its argument is an integer (noun) rather than a verb. `b.` is the common basis of a set of strongly related verbs which perform binary calculations. `b.` with no argument means `0 b.`, so for practical purposes a numeric argument must be present as in

```
   and=.1 b.
   or=.7 b.
   (1 and 0);(1 or 0)
┌─┬─┐
│0│1│
└─┴─┘
```

Each binary verb such as `and` and `or` has a 2 by 2 truth table with four entries, e.g.

```
   0 1 and/0 1
0 0
0 1
```

so that there are a total of 2^4=16 distinct truth tables whose ravels correspond to the binary representations of 0 .. 2^y and provide the numbering scheme for the arguments of `b.`.

```
   binlist 4
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

A list of the regularly used binary verbs is

```
   and=.1 b.
   or=.7 b.
   imp=.13 b.        NB. implies
   eq=.9 b.          NB. equals
   xor=.6 b.         NB. exclusive or (same as not equal)
   nand=.14 b.       NB. not and
   nor=.8 b.         NB. not or
```

to which add the verbs

```
   true=.15 b.        NB. all 1s regardless
   false=.0 b.        NB. all 0s regardless
   left=.3 b.         NB. ignore right argument
   right=.5 b.        NB. ignore left argument
```

Combining these with the selection verbs above gives, e.g.

```
   (A and B)t3
0 0 0 0 0 0 1 1
```

which is a ravelled (that is flattened) 3-dimensional truth table for A,B and C.

The Special Case of 'not'
'not' is dealt with separately because unlike the other binary verbs it is monadic. A compound verb such as `not A` is a hook in which the result of the selection verb must be processed by monadic `-.` (*not*) and not by dyadic `-.` (*less*). Imposing such a constraint is one of the main uses of `[:` (*cap*):

```
   not=.[:-.]
   (not A)t3
1 1 1 1 0 0 0 0
```

(note : without *cap* the meaning of `not A` is all but the list `A`

```
   (-. A)t3
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1                  ).
```

It is now possible to compose logical expressions such as

```
  (B imp(not C))t3
1 1 1 0 1 1 1 0
```

and to define a complex predicate involving three basic predicates as a verb to be applied to `t3` :

```
   P1=.(A imp(not B))and(B imp A)and(B imp C)
   P1 t3
1 1 0 0 1 1 0 0
```

Truth and Falsity Adverbs

It is useful to be able to obtain the columns of `t3` which correspond to 1 (truth) and 0 (falsity) through

```
   T=.adverb : '#"1~x'          NB. truth adverb
   F=.adverb : '#"1~(not x)'    NB. falsity adverb
   (P1 T t3);P1 F t3
┌───────┬───────┐
│0 0 1 1│0 0 1 1│
│0 0 0 0│1 1 1 1│
│0 1 0 1│0 1 0 1│
└───────┴───────┘
```

where the first box defines the A, B and C attributes of individuals for whom the three propositions which began this article are consistent, and similarly the second box gives the attributes of those for whom the joint propositions are false.

Sorites

A **sorites** is a series of propositions in which the predicate of one is the subject of the next, and the final proposition (conclusion) can be inferred from the rest. Lewis Carrol delighted in constructing nonsense sequences of this sort such as

> No ducks waltz.
> All my poultry are ducks.
> None of my poultry waltz.

As before construct predicates such as

> A: It is a duck.
> B: It waltzes.
> C: It is one of my poultry.

To confirm that this is a syllogism and thus a sorites, define

```
   P2p=.(A imp(not B))and(C imp A)       NB. p=premisses
   P2c=.C imp (not B)                    NB. c=conclusion
```

Then apply `P2c` to those elements of `t3` for which `P2p` is true :

```
   P2c P2p T t3
1 1 1 1
```

which establishes that the set is indeed a syllogism. However same test applied to the earlier set of propositions at the head of this article

```
   P1p=.(A imp (not C)) and (B imp A)
```

```
   P1c=.B imp C
   P1c P1p T t3
1 1 1 0
```

fails, and the individual which causes failure of the syllogism test is identified by

```
   P1p T t3
0 0 1 1
0 0 0 1
0 1 0 0
```

that is a Vector-reading American who does not eat haggis.

Some Technicalities

The list above covers eleven binary verbs. The remaining five in the set are less immediately interesting :

```
   not_left=.12 b.  NB. ignore right argument
   not_right=.10 b. NB. ignore left argument
   not_imp=.2 b.    NB. not of (x implies y)
   imp_not=.11 b.   NB. (not x) implies (not y)
   not_imp_not=.4 b.      NB. not of imp_not
```

There are 16 fundamental arguments 0 .. 15 for `b.` . To these 16 may be added or subtracted without affecting the result so that the permissible argument range covers _16 to 31. 32, 33 and 34 are also valid but relate specifically to bit/byte arithmetic (that is modulo two arithmetic discarding remainders), namely 32 (=rotate), 33 (shift) and 34 (signed shift).

Advancing Knowledge

The discussions so far have been **analytic**, that is they have consisted in testing combinations for consistency, which does not in itself increase knowledge. However **synthetic** information concerning relationships which are not directly connected in a single proposition amounts to advancing knowledge from what is already known, and can be obtained by defining

```
   AB=.0 1&{
   AC=.0 2&{
   BC=.1 2&{  etc., and

   ]t2=.binlist 2
0 0 1 1
0 1 0 1
```

whose columns (as rows) correspond in order to the marginal value combinations in the 'flat' representation of any of the 16 two by two truth tables. `t2` is used to establish the presence (or absence) of its columns in pairs of rows of a larger truth table, thereby obtaining relationships between the corresponding propositions :

```
   whatrel=.monad : '2 #.t2 e.&|:y'
   P3p=.(A eq B) and (B eq (not C))
   whatrel AC P3p T t3
6
```

6 is the `b.` number for not equal and so the required relation is `A eq (not C)` as is reasonably obvious anyway. For the first set of propositions the B/C relation is given by

```
   whatrel BC P1p T t3
14
```

that is by `B nand C` , or equivalently

```
   CB=.2 0&{
   whatrel CB P1p T t3
13
```

that is `C imp B` .

Tautology and contradiction

These are demonstrated by

```
   P4p=.(A eq B)or (B eq (not A))
   whatrel AB P4p T t2
15
   P5p=.(A eq B)and (A xor B) NB.xor=synonym for
not equals
   whatrel AB P5p T t2
0
```

Further possibilities

Applying the syllogism test to

> All men are good.
> All men are loyal.
> All women are bad.

using

A: It is a man.
B: It is good.
C: It is loyal.

gives

```
   P6p=.(A imp B) and (A imp C)
   P6c=.(not A) imp (not B)
   P6c P6p T t3            NB. syllogism test
1 1 0 0 1
```

which fails for the reasonably obvious reason that the premisses say nothing about women (i.e.not-men). Also there is no explicit connection between goodness and loyalty.

```
   whatrel BC P6p T t3
15
```

that is, goodness and loyalty are synonymous for men (tautology). However

```
   P6=.(A imp B) and (A imp C) and ((not A)imp(not B))
   whatrel BC P6 T t3
13                         NB. says goodness implies loyalty
   whatrel CB P6 T t3
11                         NB. says disloyalty implies badness
```

These last two results do not rule out the possibility of badness and loyalty occurring together (in a woman) as is made explicit by

```
    P6x T t3
0 0 0 0 1
0 0 1 1 1
0 1 0 1 1
```

Makes you think ... !!

Code Summary

```
   binlist=.monad : '|:#:i.2^y'  NB. bin nos to 2^y in cols
   A=.0&{            NB. select first
   B=.1&{
   C=.2&{
   and=.1 b.
   or=.7 b.
   imp=.13 b.        NB. implies
   eq=.9 b.          NB. equals
   xor=.6 b.         NB. exclusive or
   nand=.14 b.       NB. not and
   nor=.8 b.         NB. not or
   true=.15 b.       NB. all 1s regardless
```

```
false=.0 b.       NB. all 0s regardless
left=.3 b.        NB. ignore right argument
not_left=.12 b.   NB. ignore right argument
not_right=.10 b.  NB. ignore left argument
not_imp=.2 b.     NB. not of (x implies y)
imp_not=.11 b.    NB. (not x) implies (not y)
not_imp_not=.4 b.       NB. not of imp_not
right=.5 b.       NB. ignore left argument
T=.adverb : '#"1~x'          NB. truth adverb
F=.adverb : '#"1~(not x)'    NB. falsity adverb
AB=.0 1&{         NB. select first two
AC=.0 2&{
BC=.1 2&{
whatrel=.monad : '2 #.t2 e.&|:y' NB.what relation no?
    t2=.binlist 2
```

17. Jaesthetics

Principal Topics : `/:` (*grade up*) `=` (*self-classify*) `/.` (*key*) `~.` (*nub*) `~` (*reflex*) `;` (*raze*) occurrence numbers

I argue in this article that there are matters of taste involved in writing J, and that J programmers inevitably make choices between pieces of code which are identical as far as effects on data are concerned.

A contributor to the J forum recently made the point that for J aficionados, J phrases can become a more natural medium for expressing ideas that any natural language equivalent. Jim Lucas stated that he didn't care for expressions with lots of @s in them, because @ is a fundamentally ugly symbol. I also have some reservations about @, but for a different reason, namely that @ and @: imply <u>sequence</u>, and pieces of code strung together with @s look much like the kind of line-by-line code observed in more mundane computer languages.

Two examples illustrate both of these points. The first is `=@i.@#` which gives an identity matrix of the same shape as a square matrix. This phrase has a feeling of symmetry on account of the two @s. It is analogous to a word such as 'fever' in which consonants and vowels alternate, only now read verbs and conjunctions. As for its meaning, the @s say that there are three things which have to be done in sequence, first a *tally*, then an *integers* of that *tally*, and finally a *self-classify of* the result of *integers.* If I were to assign and name this verb for repeat use I would probably call it something like MIM or MIdMat, standing for Matching Identity Matrix. In this case I am not sure which of the character strings MIdMat or `=@i.@#` conveys more on later recall.

Now consider the somewhat similar phrase `#i.@#` :

```
      (#i.@#)0 1 0 1 1
1 3 4
```

which converts a bit string into the indices of the 1s. Again the phrase has an overall shape and feel, in this case that of a word like 'trot'. A name for assignment purposes might be something BtoInd which has a much lower mnemonic value than MIdMat, and in this case I am confident that my preference for recall purposes is `#i.@#` .

Another recent exchange on the forum offered two approaches to the problem of obtaining a list of occurrence numbers. The problem is that of taking a sequence such as

```
   m=.'mississippi'
```

and obtaining the list 0 0 0 1 1 2 3 2 0 1 3 in which each item of `m` is replaced by its occurrence number. Chris Burke and Cliff Reiter offered two different stylistic approaches. Here is Chris's:

```
   ocb=.[:((]-{)/:@/:)i.~
```

and here is Cliff's:

```
   ocr=.;@(<@i.@#/.~)/:[:/:~.i.]
```

If I were completely fluent in J, I would be able to understand and compare the two on sight without recourse to English to define intermediate verbs. Although there is a temptation to break these down into smaller verbs, I ask you, the reader, to make the experiment of pretending at this point that J is the only computer language you 'speak', and to make sense of these expressions on that basis alone.

Looking at `ocb` first, I have to say that I am not a fan of *cap* (`[:`) which I usually like to replace with an equivalent `@`, yielding in this case

```
   ocb1=.((]-{)(/:@/:))@ i.~
```

The case made for *cap* when it was first introduced into J was that it enabled trains of indefinite length to be constructed without parentheses. In reading J phrases I find that 'breathers' in the form of parentheses and @s are positively welcome, and the *cap* argument seems a bit like the case for removing gaps between movements of symphonies on the grounds that it gets you to the end sooner.

An obvious first step in solving the occurrence number problem is to translate the list into an 'order of first appearance' integer list, that is every 'i' in "mississippi" maps into the index of the leftmost 'i' which is 1, and similarly for the other characters. This is just what `i.~m` does:

```
   ]t=.i.~m
0 1 2 2 1 2 2 1 8 8 1
```

`t{m` is just `m`, since it makes no difference whether a selecting index is that of the first or any later occurrence. I can see therefore the role of

the rightmost verb in `ocb`, and also note that, without loss of generality, the rest of this discussion can be restricted to lists of this sort – this is the force of the rightmost @ in this case, and is incidentally a general useful property of @. What remains is clearly a hook, whose right verb is easily recognised as the upward ranking verb `/:@/:`. A significant problem in discussing the *grade* verbs is the promiscuity of the word 'rank', which means one thing in the context of J and another in the context of arithmetic generally. I think it best to use the word 'ranking' to describe the latter sense and follow J practice of counting from index origin 0, so that the score of the best golfer has upward ranking 0, that of the fifth best golfer upward ranking 4 and so on. The upward ranking of a list is the list of the upward ranks of its items in their original order. The left verb of the hook

```
   (]-{) (/:@/:)
```

in `ocb` is a fork in which the effect of *right* is to make the right argument `t` the left argument of *minus*, the right argument of which is `t{gt` where `gt` is the upward ranking list of `t`.
Now pause awhile and consider what happens when a list of indexes of first appearances (call this an `ifo` list for short, and define `ifo =.i.~`) selects *from* its own upward ranking list. Ranking in J is a progressive operation in the sense that if a list contains more than one 0, the second 0 from the left is ranked 1, the third is ranked 2 and so on. However, in selection using an `ifo` list, duplicate items in `t` repeatedly select the upward ranking of first occurrence making the process non-progressive. This is easier to observe by tracing than to describe in words

```
   t
0 1 2 2 1 2 2 1 8 8 1
   /:/:t                 NB. upward ranking of t
0 1 5 6 2 7 8 3 9 10 4
   t{/:/:t               NB. t indexing its own ranking
0 1 5 5 1 5 5 1 9 9 1
```

Subtracting the last two lists in the above sequence gives the occurrence number list.

```
   ocb m
0 0 0 1 1 2 3 2 0 1 3
```

All of this can be summarised by saying occurrence number is 'upward ranking' minus 'index of first appearance'. I could express this in mathematical symbols, but I doubt whether this would achieve anything like the clarity of the previous sentence. It is precisely this

derived consequence of the semantics of *grade-up* and *index* which Chris has exploited delightfully in `ocb`.

The fork in `ocb1` raises a stylistic consideration. The purpose of] is to achieve the argument reversal needed to obtain

```
(/:/:t) - t{(/:/:t)
```

I have reservations about using [and] similar to those I expressed concerning @, namely that [and] are a thinly disguised way of referring directly to the data arguments, and thereby causing reversion to traditional programming language style rather than that of tacit programming.

If `up` is defined as

```
up=./:@/:
```

then the fork in `ocb1` can be written as

```
up -({up)
```

and `ocb1` can be written

```
ocb2=.(up-({up))@ifo
```

Of course I am now taking refuge in words, or at least verb-names as pseudo-words, no doubt thereby laying bare my shortcomings in J fluency. As far as recall is concerned, while `ifo` is quite memorable (although no shorter than `i.~`), the name `up` is uselessly general. I could of course follow the practice of C++ programmers and similar infidels, and call it something like `upward_ranking` but this would be self-defeating, since `/:@/:` is much shorter and infinitely more expressive.

As a spin-off, the hook `{up` delivers the non-progressive upward ranking in which all equal values have the same ranking. For example

```
   ({up) 0 1 2 0 2 5 0
0 3 4 0 4 6 0
```

Next consider Cliff's verb:

```
ocr=.;@(<@i.@#/.~)/:[:/:~.i.]
```

The first thing to observe is the presence of the *key* adverb (`/.`) which is what I imagine many J users, myself included, would latch on to as the germ from which to develop a solution to this problem. Again pretend that J is the only available vehicle for communication. The presence of several `@`s indicates that Cliff has taken an essentially sequencing approach, particularly as *cap* can, as before, be exchanged for `@` to give

```
   ocr1=.;@(<@i.@#/.~) /: /:@( ~. i. ])
```

This reveals that the main structure is a fork with the leftmost `/:` as its central prong. As far as the right prong is concerned, two things happen in sequence. First the phrase within the parentheses is *index* by *nub*, for example

```
   (~.i.])m
0 1 2 2 1 2 2 1 3 3 1
```

which I recognise as another way of describing `ifo`, and incidentally conveniently deals with my distaste for] !

Then this is graded upwards

```
   /:ifo m
0 1 4 7 10 2 3 5 6 8 9
```

to give the positions of the 0s, followed by the positions of the 1s, followed by the positions of the 2s and so on, in other words it is the list of positions required to *grade up* dyadically the sequence of all occurrence numbers ordered by item.

key seems a natural way of obtaining this sequence. We have already seen how `i.@#` works

```
   (i.@#)m
0 1 2 3 4 5 6 7 8 9 10
```

Applying *key* using the items of `m` as keys gives

```
  m(i.@#/.)m
0 0 0 0
0 1 2 3
0 1 2 3
0 1 0 0
```

which is then *box*ed followed by *raze* - a standard device to remove unwanted fill characters. Putting all of this together and including a *reflex* to keep things monadic, the left prong may be summarised as

```
ocnobykey=.;@(<@i.@#/.~)
```

and the left prong by

```
posbykey=./:@ifo
```

leading to a further redefinition of `ocr` as

```
ocr2=. ocnobykey /: posbykey
```

An objection to this use of pseudo-English is that revisiting `sortbykey` and `posbykey` again poses as great a problem in remembering exactly what was meant by these rather vague verb names as that of interpreting directly the unambiguous J strings. The only significant arguments for decomposition of long verbs into shorter verbs is the 'breather' effect on the one hand and clarification of top-down structure on the other.

In summary, three separate dimensions of stylistic choice have evolved in the above discussion which have to be made in J programming. First there is the balance between English and J, (with 100% of the latter not ruled out), then secondly the different problem solving approaches of "derive a new property from old" versus "program using existing resources" (`ocb` style versus `ocr` style), and thirdly, choices between J equivalences such as *cap* versus *atop,* as well as avoidance techniques for [and]. Of course none of this takes account of the hugely important dimensions associated with implementation – often a longer piece of code will be appreciably more efficient than a more elegant alternative. At this point art and science come to a divide where things begin to get severely practical – most certainly relevant, but not the province of Jaesthetics!

Code Summary

The following verbs all deliver the indexes of successive occurrences of items in a list.

```
ocb=.[:(([-{)/:@/:)i.~
ocr=.;@(<@i.@#/.~)/:[:/:~.i.]
ocb1=.(([-{)(/:@/:))@ i.~
ocb2=.(up-({up))@ifo
   ifo=.i.~           NB. index of first appearance
   up=./:@/:          NB. upward ranking
ocr1=.;@(<@i.@#/.~) /: /:@( ~. i. ])
ocr2=. ocnobykey /: posbykey
   posbykey=./:@ifo
```

```
ocnobykey=.;@(<@i.@#/.~)
```

18. The problem with J is…

Principal Topics :] (*right*)   /. (*infix*)   \. (*suffix*)   /: (*grade up*)   ` (*gerund*) list constructor, statement separator, zigzag matrix.

… identifying instantly the various parts of speech. Even in the second or so it took you to read this sentence you subconsciously identified its form and shape because you recognised 'problem', 'part', 'speech' as nouns, 'is' as a verb, 'instantly' as an adverb and so on. In linguistic terms you sorted out the syntax with scarcely a thought, deferring for the moment whether or not you were going to bother working out the semantics.

By contrast, in J the parts of speech to which the various symbols belong are not immediately obvious. Take for example a discussion on the J programming forum concerning how to rearrange the square matrix `i.n n` in zig-zag format, so

```
   zigzag=.($ /:@:;@:(|.&.>`]/.)@:(</.)@:i.)@:(2&#)
   zigzag 4
0  1  5  6
2  4  7 12
3  8 11 13
9 10 14 15
```

The above one-line solution submitted by Henry Rich states the solution admirably, but its structure and meaning is clear only to the super-expert.

J is best understood in terms of processing *lists*, which is what goes on behind the scenes anyway. Verbs like *transpose* which operate at the surface level on objects of higher dimensions than lists are more accurately conceived as list constructors, in other words the verb *transpose* is not so much 'switch matrix rows and columns' as 'construct a new list of lists from a given list of equal length lists'. There is no problem in extending this idea to dyadic *transpose*, that is to lists of lists of lists … *ad infinitum.*

Adverbs such as / and /. make 'straightforward' verbs like < and +/ into list constructors. For example

```
   </.i.4 4
┌─┬───┬─────┬────────┬─────────┬─────┬──┐
│0│1 4│2 5 8│3 6 9 12│7 10 13│11 14│15│
└─┴───┴─────┴────────┴─────────┴─────┴──┘
```

constructs a 7-list systematically from the elements of four 4-lists, sometimes with dimensionality (rank) bein reduced in the process

```
   +//.i.4 4          NB. one 7-list from four 4-
lists
0 5 15 30 30 25 15
   (<0 1)|:i.4 4      NB. one 4-list from four 4-
lists
0 5 10 15
```

An interesting verb which can bring about list construction is `]` (*right*) which it is easy to dismiss trivially as a 'do-nothing' sort of verb, or at least 'deliver the right argument and do nothing with it'.

```
   2]i.3 3
0 1 2
3 4 5
6 7 8
```

However, if qualified with the *prefix* adverb

```
  ]\i.3
0 0 0
0 1 0
0 1 2
```

it gives successive sublists (with fill) constructed either forwards from a simple numeric, or backwards :

```
   (]\.)i.3
0 1 2
1 2 0
2 0 0
```

or as boxed lists suppressing the fill items :

```
   (<@:]\)i.3
┌─┬───┬─────┐
│0│0 1│0 1 2│
└─┴───┴─────┘
```

The same effects could be achieved by replacing] with > or + or many other verbs.

Ken Iverson used J to suggest that the numeracy functions of the brain are more closely allied to the literacy ones than is popularly believed, the first step being to overcome the syntax barrier. Part of the problem is that whereas statement separators tend to stand out in other languages, @: meaning 'after' acts in J like a statement separator as in `this@:that@:thenext`, but is often harder to pick out in a blitz of non=alphabetic characters. In the light of this look again at

```
   zigzag=.($ /:@:;@:(|.&.>`]/.)@:(</.)@:i.)@:(2&#)
```

in which it is difficult at a glance to pick out the all-important @: symbols. As a first step in understanding construct lists of the leading diagonals of `i.n n` and consolidate the procedure as

```
   diagilist=.monad : '</.i.y,y'
   diagilist 4
┌─┬───┬─────┬────────┬────────┬─────┬──┐
│0│1 4│2 5 8│3 6 9 12│7 10 13│11 14│15│
└─┴───┴─────┴────────┴────────┴─────┴──┘
```

Another important indicator which is easy not to spot is the gerund marker, that is the dash in the middle. The adverb `/.` activates the gerund as in

```
   (+`-/.)4 5 6 7
 4
_5
 6
_7
```

and so a fuller parenthesising is `((|.&.>)`(])/.)`, that is the two parts of the gerund are `|.&.` and `]` and *not* `|.&.` and `]/.` , so define

```
   reversealt=.|.each`]
   zz=.(reversealt/.)@:diagilist
   ,zz 4
┌─┬───┬─────┬────────┬────────┬─────┬──┐
│0│1 4│8 5 2│3 6 9 12│13 10 7│11 14│15│
└─┴───┴─────┴────────┴────────┴─────┴──┘
```

or, removing the boxes

```
   ;zz 4
0 1 4 8 5 2 3 6 9 12 13 10 7 11 14 15
```

Alternate diagonals have been reversed, and the problem now is to reverse the diagonalisation. Here is where an ingenious and insightful verb comes in. For any permutation `p` of `i.n` the following applies :

```
   p=.16?16
   (/:p){p
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

that is *grade-up* restores `i.n` from any of its permutations. Now `;zz 4` as shown above is a permutation of `i.16` whose *grade-up*, just like any other permutation, puts the numbers of `i.16` back into natural order, which is exactly what is required in order to display the zigzag matrix in its conventional 4 by 4 format. Thus define a verb

```
   gradeupraze=./:@;
   zz=.gradeupraze@:(reversealt/.)@:diagilist
   zz 4
0 1 5 6 2 4 7 12 3 8 11 13 9 10 14 15
```

and finally reshape this as `n` by `n` :

```
   zigzag=.(2&#)$gradeupraze@:(reversealt/.)@:diag-
ilist
   zigzag 4
0  1  5  6
2  4  7 12
3  8 11 13
9 10 14 15
```

The sophistication of the J involved cannot be disguised, but using pseudo-English to break up the primitive symbols greatly helps comprehensibility. It also helps distinguish the more subtle thinking about permutations and *grade-up* from the otherwise relatively mundane data structuring operations.

Solving programming exercises of this sort often lead to further generalisations. For example consider how restructuring of the 4 by 4 matrix might be achieved :

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

`udilist` standing for 'up and down integer-list' is a list constructor analogous to `diagilist`.

```
   udilist=.monad : '<"1 |:i.y,y'
   udilist 4
┌────────┬────────┬──────────┬──────────┐
│0 4 8 12│1 5 9 13│2 6 10 14│3 7 11 15│
└────────┴────────┴──────────┴──────────┘
```

Now instead of the first, third, and so on list being reversed it is now the second, fourth …, so define the gerund

```
   revalt=.]`(|.each)
```

and finally retread the ground of `zigzag` :

```
   updown=.(2&#)$gradeupraze@:(revalt/.)@udilist
   updown 4
0 7  8 15
1 6  9 14
2 5 10 13
3 4 11 12
```

If the start point is required to be in the south-west corner, replace `revalt` with `reversealt`.

Code Summary

```
   diagilist=.monad : '</.i.y,y'
   reversealt=.|.each`]
   graduprraze=./:@;
   zigzag=.(2&#)$graduprraze@:(reversealt/.)@:diag-
ilist
   udilist=.monad : '<"1 |:i.y,y'
   revalt=.]`(|.each)
   updown=.(2&#)$graduprraze@:(revalt/.)@udilist
```

19. Symphony in J minor, op.31

Principal Topics : |: (*transpose*), \. (*suffix*) /. (*oblique*), minors (of a matrix)

Composer's programme note : In response to a question on the forum "is J beautiful?", Roger Hui answered as follows :

```
minors =. 1&(|:\.)"2^:2
```

If you know mathematics, you already knew what minors are. If you know J but didn't know mathematics, you will also now know what minors are, even if you didn't before! For the benefit of listeners who fall between these two categories, any matrix with m rows and n columns can be transformed into a set of mn matrices each of size (m-1) by (n-1) by omitting the row and column of every element in the original matrix in turn. That much is a statement in English of the phrase above. The phrase itself forms the symphony's motif, whose meaning unfolds movement by movement as the work progresses.

First movement : **Introduction and andante syntactico**. The end of the motif, has a repeat sign `^:2` indicating that the theme is to be played twice. The theme itself, to be found within the parentheses, is `|:\.` , that is a melody `|:` played with orchestration `\.` . Next the opening bar of the motif shows that there are two staves, in which `&` braces the melody to a pedal bass 1. The coda `"2` adds dynamics to the main theme, indicating the level at which it is to be played. The structure of the motif should now be apparent, and the remaining movements bring into increasingly sharp focus how the four repeated musical elements `|:` `\.` `&` `"` combine to make a single coherent form. As these component ideas emerge and crystallise, the work itself surges forward with a compelling, indeed inevitable momentum.

Second movement : **Pastorale** The movement opens with the folk melody `|:` which with its easy turn of phrase is often to be heard being hummed and whistled in its native matrix environment. `\.` , called *suffix*, is perhaps not quite so well known. As the name suggests, it is associated with Suffolk, the most easterly of England's counties, which slopes gently downwards from west to east until it disappears below the surface of the ever encroaching and unforgiving sea which is portrayed by the dot. The following illustration shows how it works, as first one part and then another withdraws from the melodic strand, in the style of the Farewell Symphony:

```
   ]m=.>'life';'ebbs';'away'NB. 3 strands each played to
life                        NB. completion before the other be-
gins
ebbs
away

   ,m
lifeebbsaway               NB. .. then played altogether as one

   ,\.m
lifeebbsaway               NB. .. now first three,
ebbsaway                   NB. .. then two,
away                       NB. .. then one
```

Gentle *suffix* is quite unlike the rocky coastal cliff scenery of Cornwall on the other side of the country where her cousin *oblique* evokes the music of the waves as they surge and then fall away in natures's vertical polyphony :

```
   ,/. m             NB. flow at first …….. 
l                    NB. the three melodies enter
ie                   NB. one after the other
fba                  NB. in downward progression
ebw
sa                   NB. .. then ebb
y
```

The music of the integers, if not the spheres, can be heard in the same fashion :

```
   +/. 1+i.4 4
 1  0  0  0   NB. strata of sequences increasing by 3
 2  5  0  0   NB. padded with fill items
 3  6  9  0   NB. rising in length ..
 4  7 10 13
 8 11 14  0   NB. .. and then falling
12 15  0  0
16  0  0  0
```

Third movement : **Presto semantico**   Now the number 1 is heard knocking at the door like Fate in Beethoven's fifth. No longer is `\.` a gentle ebb and flow, now decline (*suffix*) has become a relentless forward march to the rhythm of *out,fix,out,fix,out,fix,out..*, as first one and then another of the underpinning strands is left *out* :

```
   1 ,\. m
ebbsaway             NB. strands are now played in all
lifeaway             NB. possible combinations, leaving out
lifeebbs             NB. one at a time
```

Leave them out two at time, and there are just single strands left, not three as might be expected, but only two, since those left out must be in succession :

```
   2 ,\. m
away                NB. leave out strands 1 and 2
life                NB. leave out strands 2 and 3
```

(Note : the bass line may be played with either positive or negative vibrato, that is it makes no difference if 1 is replaced by _1 or 2 by _2.)

A percussion line can be added to each strand to make `,` into a two-stave melody :

```
   (,&'!')\.m
life
ebbs
away
!!!!

ebbs
away
!!!!


away
!!!!
```

This illustration can be played more elegantly on the box piano `<"2`

```
   <"2 (,&'!')\.m
```

life ebbs away !!!!	ebbs away !!!!	away !!!!

Adding a second stave to *oblique* transforms polyphony into modulation through a variety of *keys,* so in the next example theme A is in one key and themes B and C in another :

```
   1 2 2 ,/.m
life
ebbsaway
```

Fourth movement : **Finale grandioso** In the final movement, the influence of Eastern music becomes apparent, since the performer is free to choose an initial foundational thread to start off each individual performance. Following this the various musical elements of the preceding movements are brought together to form a grand climax of re-

sounding symbols. Rank 2 is the natural medium for the foundation thread, which might, for example, be

```
   i.3 2
0 1
2 3
4 5
```

but it could equally well be a matrix of any size, rank or type (character or numeric). For a rank 2 foundation thread the explicit dynamic `<"2` is not strictly necessary for the first statement of the melody in which the full force of the orchestra is unleashed, slicing and transposing the rows in a crescendo which finishes at rank 3 level

```
   (1&(|:\.)) i.3 2
2 4
3 5

0 4
1 5

0 2
1 3
```

In the repeat section, each of the inner level (<"2) matrices is sliced and transposed, the latter ensuring that the original tonality of rows and columns is reinstated :

```
   t=. (1&(|:\.)) i.3 2
   <"2 (1&(|:\.))"2 t
┌───┐
│3 2│
│5 4│
├───┤
│1 0│
│5 4│
├───┤
│1 0│
│3 2│
└───┘
```

In the most popular performances, the foundation thread is rank 2, numeric and square :

```
   <"2 minors i.3 3
┌───┬───┬───┐
│4 5│3 5│3 4│
│7 8│6 8│6 7│
├───┼───┼───┤
│1 2│0 2│0 1│
│7 8│6 8│6 7│
├───┼───┼───┤
```

```
│1 2│0 2│0 1│
│4 5│3 5│3 4│
└───┴───┴───┘
```

This has led to repeated demands for the work to be performed globally wherever a popular audience for linear algebra is to be found. However, it has also been played with acclaim to audiences of a more literary inclination:

```
   <"2 minors 2 4$'lasttime'
┌───┬───┬───┬───┐
│ime│tme│tie│tim│
├───┼───┼───┼───┤
│ast│lst│lat│las│
└───┴───┴───┴───┘
```

New arrangements for different combinations of instruments and platforms are constantly in demand.

Encore ?? …

Code Summary

```
   minors =. 1&(|:\.)"2^:2
```

20. How to Do Things with Words

Principal Topics : < (*box*) " (*rank* conjunction) noun rank, verb rank, cell, matrix inverse, tables

David Ness recently drew my attention to a fascinating little book called "How to Do Things with Words". This is an edited version of lectures given in 1955 at Harvard by an Oxford philosopher called John Austin, but the title is equally appropriate as a succinct answer to an enquirer seeking to discover what J is all about. The gist of the book is that there are some things in life which can only be done by making utterances, "I name this ship ..", "I baptise this baby …" and so on. The pursuit of analogies with the grammars of natural languages has always been one of the driving forces of J, and it is not accidental that what in other languages are called 'functions' are called 'verbs' in J, while 'constants' and 'variables' are called 'nouns'.

A J interpreter provides answers to questions about what happens when words are put together in combinations which its creator could not possibly have ever conceived. In its rigorous judgements of J competence, a J interpreter contrasts with the manner in which 'perfect' English is not an absolute standard but an inherently unattainable ideal. When my German born travel agent answers my question "Is that Brigette?" with the response "This is she", by every formal rule of English grammar this has to be judged totally correct, but not many native English speakers would instinctively consider it so. On the other hand a J nterpreter answers all possible questions of grammar and syntax (including millions which have not yet been asked) with total certainty and clarity. It is in effect a super-intelligence to which even the most accomplished J 'speakers' must submit an implicit validity test with every J phrase which they utter or compose. Such speakers rarely react to an error report or unexpected result by presuming a bug is present, but rather accept that their instinctive and developing notions of the language do not yet quite measure up to the performance ideals of the unforgiving language examiner!

Much was made of APL as a notation for thought, and of the idea that, to quote A.N.Whitehead, "by relieving the brain of unnecessary work a good notation sets its users free to concentrate on more advanced problems thereby increasing the powers of the race". It might be supposed that this notion carried over into J, but this is not entirely correct. APL is first and foremost a shorthand, J only incidentally so. At an early age we develop fluency in the language we hear all around

us, and we rapidly achieve speech as opposed to language skills. For example the child who says "throw Daddy" has acquired a language skill in combining words in a fashion which mimics the rhythm of the social interactions he hears in the adult speech surrounding him. The speech skill by which this utterance develops into "throw it to Daddy" comes at a later stage. Combining words in J and submitting them to an interpreter is like submitting "throw Daddy" to the full glare of perfect adult intelligence, leading to a possible unwanted response where a physically strong and highly spirited Mummy happens to be present!

This analogy must resonate clearly with anyone who has crafted and executed a J verb with a clear expectation of what it will perform. Surprise is frequently conveyed not by an error report, but by the delivery of some result, which, if carefully observed, can advance a general background goal of bringing the user's fluency in J closer to the perfection which only the super-intelligence can attain. In other words, reflective consideration of faulty utterances is the prime route towards the attainment of language mastery. It is all too easy to set such explorations aside without fulfilling good intentions to return and learn from the experience. Thus many J speakers never progress much beyond the "throw Daddy" stage, or at best graduate only as far as the language equivalent of infanthood!

I said above that J, at least in the perceptions of some of us, is about doing things with words, and I return now to that theme in the context of the two types of rank which seem to be one of the principal stumbling blocks for initial language learners. The two types of rank are *noun rank* and *verb rank*. The former is a relatively simple idea to grasp, namely the result of applying #$ to an object to give the tally of its shape, or, in a single word, its **dimensionality**. Shape itself has geometrical connotations, so that a scalar has rank 0, akin to a point in space, a list has rank 1 like a line, a matrix has rank 2 analogous to points in a plane, and so on.

Verb rank is somewhat more subtle, but has more than a passing association with the grammatical idea of 'number', that is singular or plural. It is also helpful to consider rank alongside the concepts of *box* and **cell**, both of which are means of 'singularising' things, the former in a much stronger way by disguising the nature of the contents from external agents such as verbs. The Object Oriented concept of **encapsulation** is highly relevant here, so that a box is a scalar whereas a cell is not, the key difference being the property of penetrability.

The majority of J verbs have rank of either zero or infinity (see E #5 “Conjugacy and Rank” and E #6 “Punctuation and Rank”). It is only for verbs which have special semantic requirements such as *shape* and *matrix inverse* that any other rank is associated with the verb. Basically, the ‘safe’ assumption about the rank of an unknown verb is that it is infinite, which is why all but the most trivial user-defined verbs, are assigned infinite rank. (There are occasions where, with analysis, a ‘true’ rank could in principle be worked out, but such computing effort would not only overload the interpreter for little practical gain, but would also cost greatly in performance).

Simple addition at different rank levels involves thinking about cells with possibly complex list structures. Addition without explicit rank can penetrate list structures by operating at rank 0 :

```
   x=.i.2 3          NB. a list of two 3-lists
   y=:>:x            NB. another list of two 3-lists
   x;y;x+y
┌─────┬─────┬──────┐
│0 1 2│1 2 3│1 3  5│
│3 4 5│4 5 6│7 9 11│
└─────┴─────┴──────┘
```

However, adding objects with different structures may result in shape incompatibility, for example in the following case

```
   ]z=.10*i.2 2 3   NB. two 2-lists of 3-lists
 0  10  20
30  40  50

60  70  80
90 100 110
   x+ z             NB. two 2-lists plus two 3-lists
|length error
|   x    +z
```

you cannot meaningfully add 3-lists to 2-lists! However, addition at rank 1 <u>is</u> valid

```
   x +"1 z    NB. add two lists to two lists of lists
  0  11  22   NB. this is 0 1 2 + plane 0 of z ..
 30  41  52   NB. ..with list 0 1 2 replicated.

 63  74  85   NB. and this is 3 4 5 + plane 1 of z ..
 93 104 115   NB. ..with list 3 4 5 replicated.
```

because it is an addition of two compatible 2-lists. Each of `x`’s two lists is a 3-list, whereas for `z`, both its lists are 2-lists, each item of

which is a 3-list, and it is at the level of the 3-lists that the actual addition takes place. Addition at rank 2 is also valid:

```
   x+"2 z      NB. add one list to two lists
 0  11  22     NB. this is x + plane 0 of z ..
33  44  55

60  71  82     NB. .. and this is x + plane 1 of z
93 104 115
```

In this case, a single 2-cell is added to a list of 2-cells, and so in this case the left argument `x` is replicated in its entirety.

+ is one of a subset of verbs, mainly arithmetic, which 'penetrate' indefinite layers of structure to operate at atomic (that is numeric or character) level, as in the case of `x+y` above. In APL these are called **scalar verbs**, and their counterparts in J are said to possess rank 0. Where scalar verbs operate on conformable arguments as with `x+y`, increasing verb rank has no effect on the result. This is because, for example, `x+"1 y` is the addition of two pairs of 1-cells and `x+"2 y` is the addition of a single pair of 2-cells, and in each case penetration takes place down to atomic level. In J terms

```
   (x+"1 y)-:x+y    NB. rank 1 add matches rank 0 add"
1
   (x+"2 y)-:x+y    NB. rank 2 add matches rank 0 add
1
```

Returning to *box*, whereas rank 0 verbs penetrate list structures, they do <u>not</u> penetrate boxed structures:

```
   (<x)+<z
|domain error
|   (<x)    +<z
```

In general, the permitted operations on boxed scalars are restricted to shaping and joining, since encapsulation means that the nature of their contents is not available to verbs. To repeat the warning given earlier, do not confuse cells and boxed scalars!

Applying adverbs to verbs with rank continues the same rigorous logic. For example applying *table* :

```
   $x+/z      NB. All possible atomic pairs are added
2 3 2 2 3
   $x+"1/z    NB. A 2 by 2 table (items=3-lists)
2 2 2 3
   $x+"2/z    NB. 1x2 table(items=2-lists of 3-lists)
2 2 3
```

In the first case the shape of the result is the catenation of shapes. If this were not so it would be impossible to accommodate all possible number pairs. However, when two lists are added, addition takes place only between matching items, which means that in forming the second table the two `3`s at the lowest shape level in each argument are merged. Similarly, in the last case it is the two `2 3`s which are merged by addition to give a 2-list, each item of which has shape `2 3`. If you want to confirm your understanding of all this, try to predict the results of `x+/"1 z` and `x+/"2 z`. It helps to appreciate that 'table addition' is not a scalar verb, that is, it has infinite rank. You can use verb rank to specify the level at which tables are to be formed so that, for example, `x+/"0 y` is identical to `x+y`. However, no penetration takes place below the level specified by an explicit rank. Thus :

```
   $x+/"1 z  NB. A 2 by 2 table of 3 by 3 tables
2 2 3 3
   $x+/"2 z  NB. A 1 by 2 table of 2 by 3 tables
2 2 3 2 3
```

In the first case above the verb `+/`operates item by item on two pairs of lists to produce a 2 by 2 structure of 3 by 3 matrices.

Now return to natural language. Verbs like 'quit' and 'bid' make no number distinction – "he quit, they quit" and so on – as opposed to verbs like 'spell' – "he spells", "they spell" - which do. The concept of **plurality** with respect to verbs is a little more subtle than it might at first appear. For example if I say (talking about a group of people) "they sang", it could be taken to mean that each individual in the group sang, that is, the singing was performed at rank 0. On the other hand, the statement "they sang" could equally be applied to, say, a crowd at a football match where it was almost certainly <u>not</u> the case that every single individual sang. On the other hand "they multiplied" (in the reproductive sense!) could not have been done by individuals, and so we have here a verb which whose rank is certainly greater than zero! Similarly, if an utterance like "they quit" meant that <u>all</u> of them did as a body, then we have an instance of infinite rank. At best English verbs allow the distinction between one and many, that is between singular and plural. Explicit verb rank in J allows all possible gradations in between, 0, 1, 2 and so on up to infinity, and further, it allows ranks to be specified separately for subject and objects, much as if English were to provide separate inflections for "he hitss her, "he hits them", "they hit them" and "they hitt him". With a commutative verb like + there is no difference in result be-

tween `x+"0 1 y` and `x+"1 0 y`. However, the distinction is immediately apparent with a non-commutative verb :

```
    x-"0 1 y        NB. atoms of x minus rows of y
_1 _2 _3                  NB. 0 minus 1 2 3
 0 _1 _2                  NB. 1 -minus 1 2 3
 1  0 _1                  NB. 2 -minus 1 2 3

_1 _2 _3                  NB. 3 -minus 4 5 6
 0 _1 _2                  NB. 4 -minus 4 5 6
 1  0 _1                  NB. 5 -minus 4 5 6
   x-"1 0 y         NB. rows of x minus atoms of y
_1  0  1                  NB. 0 1 2 minus 1
_2 _1  0                  NB. 0 1 2 minus 2
_3 _2 _1                  NB. 0 1 2 minus 3

_1  0  1                  NB. 3 4 5 minus 4
_2 _1  0                  NB. 3 4 5 minus 5
_3 _2 _1                  NB. 3 4 5 minus 6
```

In English we tolerate the absence of such inherent number precision, which makes explicitly ranked J verbs harder to grasp than English ones - but on the other hand what interesting and varied things one can do with them!

21. Are you thinking what I'm thinking?

Principal Topics : = (*self classify*), Sapir-Whorf Hypothesis

"APL as a Tool of Thought" was the title of a long-running series of seminars held in New York, which, by a tiny change could be made into the claim "APL is a Tool of Thought". Generalise this into the statement "Some language is necessary as a Tool of Thought" and the resulting statement is what is known to linguists as the Sapir-Whorf Hypothesis (SWH). In short, you cannot think unless you have a language.

Whether this is true or not is open to question. Is it possible for an individual to think thoughts which are totally beyond his or her powers of articulation? Could there have been a caveman possessing all the scientific and cosmological vision of an Einstein or a Feynman, but who, having no language, was never able to communicate the fact? Proponents of SWH say no.

"APL is a tool of thought" can be paraphrased "APL is a labour-saving device for thinkers" in the same way that a Dyson is a labour-saving device for cleaners. Given that serious thinking is a hard, brain-taxing activity, "APL is a labour-saving device for thinkers" is probably not in dispute, and indeed the same could be said of any computer language. So what about the proposition "J is a better labour-saving device for thinkers", or more controversially still "APL (and J) enable people to think thoughts which were previously unattainable to them".

Computer languages in general differ from natural languages in the privacy of dialogue. Spoken human language competence is tested by a continual stream of reactions from receivers; if the latter respond in a puzzled or incomprehensible fashion, the speaker can often repeatedly retry, thereby learning by failure and feedback, one of the richest means of increasing language competence. By contrast, the response to failure with computer languages is normally an error message, impassive, and emotionally neutral.

One of the ways in which J is sharply differentiated from other computer languages, even APL, is in the relative infrequency of show-stopping error messages. Whereas erroneous input in other languages is likely to cause suspension of execution, as often as not J continues, but delivers an output which is often quite different from that expected. Put in another way, whereas a faulty expression in other

languages leads to non-understanding, in J it is just as likely to lead to misunderstanding. A possible response to an unexpected J output might be to shift a parenthesis a place or two, or to switch the odd conjunction, perhaps with the dialogue and resubmissions becoming more rapid as wishful optimism begins to overtake well-considered thought.

Any faulty input which did not deliver an error message is nevertheless the expression of a valid thought, failure to explore which is to forego a valuable opportunity for learning. Thus as well as working towards a correct expression, there is at least as much learning value in ascertaining what thought or intention would have resulted in the faulty input, which is after all the thought which would have been communicated to a more linguistically competent J receiver.

To be specific, consider a J dialogue in which I attempted to find all the combinations of `x` integers from `i.y`. My strategy, at first vaguely conceived in natural language, was first to obtain bit string patterns of all the natural numbers from 0 to 2^y-1, and then select those which contain exactly x bits, with the final step being to convert these bit strings into indices.

The first step is provided by a single J symbol. For the sake of definiteness, think in terms of combinations of 2 items out of 3.

```
   ]t=.#:i.8         NB. 8 is 2^3
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

or more generally

```
   b=.#:@i.@(2&^)    NB. 0,.._1+2^y as binary numbers
   t=.b 3            NB. same t as before
```

There is no problem in calculating the number of bits in each of the 2^y lists

```
    (+/"1)t          NB. sum all the lists in t
0 1 1 2 1 2 2 3
```

My next thought was to select those lists whose bit-sum equals 2 :

```
   ((2&=)@+/"1)t    NB. mark lists whose sum is 2
0 0 0 0 0 0 0 1
```

Something wrong! – what I should have thought was

```
   (2&=@(+/"1))t    NB. right this time?
0 0 0 1 0 1 1 0
```

Following a well-established route for selection using `#~` , the above thinking leads to

```
   (#~2&=@(+/"1))t  NB. (#~f)y means f(y) # y
0 1 1
1 0 1
1 1 0
```

From there the route to the summit is clear, but before proceeding, what about the wrong thought? Following the advice given earlier, I should, as a conscientious J learner, ask what it was which I conveyed to the hypothetical perfect J speaker/reader when I said `((2&=)@+/"1)t` ? In the first place such a reader would have mentally inserted a right parenthesis following the first verb after @ so that what was *inserted* in the J sense was the verb `v=.(2&=)@+` . What does this convey? Any verb to the left of @ must be monadic; the fact that the adverb *insert* follows `v` means that the + in `v` must be dyadic, and so `v` is 'compare-with-2 addition'. Next since the *insert* verb is defined as operating on lists (`"1`), the insertions are between individual items. Given that each list is a bit-list, the only possible results of adding neighbours are 0, 1 and 2. The last occurs only when both inputs are 1, or equivalently compare with 2 is true, and so in the present context `v` is equivalent to `*` with bits, that is `*.` (*logical and*). Here is confirmation :

```
   v/"1 t
0 0 0 0 0 0 0 1
```

As an aside (and asides can be very valuable in language learning) suppose now that a second thought was to change `v=.2&=@+` to `w=.2&(=@+).` Only monadic verbs may appear to the left of @, and so = must now be *self-classify*, which returns the value 1 for all scalars. The next question is what does `2&` mean? The definition of *bond* with a noun is

```
   x m&v y ↔ m&v^:x y
```

so `x 2&(=@+)y` means `2&(=@+)^:x(y)` in the present case. In words, the left argument is a repeat factor, so in the case of bits the verb `=@+` is repeated either 0 or 1 times. If it is 0 the result is simply the right argument, if it is 1, the result is 1 by virtue of *self-classify*. The 2 in the *bond* is irrelevant, and could have been any number, real or complex. Put all this together and dyadic `w` is equivalent to 'or' :

```
   0 w 1
1
```

Or is it? There is a snag, namely that the rank of the result of *self-classify* is a table comparing an object with its *nub*, and so has rank at least 2, so unlike `v/"1 t`, `w/"1 t` is not a list of simple binary scalars :

```
     (0 w 0),(0 w 1),(1 w 0),(1 w 1)
0
1
1
1
  $w/"1 t
8 1 1 1 1
```

The 8 above is easy to explain, but why four 1s when each item of `t` is a 3-list? The answer is that each insertion (two in the case of `t`) generates a 1 by 1 table, as is confirmed by

```
   $w/"1 b 4
16 1 1 1 1 1 1
```

To return to the correct route, or call it perhaps the clear-thinking path, the experience of the exploration of wrong avenues suggests separating out the list summation process

```
   s=.= +/"1        NB. 1 if x = list sums of y, else 0
   3 s i.2 3        NB. sum two lists, return 1 if eq to 3
1 0
```

The right argument in the context of combinations is `b y` and so the list which identifies the relevant binary lists is the hook `(s b)y` .

Hence

```
   d=.(s b) # b@]   NB. binary representation of x-combs
```

delivers the bit strings which corresponds to the required combinations :

```
   2 d 3
```

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

and a final tally using bits selects the relevant lists of indices

```
    2 c 3
1 2
0 2
0 1
```

Here is the final (and correct!) stream of thought which led to this approach to the problem :

```
   b=.#:@i.@(2&^)    NB. i.2^y as binary numbers
   s=.= +/"1         NB. 1 if x=list sums of y, else 0
   d=.(s b) # b@]    NB. binary representation of x-combs
   c=.d # i.@]       NB. convert binary to indices
```

But to return to the beginning, the primary object of this article was not the final algorithm itself, but rather reflection on the journey to get there, and on the value of following up wrong paths taken on the way. If you have read this article, you will almost assent to the propositions that APL and J are labour-saving devices for thinkers. But more subtly :

(a) are there people for whom both APL and J allow the expression of thoughts which they were previously able to think but not express?; and.
(b) are there people for whom both APL and J allow the thinking of thoughts which they were previously unable to think?

Code Summary

```
   c=.d # i.@]              NB. all x-combinations from i.y
     d=.(s b) # b@]                NB. binary representation of
x-combs
       b=.#:@i.@(2&^)       NB. i.2^y as binary numbers
       s=.= +/"1            NB. 1 if x=list sums of y, else 0
```

22. Index Lists, Grade Lists, and some simple joins

Principal Topics : { (*from*) {. (*take*) , (*append*) ,. (*stitch*) ,: (*laminate*) ; (*link*) /: (*grade up*) # (*tally, copy*) >: (*increment*) ": (*rank conjunction*) ": (*format*) >. (*larger of*) merging lists, row and column headings, row and column proportions

Every time a list is created, several other lists are automatically created whether the user realises it or not. The most obvious is the index-list `i.@#`. Next are the *grade up* and *grade down* lists which are permutations of the index-list which give the indices necessary to arrange the original list in either ascending or descending order. Go one step further and obtain the index-list of the result of appending two lists, and you have the basis for merging lists as the next section shows.

Merging lists with alternate items

1 2 3 4 merged with 5 6 is to become 1 5 2 6 3 4. This illustrates well the concept of fork.

```
    malt=.mselect { ,
```

that is start by joining the two lists as they stand (1 2 3 4 5 6), and then find the right selection list in order to put the items of the joined list in the required order. A moment's thought shows that the required selection list is going to be 0 0 1 1 2 2 …

Every list has an index list given by

```
   ilist=.i.@#
   ilist 'abcde'
0 1 2 3 4
```

Joining the index lists of the lists to be merged is a first step along the road

```
   ijoin=.,& ilist
   1 2 3 ijoin 4 5
0 1 2 0 1
```

The positions of the items in this list in ascending order are provided by *grade up*, so

```
   mselect=./:@ijoin
   1 2 3 mselect 4 5
0 3 1 4 2
```

and so finally

```
   1 2 3 4 malt 5 6
1 5 2 6 3 4
```

Other merge patterns are now easy to improvise, for example if two from the left are to be taken for every one from the right adjust two of the verbs

```
   ijoin21=.dyad : '(2#ilist x),(ilist y)'
   mselect21=./:@ijoin21
   malt21=.dyad : '(x mselect21 y){ (2#x),y'

   1 2 3 malt21 7 8
1 1 7 2 2 8 3 3
```

The remaining sections deal with joining lists to tables.

Row and Column Headings

Readers will be familiar with automatically adjusted headings generated in Excel and other spreadsheets. The algorithms for doing this can be conveniently analysed in J, and illustrate when it is appropriate to join and when to stitch when working with lists of lists. Start with some random data, a list or row headings and a list of column headings

```
   ]a=.?3 4$100
90 47 58 29
22 32 55  5
55 73 58 50
   rh=.'London';'Paris';'Dublin'
   ch=.'North';'South';'East';'West'
```

The items in `a` can be boxed in a neat turn of phrase :

```
      <&>a          NB. otherwise <each a
┌──┬──┬──┬──┐
│90│47│58│29│
├──┼──┼──┼──┤
│22│32│55│5 │
├──┼──┼──┼──┤
│55│73│58│50│
└──┴──┴──┴──┘
```

However to merge column headings with the data it is necessary to do some counting of character spaces as well as determining how many decimal places are to be displayed in the data. Assume a field

width of 6 and 1 decimal place, and column headers can be dealt with by

```
   ]b=.<every 6j1":each a
```

90.0	47.0	58.0	29.0
22.0	32.0	55.0	5.0
55.0	73.0	58.0	50.0

```
   Ch=._6{.each ch
   Ch,b                          NB. Column headers are merged
```

North	South	East	West
90.0	47.0	58.0	29.0
22.0	32.0	55.0	5.0
55.0	73.0	58.0	50.0

The row headers also require some enhancement to deal with the space in the North-west corner :

```
   Rh=.' ';rh
   Rh,.Ch,b                      NB. Row headers are stitched (,.)
```

	North	South	East	West
London	90.0	47.0	58.0	29.0
Paris	22.0	32.0	55.0	5.0
Dublin	55.0	73.0	58.0	50.0

Mission accomplished! A generalization of this process is to compute the maximum width required for a single column header (assuming that this will be no greater than the required header width), allowing for a space (>:), and use this to determine the file width in the data,

```
   eqlise=.monad : '->:>./>#every y'
   Ch=.(eqlise ch){.every ch
   b=.<&>((-eqlise ch)j.1)":each a
```

`Rh,.Ch,b` then gives the same result as above.

Some variations might be to eliminate the vertical grid lines within the data :

```
   Rh,.<"1(,>Ch),6j1":a
```

	North South East West
London	90.0 47.0 58.0 29.0
Paris	22.0 32.0 55.0 5.0
Dublin	55.0 73.0 58.0 50.0

Alternatively for a display without grid lines

```
     (>Rh),.(,>Ch),6j1":a
       North South  East  West
London  90.0  47.0  58.0  29.0
Paris   22.0  32.0  55.0   5.0
Dublin  55.0  73.0  58.0  50.0
```

In summary to decorate a basic table `a` with grids and row and column headers requires equalisation for the column headers and augmentation for the row headers leading to a description is `Rh,.Ch,b` .

Row and Column Totals

Appending row and column totals to a table :

```
   a=.2 3$1 7 5 5 2 0
   colsum=., +/
   rowsum=.,. +/"1
   colsum a
1 7 4
5 2 0
6 9 4
   rowsum a
1 7 4 12
5 2 0  7
   totsum a
1 7 4 12
5 2 0  7
6 9 4 19
```

To dress this with nice grid lines, proceed as follows :

```
   f=.5j1&":                NB. decide width and dec pl
   Rs=.f,.+/"1 a            NB. formatted rows
   Cs=.f+/a                 NB. formatted columns
   Ts=.f+/,a                NB. formatted total
   ((<f a),.<Rs),:<every Cs;Ts
┌───────────────┬─────┐
```

```
   1.0  7.0  5.0│ 13.0
   5.0  2.0  0.0│  7.0
 ───────────────┼─────
   6.0  9.0  5.0│ 20.0
```

all of which can be brought together in a user-defined verb :

```
   ttable=.dyad :0
f=.x&":                 NB. x is (width)j(dec places)
Rs=.f,.+/"1 y           NB. formatted rows
Cs=.f+/y                NB. formatted columns
Ts=.f+/,y               NB. formatted total
((<f y),.<Rs),:<every Cs;Ts
)
   5j1 ttable a
┌───────────────┬─────┐
│  1.0  7.0  5.0│ 13.0│
│  5.0  2.0  0.0│  7.0│
├───────────────┼─────┤
│  6.0  9.0  5.0│ 20.0│
└───────────────┴─────┘
```

If row and column proportions are wanted :

```
   colpro=.%"1 +/
   rowpro=.% +/"1
   colpro a
0.166667 0.777778 1
0.833333 0.222222 0
   rowpro a

0.0833333 0.583333 0.333333
 0.714286 0.285714        0
```

which might be clearer in rational (that is a**r**b) notation :

```
   fr=.'r1' ,~ ":   NB. convert to arb notation
   b=.".fr each a
   colpro b
1r6 7r9 1
5r6 2r9 0
   rowpro b
1r12 7r12 1r3
 5r7  2r7   0
```

Code Summary

```
   malt=.mselect { ,                                 NB. merge
alternate
      mselect=./:@ijoin
      ijoin=.,& ilist
      ilist=.i.@#
   malt21=.dyad : '(x mselect21 y){ (2#x),y'   NB. 2/1 merge
      mselect21=./:@ijoin21
```

```
   ijoin21=.dyad : '(2#ilist x),(ilist y)'
```

Adding row and column headings

Data:

```
a=.?3 4$100                    NB. table of random values
rh=.'London';'Paris';'Dublin'  NB. row headings
ch=.'North';'South';'East';'West'    NB. column headings
```

Adjustments to data

```
eqlise=.monad : '->:>./>#every y' NB. equalise column widths
Ch=.(eqlise ch){.each ch
b=.<&>((-eqlise ch)j.1)":each a      NB. generalised b
Rh=.' ';rh             NB. modified row headers
Rh,.Ch,b               NB. row & col headers added to a
```

Alternative forms of display

```
Rh,.<"1(,>Ch),6j1":a    NB. no vertical grid lines
(>Rh),.(,>Ch),6j1":a    NB. no grid lines
```

Adding row and column totals

```
   ttable=.dyad :0  NB.table with row & col totals
f=.x&":             NB. x is (width)j(dec places)
Rs=.f,.+/"1 y       NB. formatted rows
Cs=.f+/y            NB. formatted columns
Ts=.f+/,y           NB. formatted total
((<f y),.<Rs),:<every Cs;Ts
)
```

Row and column proportions

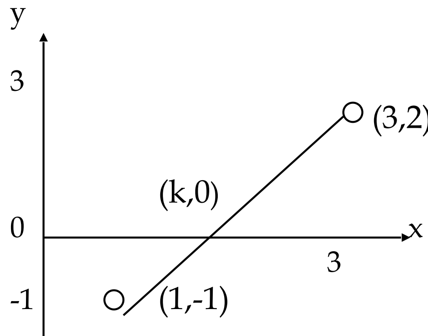
```
colpro=.%"1 +/   NB. column proportions
rowpro=.% +/"1   NB. row proportions
fr=.'r1' ,~ ":   NB. convert to arb notation
```

23. Some Numerical Problems Analysed in J

Principal Topics: | (*magnitude, residue*) |. (*reverse, rotate*) D. (*derivative*) }. (*head,take*)] (*right*) ^: (*power conjunction*) _ (*infinity*) { (*from*) {: (*tail*) }. (*behead*) -: (*halve*) #. (*base*) >: (*increment*) inner product, recursion

In this article it is assumed that the reader has at least a basic interest in or knowledge of elementary numerical analysis, and in particular of the three requirements of linear interpolation, root-finding and numerical integration.

Linear Interpolation
Suppose that a curve is known to go through the points (x_0, y_0) and (x_1, y_1). The point in which the straight line connecting these points cuts the x-axis is (k, 0) where
$k=(x_0y_1 - x_1y_0)/ (y_1 - y_0)$. This technique is the basis of the simplest for of root-finding. In the verb lint the left argument is x_0,x_1 ; right argument is y_0,y_1




```
   lint=. dyad : '((|.x)-/ .*y)%-/y'
   1 3 lint _1 2
1.66667
```

The two points involved do not necessarily have to be on the same side of the x axis for linear interpolation, but this is a necessary requirement for root finding.

Root-finding : Newton's Method
Three functions will be used as as test cases throughout the rest of this article. These are

```
   f1=.#.&1 _1 _2    NB. polynomial x²-x-1
   f2=.2&o.-(*^)     NB. cos x - xeˣ
   f3=.2&(-^&3)      NB. 2 - x³
```

Sometimes they will appear as an left argument, in other cases they will be pre-assigned to the variable `fn`. First Newton's iterative formula $x_{n+1} = x_n - \frac{y_n}{y'_n}$ for which the right argument is the initial guess.

`Newton` is an adverb qualifying the function whose root is to be found:

```
    Newton=.adverb : ']-x%x D.1'(^:_)("0)
   f1 Newton 1
2
   f2 Newton 1
0.517757
   f3 Newton 1
1.25992
```

Root-finding : Secant Method

This requires two initial guesses between which the root is to be found. The result is the two most recent approximations. Like `Newton`, it is an adverb qualifying a function. Since two points are needed to define a secant as opposed to one for a tangent, the secant adverb requires two start values, and at every step delivers a further two values to kick off the next approximation. This means that the infinity option with the *power* conjunction is not available, instead the number of iterations is given explicitly as a left argument.

```
   sec=.adverb : '}.@],] lint (x every@])' (^:[)
   7 f1 sec 0.5 1
1.99963 2
   4 f2 sec 0.5 1
0.517767 0.517757
   6 f3 sec 0.5 1
1.25991 1.25992
```

To clear up the untidiness of receiving two result values when only one is really wanted, here is a short iteration verb which stops when things get sufficiently close :

```
   sec1=.adverb : '}.@],] lint fn every@]'
   secant=.dyad : 0
t=.fn sec1 y
while.(x<|t-r=.fn sec1 t)do.
  t=.r end. {.r
)
   fn=.f1
   0.00001 secant 0.5 1
2
   fn=.f3
   0.00001 secant 0.5 1
1.25992
```

Root-finding : Regula Falsi Method

This is a variation of the secant method, sometimes called the Method of False Position, in which the end points at each step are reordered to ensure that the function values at the next iteration are of opposite sign. The left argument is the precision at which iteration is to stop.

```
   regf=.dyad :0
yi=.fn i=.y lint z=.fn every y
if. +./x>|i-y do.r=.i return. end.
if. yi=&*1{z do.r=.x regf(0{y),i
else.r=.x regf i,1{y end.
)
   fn=.f1
   0.00001 regf 0.5 1
1.99999
   fn=.f2
   0.00001 regf 0.5 1
0.517755
   fn=.f3
   0.00001 regf 0.5 1
1.25992
```

Root-finding : Illinois Method

This is a variation adaptation of the previous method which was described in Vector 12.2 pp. 87-94 and Vector 12.4 pp. 98-106. It compensates for a curve turning more steeply on one side of an interval by halving the relevant ordinate. A second iteration step is inserted into `regf`:

```
   illi=.dyad :0
yi=.fn i=.y lint z=.fn every y
if. +./x>|i-y do.r=.i return. end.
if. yi=&*0{z do.
   yi=.fn i=.y lint z=.1 0.5*z
else.yi=.fn i=.y lint z=.0.5 1*z end.
if. yi=&*1{z do.r=.x illi(0{y),i
else.r=.x illi i,1{y end.
)
   fn=.f1
   0.00001 illi 0.5 1
_0.999971
   fn=.f2
   0.00001 illi 0.5 1
0.517755
   fn=.f3
   0.00001 illi 0.5 1
1.25988
```

There is no particular magic about the 0.5, and other fractions could be experimented with. The broad objective of such variations of the Secant Method is either to improve convergence for 'difficult' func-

tions, or sometimes to achieve a result where simpler methods fail. There is no universal best root finding function, and the choice of method in any particular case belongs to the realm of the Numerical Analyst's art.

Integration : Simpson's Method

The following methods require the function to be assigned to `fn` in advance of the call to the verb. The right argument is the range of integration. Simpson's method is the basic way of doing numerical integration when analytic methods fail. The left argument is the number of intervals into which the range must be divided, and must be an even number. The greater the number the greater is the accuracy of the approximation to the integral.

```
   Simpson=.dyad :0
h=.-y-/ .*%x
(h*+/(1,((x-1)$4 2),1)*fn each({.y)+h*i.x+1)%3
)
   fn=.f2
   2 Simpson 0 0.5
0.303737
   4 Simpson 0 0.5
0.303783
   fn=.f1
   4 Simpson 0 0.5
_1.08333
   fn=.f3
   4 Simpson 0 0.5
0.984375
```

Integration : Adaptive Method

This is a development of the Simpson method which subdivides the range of integration in two at each step, and carries on applying Simpson integration recursively in each half until the required level of approximation specified in the left argument is attained.

```
   adapt=.dyad :0
r=.4 Simpson y
if.x>|r- 2 Simpson y do. return.
else.m=.0 1|.each y,each-:+/y
r=.+/(x adapt {.&>m), x adapt {:&>m end.
)
   fn=.f3
   0.00001 adapt 0 0.5
0.984375
   fn=.f1
   0.00001 adapt 0 0.5
_1.08333
   fn=.f2
```

```
   0.00001 adapt 0 0.5
0.303786
```

Integration : Romberg's Method

The next variation is the Romberg method which uses series expansions for the error terms in Simpson approximation and applies corrections to progressively more refined Simpson estimates. The process is too complicated to describe in succinct comments, but can be found in most elementary texts on Numerical Analysis.

```
   romb=.dyad :0
r=.>(2^1+i.x)Simpson every<y
while.1<#r do.
  r=.}.r+(_1+2^4*2+x-#r)%~r-_1|.r end.
)
   Romberg=.dyad :0
t=.(i=.1)romb y
while.(x<|t-r=.(i=.>:i)romb y) do.t=.r end.
)
   fn=.f2
   0.00001 Romberg 0 0.5
0.303783
   fn=.f1
   0.00001 Romberg 0 0.5
_1.08333
   fn=.f3
   0.00001 Romberg 0 0.5
0.984375
```

These few verbs and adverbs make root-finding and integration both practical and self-describing, and thereby give some insight into the routines offered within the more sophisticated mathematical packages.

Code Summary

Linear Interpolation

```
   lint=. dyad : '((|.x)-/ .*y)%-/y'
```

The remaining verbs require fn to be redefined as functions such as

```
   f1=.#.&1 _1 _2   NB. polynomial x²-x-1      NB. test case 1
   f2=.2&o.-(*^)    NB. cos x - xeˣ            NB. test case 2
   f3=.2&(-^&3)     NB. 2 - x³                 NB. test case 3
```

Root-finding

```
    Newton=.adverb : ']-x%x D.1'(^:_)("0)

   secant=.dyad :0  NB. secant method
```

```
t=.fn sec1 y
while.(x<|t-r=.fn sec1 t)do.
  t=.r end. {.r
)
   sec1=.adverb : '}.@],] lint fn every@]'

   regf=.dyad :0           NB. regula falsi
yi=.fn i=.y lint z=.fn every y
if. +./x>|i-y do.r=.i return. end.
if. yi=&*1{z do.r=.x regf(0{y),i
else.r=.x regf i,1{y end.
)
   illi=.dyad :0           NB. Illinois method
yi=.fn i=.y lint z=.fn every y
if. +./x>|i-y do.r=.i return. end.
if. yi=&*0{z do.
   yi=.fn i=.y lint z=.1 0.5*z
else.yi=.fn i=.y lint z=.0.5 1*z end.
if. yi=&*1{z do.r=.x illi(0{y),i
else.r=.x illi i,1{y end.
)
```

Integration

```
   Simpson=.dyad :0
h=.-y-/ .*%x
(h*+/(1,((x-1)$4 2),1)*fn every({.y)+h*i.x+1)%3
)
   adapt=.dyad :0
r=.4 Simpson y
if.x>|r- 2 Simpson y do. return.
else.m=.0 1|.each y,each-:+/y
r=.+/(x adapt {.&>m), x adapt {:&>m end.
)
   Romberg=.dyad :0        NB. Romberg method
t=.(i=.1)romb y
while.(x<|t-r=.(i=.>:i)romb y) do.t=.r end.
)
   romb=.dyad :0
r=.>(2^1+i.x)Simpson every<y
while.1<#r do.
  r=.}.r+(_1+2^4*2+x-#r)%~r-_1|.r end.
)
```

24. Here we go round… and round and round…

Principal Topics : gerund, Balanced rounding

Do you feel mildly irritated when a report says something like "the figures may not add up to exactly 100 because of rounding"? I do. For one thing it would probably be just as easy to make the figures add up as to make the excuse, and in any case surely the whole essence of rounding is to make figures tally. You have probably guessed the next bit is going to be "it's easy in J", and yes, you are right - moreover it provides a nice illustration of the development of a simple, but not trivially simple, J verb. So let's begin by breaking down the process of rounding.

Consider the problem of rounding a value to a given number, say n, of decimal places. The simplest approach involves a three stage process. In the first stage the number is raised by moving the decimal point n positions to the right, the second stage consists of swinging the resulting up or down to the nearest integer according to whether the fractional part of the resulting number is above or below 0.5, and the third stage reverses the first stage, that is it lowers the numbers by moving them n positions to the left. "Moving the decimal point" is in turn a process which, in more exact terms, consists of multiplying by 10 to the power n, with positive n meaning move to the right and negative n meaning move to the left. These decimal point movements can be summarised as

```
   pow=.10&^         NB. 10 to the power
   raise=.(*pow)~          NB. 1 raise 2.57 is 25.7
   lower=.(%pow)~          NB. 1 lower 25.7 is 2.57
```

Next let's sharpen the swinging process. Again this is a sequence of two simpler processes, the first consisting of adding 0.5, and the second taking the floor:

```
   swingu=.<.@+&0.5        NB. move to nearest integer
   swingu 4.4 5.6 2.5
4 6 3
```

Notice that 2.5 goes up to 3 - more of that later. At first sight the symmetry of the three stage process - raise, swing, lower - might suggest that these three verbs composed as a fork would be appropriate. However maturer consideration shows that this is not the case, since the essential operation of a fork can be summarised informally

as : first execute the left and right prongs concurrently, then execute the middle prong on the transformed data, whereas in this case the lowering can only take place after the other two processes have completed, and so raise and lower are not concurrent.

There *is* in fact a fork present in the rounding process, but it the lower sub-process which forms its central prong, the other prongs consisting of (1) the raising and swinging sub-process in sequence, and (2) the re-extraction of n, the number of decimal places which was "consumed" in the raising sub-process.

This analysis leads to the following development of a rounding algorithm :

```
   rnd=.[lower swingu@raise      NB. syntax is 1 rnd 2.57
```

The reason for the 'u' in `swingu` is that swinging possesses a degree of asymmetry in that a number with a fractional part which is exactly equal to 0.5 is rounded up. The 'mirror image' verb

```
   swingd=.<.@+&0.5        NB. move to nearest integer, 0.5
goes down
   swingd 4.4 5.6 2.5
4 6 2
```

has the opposite effect. For most practical purposes the two can be used interchangeably, that is `rnd` is equivalent to

```
     rndd=.[lower swingd@raise   NB. syntax is 1 rndd 2.57
```

In the days before computers, the practice was sometimes followed of rounding exact halves to the nearest even integer in the hope that any errors so arising might roughly balance out. A 'neutral' swing verb `rndn` incorporating this can be constructed using a gerund which is J's mechanism for the case statement.

```
   isodd=.2&|              NB. 1 for an odd integer, 0 for even
   getfrac=.1&|            NB. Get the fractional part of a
number
   swing=.swingd`swingu @.(isodd&(-getfrac))
```

If `swing` is used the `(isodd&(-getfrac))` decision has to be made separately for each value in a vector and so the `rndn` must be written

```
     rndn=.[lower swing every@raise           NB. syntax is 1
rndn 2.57
```

Now think about how to extend this basic rounding algorithm to do **balanced rounding**, that is the process of making the rounded values of a vector tally exactly to their rounded total. This problem happens typically with percentages, for example the three values in

```
   a=.36.24 29.73 34.03
   +/a
100
```

total exactly 100, but the individual values round to 36.2 29.7 and 34.0 which total 99.9 .

```
   1 rnd a
36.2 29.7 34
```

Begin by considering what adjustment might sensibly be made in the absence of a calculating device. This would probably be something like: following basic rounding, look for the value whose fractional part after raising comes closest to 0.5 (that is the 36.24) and round that one up. If the deficit between the rounded total and 100 had been 0.2 say, the two values nearest to 0.5 would be rounded up, and so on. Since this procedure happens strictly in the swinging stage, the structure of an algorithm for balanced rounding is going to differ only in one of the verbs already developed for rounding, so write it as

```
    brnd=.[lower balance@raise NB. Balanced round
```

The problem is then reduced to that of sharpening what is meant by the verb 'balance'. Take `swingu` as a starting point. Instead of adding 0.5 indiscriminately to all the floors in preparation for lowering, it would be better if all raised values were initially truncated (that is floored), and then a value of 1 added to as many as are necessary to fill the gap between the sum of the raised values (1000) and the sum of their floors (999). Call this gap the rounding gap which, translating the previous sentence into J, is `(+/)  -  +/@<.`

Clearly the order of candidature for adding a 1 is that of the size of the fractional part and so a verb must be constructed which determines the ranking of the fractional parts.

Define

```
   getfrac=.1&|      NB. get fractional part
   rkd=:/:@\:        NB. rank a vector downwards
   rkd 4 2 7 1       NB. eg. 7=max and so has rank 0
1 2 0 3
```

and compose these two verbs, which is incidentally a nice little illustration of the use of the conjunction *bond* (`&`)

```
   (rkd&frac)1 raise a
1 0 2
```

This says that, in the present example, the biggest fractional part is associated with the first term. Since the rounding gap is 1, this is the only item to which 1 will be added at the swing stage, and so the result of the comparison `rkd&getfrac < calcrgap` will supply exactly the right mix of 1s and 0s to add prior to lowering.

For `a` the rounded total is an integer and so too is the rounding gap. When this is not the case as with

```
   b
1.631 0.478 1.939 4.236
   2 raise b
163.1 47.8 193.9 423.6
   +/2 raise b
828.4
   +/<.2 raise b
826
```

where the rounding gap is 2.4, only two of the items require 1 to be added, whereas if the rounding gap had been 2.6 then the round of the sum would be 8.29 and <u>three</u> of the items would require 1. Thus the rounding gap itself should be swung in order that the balance property be fulfilled that the sum of the rounded values should exactly equal the round of the sum of the original values. This leads to the definition of the verb

```
   calcrgap=.swingu@(+/) - +/@<. NB. Calculate rounding gap
```

and the verb `balance` completes the operation of adding 1 to qualifying items before lowering

```
   balance=:(rkd&getfrac < calcrgap) + <. NB. adjusted floor
```

All the elements of `brnd` are now in place and so its definition is complete.

```
   2 brnd b
1.63 0.48 1.94 4.23
   (+/b),(+/2 rnd b),+/2 brnd b
8.284 8.29 8.28
```

Finally here is everything brought together to bring this well-rounded article (pun intended!) to a close :

Code Summary

```
rnd=.[lower swingu@raise               NB. y to x dec.pl, 0.5
rnds up
  raise=.(*pow)~                     NB. multiply y by 10^x
    pow=.10&^                        NB. 10 to the power y
  swingu=.<.@+&0.5                   NB. adjust y to nrst integer
  lower=.(%pow)~                     NB. divide y by 10^x

rndd=.[lower swingd@raise            NB. y to x dpl., 0.5 rnds down
   swingd=.>.@-&0.5                  NB. move to nearest integer

rndn=.[lower swing every@raise       NB. 0.5 rnds to nearest even
   swing=.swingd`swingu @.(isodd&(-getfrac))   NB. gerund
   isodd=.2&|                        NB. 1 for an odd integer, 0
for even

brnd=.[lower balance@raise           NB. balanced rounding,  OK
  balance=.(rkd&getfrac<calcrgap)+<. NB. y to nrst integer
    calcrgap=.swingu@((+/)- +/@<.)   NB. calculate rounding gap
      getfrac=.1&|                   NB. get fractional part of y
      rkd=./:@\:                     NB. rank y downwards
```

25. Two for the Price of One

Principal Topics : ^: (*power* conjunction) \ (*infix* adverb) {: (*take*) upper triangular matrices, Stern-Brocot trees, Farey series, closest rational approximations

J has two ways of representing pairs of numbers as a single atomic entity, viz. `ajb` and `arb`. It is customary to associate `ajb` with complex numbers, and indeed E #13 ("If you think J is complex try j") deals with the associated arithmetic in some detail. However there are other situations where it is convenient to deal with number pairs as single entities; E #14 "j complex? you bet!") illustrated how the `ajb` representation could be used in the context of betting and odds. Representing all rational numbers systematically is another such context in which, rather surprisingly, the `ajb` representation is more helpful than the `arb` one. The starting point is the so-called Stern-Brocot tree which is described below in a combination of words and J, with emphasis on those J features which are particularly useful. On first sight this may seem a rather 'pure' mathematical concept, but as the article goes on to show it has a practical usefulness in finding rational approximations to irrational numbers.

The Stern-Brocot Tree

The totality of rational fractions can be systematically generated by a binary tree named after the two mathematicians (German and French respectively) who first proposed it. Starting with the fractions `0j1` and `1j0`, at every step a new fraction is inserted between every pair of fractions currently in the list, the new fraction consisting of the sums of the neighbouring numerators and denominators.

```
   x=.0j1 1j0
   step=.dyad : 'x,(x+y)'
```

Since items are to be processed in overlapping pairs, use the dyadic *infix* adverb (\) with `step/` as its verb. The final *take* (`{:`) is because the last item in the argument must be carried forward :

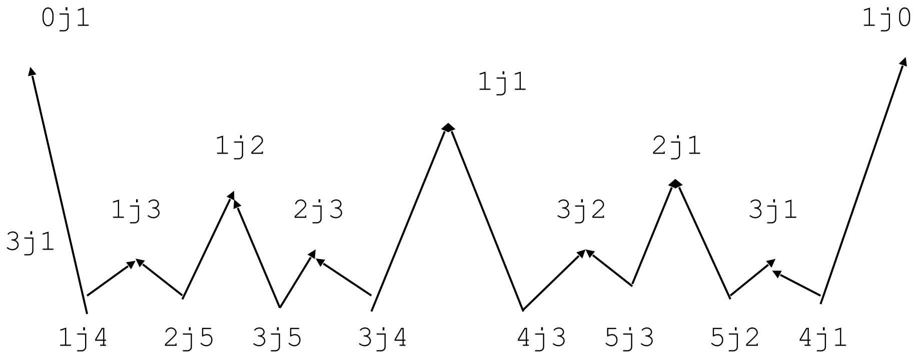
```
   SB=.monad : '(;2 step/\y),{:y'
   SB x
0j1 1j1 1
```

The *power* conjunction (`^:`) is used to generate increasingly long Stern-Brocot tree lists :

```
   SB^:4 x
```

```
0j1 1j4 1j3 2j5 1j2 3j5 2j3 3j4 1j1 4j3 3j2 5j3 2j1 5j2 3j1 4j1
1
```

To see the sense in which this is a binary tree display the successive powers of SB adding explicitly the branches which connect upwards from the final line :



Farey Series

The Farey series is a way of systematically representing all fractions. These appear as the left hand half of Stern-Brocot_trees. Use the auxiliary verb ut (upper triangular)

```
   ut=.<:/~@i.
   ut 5
1 1 1 1 1
0 1 1 1 1
0 0 1 1 1
0 0 0 1 1
0 0 0 0 1

   t=.1 2 3 4 5
   f=.t,each/t              NB. square of all possible pairs
```

j./ converts each 2-list into the corresponding ajb form. ut confines items to regular fractions :

```
   |:(ut 5)*each j./each f
```

1j1	0	0	0	0
1j2	2j2	0	0	0
1j3	2j3	3j3	0	0
1j4	2j4	3j4	4j4	0
1j5	2j5	3j5	4j5	5j5

This representation is <u>not</u> a tree form on account of the multiple representations of e.g. `1j2` as `2j4`, `3j6`, etc. Another way of presenting the Farey series uses the `arb` representation, but this time cancellation to lowest rational form occurs :

```
    |:(ut 5)*x:t%/t=.>:i.5
  1   0   0   0 0
1r2   1   0   0 0
1r3 2r3   1   0 0
1r4 1r2 3r4   1 0
1r5 2r5 3r5 4r5 1
```

Finding closest approximations

A practical use of Stern-Brocot_trees is to find close rational approximations to irrational numbers. First the value of a fraction `ajb` is given by

```
   value=.(%/ @ +.)every
   value 2j3
0.666667
```

using the fact that +. transforms `ajb` into the 2-list `a b`. The values in an Stern-Brocot_tree list which bound a given value x below and above are then

```
   clappd=.dyad :'(_1 0++/x>y){y=.value y'
   z=.SB^:(4)0j1 1j0
   0.35 clapp z
0.3333 0.4
```

It is not necessary to use `0j1` and `1j0` as start values. For example pi is approximated using the power 10 list by

```
   z=.SB^:(10)3j1 22j7
   (o.1) clapp z
3.140625 3.141667
```

A small modification gives the bounding values in fraction form :

```
   clappf=.dyad :'(_1 0++/x>value y){y'
   z=.SB^:(16)3j1 22j7
   (o.1) clappf z
333j106 355j113
```

If there is a requirement is for a rational approximation correct within a given tolerance, construct an iterative verb such as

```
   approx=.dyad : 0
r=.y                          NB. initial upper and lower bounds
while.({:x)<<./t=.|({.x)-({.x) clapp r do.
r=.SB r end.                  NB. do further Stern-Brocot step
(t=<./t)#({.x)clappf r        NB. choose closest value of two
)
   (pi,0.00001) approx 3j1 22j7
355j113
```

Here are rational approximations to e :

```
   z=.SB^:(16)27j10 28j10
   (^1) clappf z
2528j930 7285j2680
```

Code Summary

```
   SB=.monad : '(;2 step/\y),{:y'NB. Stern-Brocot list
      step=.dyad : 'x,(x+y)'
      x=.0j1 1                      NB. argument for SB
     f=.t,each/t=.>:i.5
   |:(ut 5)*each j./each f          NB. Farey series
      ut=.<:/~@i.                   NB. upper triangular matrix
```

The verbs clapp and clappf return closest Stern-Brocot approximation intervals in decimal and fractional form respectively.

```
   clapp=.dyad :'(_1 0++/x>y){y=.value y'
      value=.(%/@+.)every
   clappf=.dyad :'(_1 0++/x>value y){y'

   approx=.dyad :0
r=.y                          NB. initial upper and lower bounds
while.({:x)<<./t=.|({.x)-({.x) clapp r do.
r=.SB r end.                  NB. do further Stern-Brocot step
(t=<./t)#({.x)clappf r        NB. choose closest value of two
)
```

26. Working in Groups

Principal Topics : " (*rank* conjunction) `|.` (*shift*), `/:` (*grade up*) `\:` (*grade down*) `|:` (*transpose*) `` ` `` (*gerund*) `~` (*reflex*) subgroups, identity, inverse, cyclic groups, anti-cyclic groups, dihedral group.

Groups in mathematics are a reflection of patterns observed in what seem to be totally disparate contexts. Given that some of J's founding concepts are based in mathematical ideas, it is not too surprising that the primitive verbs exhibit their own form of group behaviour. To start with, here is some simple geometry -

Plane transformations

Suppose the character 'J' is fitted into a four by four character frame thus

```
   ]J=.4 4 $'   *   **  * **'
   *
   *
*  *
 **
```

To turn it about its horizontal middle line do

```
   (X=.|.)J
 **
*  *
   *
   *
```

To turn J about a north-west/south-east axis :

```
  (Q=.|:)J
  *
   *
   *
***
```

The effects of X then Q, and of Q then J are

```
┌────┬────┐
│ *  │*** │
│*   │   *│
│*   │   *│
│ ***│  * │
└────┴────┘
```

Use a *gerund* to generate two more verbs P and Y :

```
      ((Y=.X&.Q)`(P=.Q&.X))/.  J
```

```
*
*
*  *
 **

 ***
*
*
 *
```

Two things are needed to complete the repertoire needed to form a group. First a half turn, for which either `R` or `S` could be applied twice (e.g.`H=.R^:2`), or `X` and `Q` can be continued with as atomic items :

```
    (H=.X&.Q&X)J
 **
*  *
*
*
```

... and secondly a 'do nothing' operation `I=.[` or equivalently `I=.]` .

There are many alternative ways of defining these eight verbs. For example, using the rank conjunction, reverse 'at rank 1' means 'at list level' so that each member of the stack of four lists is separately reversed instead of the stack as a whole, which results in a Y reflection. The full set of eight transformations are given in the table below with some of the many possible variants which are increased on account of the equivalence of `&` and `@` in this context .

	alternative form	meaning	algebra
`I=. [`	`NB. ]`	`identity`	
`H=. X&.Q&X`	`NB. X"1@X`	`half-turn`	`H=YX=XY`
`X=. |.`	`NB.`	`refln in Ox`	`X=QS=YH`
`Y=. X&.Q`	`NB. X"1`	`refln in Oy`	`Y=QR=XH`
`S=. Q&X`	`NB. X"1@Q`	`rotn π/2 clockwise`	`S=QX=YQ`
`R=. X&Q`	`NB. Q@X"1`	`rotn π/2 a-clockwise`	`R=XQ=QY`
`P=. Q&.X`	`NB. X&R`	`rotn in x=y`	`P=XS=YR`
`Q=. |:`	`NB.`	`rotn in x=-y`	`Q=YS=XR`

The column headed 'algebra' contains some equivalences in which the conjunction `&` has been elided. A full composition table for the conjunction `&` applied to the eight verbs is

```
      I R H S X P Y Q

I     I R H S X P Y Q
R     R H S I P Y Q X
H     H S I R Y Q X P
S     S I R H Q X P Y
X     X Q Y P I S H R
P     P X Q Y R I S H
Y     Y P X Q H R I S
Q     Q Y P X S H R I
```

This is an example of a mathematical **group**, or more fully a group of order eight since the body of the table contains nothing other than the basic eight distinct elements (this property is called **closure**).

Properties of groups

A group is characterised by the following properties:

(a) the underlying binary operation is associative, that is **a(bc) = (ab)c** where **a**,**b**, and **c** are any elements of the group;

(b) there is a single **identity element**, in this case, **I**, with the property that **Ia = aI=a** for all elements **a** belonging to the group; and

(c) every element **e** has a **left and right inverse**, that is there are elements **e**' and **e**'' with the properties that **ee'=e''e=I**. If left and right inverses are equal for all **a** the group is **symmetric** and is known as an **Abelian** group.

The body of the operation table given above contains no elements outside the group, which is often expressed by saying that the group is 'closed under the operation'. Specifically the eight transformations of the plane which transform a square into its own outline are closed under the operation 'perform in sequence', which in J is rendered by the conjunction *atop* (`@`). Also because there are eight elements the group is said to be of **order** 8.

Groups and subgroups

Now look for structures in the ways in which group operations combine. To start with, the first four rows and columns in the above table themselves form a group of order four, which is structurally identical (isomorphic) to the addition table for addition in arithmetic modulo 4 (that is clock arithmetic applied to a clock with just four numbers). This table is :

```
   cyc=.| +/~@i.     NB. cyclic group of order y.
   cyc 4
0 1 2 3
1 2 3 0
```

```
2 3 0 1
3 0 1 2
```

which can be transcribed into subsets of the 8 by 8 table by

```
    (cyc 4)&{ each 'IRHS';'XPYQ'
┌────┬────┐
│IRHS│XPYQ│
│RHSI│PYQX│
│HSIR│YQXP│
│SIRH│QXPY│
└────┴────┘
```

The first of these is a subgroup of the full table but the second is not a subgroup because it has no identity element (that is an element with the property that Ia = aI=a for all elements of the subgroup). The two bottom quadrants have row shifts which go in the opposite direction to cyc 4 . A general algorithm for this uses the minus table for i.n in modulo n arithmetic together with two hooks :

```
   ac=.|(+-/~@i.)   NB. anti-cyclic table of order y
   ac 4
0 3 2 1
1 0 3 2
2 1 0 3
3 2 1 0
```

`(ac 4){'XPYQ'` and `(ac 4){'IRHS'` give the bottom left-hand and right-hand quadrants respectively of the full table. These two expressions can be merged as `(4+ac 4){'IRHSXPYQ'`, and the indices in the string `'IRHSXPYQ'` of the cells in the four quadrants are, in clockwise order, given by `(cyc 4)`, `(4+cyc 4)`, `(ac 4)` and `(4+cyc 4)`. These define the non-symmetric group which is known as the **dihedral group** of order 4. An adverb based on a hook which adds the *tally* of a matrix to itself (recall that *tally* for a matrix is simply the number of rows) leads to the following general definition of dihedral groups :

```
   a2n=.(+#)@ NB. adverb : adding 2 to the power n
   di=.(cyc,.cyc a2n),((ac a2n),.ac)NB. Dihedral group order n
      di 4
0 1 2 3 4 5 6 7
1 2 3 0 5 6 7 4
2 3 0 1 6 7 4 5
3 0 1 2 7 4 5 6
4 7 6 5 0 3 2 1
5 4 7 6 1 0 3 2
6 5 4 7 2 1 0 3
7 6 5 4 3 2 1 0
```

and

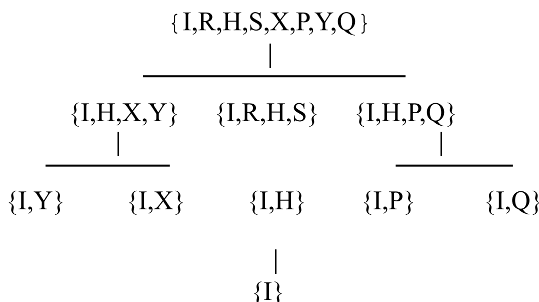
```
   (di 4){'IRHSXPYQ'
```

generates the 8 by 8 composition table given above.

The table of `I`, `H`, `X` and `Y` rotations only is simply the dihedral group of order 2 (D2).

```
   (di 2);(di 2){'IHXY'
┌───────┬────┐
│0 1 2 3│IHXY│
│1 0 3 2│HIYX│
│2 3 0 1│XYIH│
│3 2 1 0│YXHI│
└───────┴────┘
```

and the set of all possible subgroups is exhibited in the following lattice diagram in which a connecting line indicates that the lower node is a subgroup of the higher one.



On a two-dimensional plane there are eight shape-preserving transformations of the square, namely four rotations about the centre through 0,1, 2 and 3 right angles, together with four reflections, two in the axes joining midpoints of opposite sides, and two in the diagonals. These geometrical transformations can be demonstrated by performing J transformations on the matrix `i.4 4`, for example reflections about the x axis and about the diagonal x=-y are described by

```
   (|.m);|:m=.i.4 4
┌───────────┬──────────┐
│12 13 14 15│0 4  8 12 │
│ 8  9 10 11│1 5  9 13 │
│ 4  5  6  7│2 6 10 14 │
│ 0  1  2  3│3 7 11 15 │
└───────────┴──────────┘
```

The full set of eight transformations are given in the table below, in which the final column gives an algebraic shorthand in which, for ex-

ample, R=QX means the rotation R is equivalent to a reflection X followed by a reflection Q.

```
                  alternative   meaning                    algebra
 I          NB.              identity
 H=. |.@|"1 NB. |."1@|. half-turn                   H=YX=XY
 X=. |.     NB.              refl in Ox              X=QR=YH
 Y=. |."1   NB.              refl in Oy              Y=QS=XH
 R=. |:@|.  NB. |."1@|: rotn pi%2 clockwise         R=QX=YQ
 S=. |.@|:  NB. |:@|."1 rotn pi%2 a-clockwise       S=XQ=QY
 P=. |.@R   NB. |:"1@S  rotn in x=y                 P=XR=YS
 Q=. |:                 rotn in x=-y                Q=YR=XS
```

general the dihedral group Dn describes the symmetries of an n-sided regular polygon. For example

```
   di 3
0 1 2 3 4 5
1 2 0 5 3 4
2 0 1 4 5 3
3 4 5 0 1 2
4 5 3 2 0 1
5 3 4 1 2 0
```

gives the non-symmetrical group of symmetries of the equilateral triangle. 0 1 and 2 correspond to rotations by 0, $\pi/3$ and $2\pi/3$, and 3, 4 and 5 to reflections in the three perpendicular bisectors of the sides.

Groups and grade

Group relationships exists between the grade operations, `/:` and `\:`, applied to permutation vectors and the symmetries of the square. To demonstrate here are two verbs with obvious similarities, which convert permutation vectors to matrices and vice versa :

```
   pm=.=/~ i.@#   NB. transforms permn vector to matrix
   pm 1 3 2 0
0 0 0 1
1 0 0 0
0 0 1 0
0 1 0 0

   pv=.+/ .*~i.@#  NB. transforms perm matrix to vector
   pv pm 1 3 2 0
1 3 2 0
```

The compositions of grade verbs which match the analogous transformations of the permutation matrix can be added to the previous table in the following way:

```
                                grade verb
                              composition    alternative
I                   NB.
H=. |.@|"1          NB.   \:@\:           |.@X
X=. |.              NB.   /:@\:           |.@H
Y=. |."1            NB.   \:@/:           |.
R=. |:@|.           NB.   \:@\:@\:        /:@|.
S=. |.@|:           NB.   \:              |.@Q
P=. |.@R            NB.   /:@\:@\:        \:@|.
Q=. |:              NB.   /:              |.@S
```

The binary operation 'perform in sequence' is again modelled by the conjunction *atop* . Since the grade operations are dyadic, @ may not be elided, that is if u and v are any of the eight permutation operations, u@v (= u v y) is not in general equivalent to the hook uv which equals y u v y .

Code Summary

```
cyc=.| +/~@i.      NB. cyclic group of order y
ac=.|(+-/~@i.)     NB. anti-cyclic table of order y
a2n=.(+#)@         NB. adverb : add 2 to the power n
di=.(cyc,.cyc a2n),((ac a2n),.ac)  NB. Dihedral grp order n
pm=.=/~ i.@#       NB. Transforms permn vector to matrix
pv=.+/ .*~i.@#     NB. Transforms perm matrix to vector
```

27. All but one

Principal Topics : " (*rank* conjunction) \. (*suffix*), |: (*transpose*), ~ (*passive*) determinants, minors, cofactors, subtotallng.

A frequent requirement in applied mathematics and statistics is to evaluate sums and products omitting just one variable or dimension. The notion of 'all but one' can be interpreted in two ways, depending whether the 'one' is to be systematically omitted, or obtained by a merge with an existing dimension, that is by reduction.

Retaining all but one

As an example of the first case, adding or multiplying 'all but one' items in a list progressively can be done by using the hook `f⁻¹~f/` which means `{f/x}f⁻¹ x`, for example :

```
   i.5
0 1 2 3 4
   (-~+/)i.5        NB. sum 5 items omitting one at a time
10 9 8 7 6
   (%~*/)1+i.5      NB. multiply 5 items omitting one at a time
120 60 40 30 24
```

This extends readily to objects of higher rank :

```
    ]b=.i.3 4
0 1  2  3
4 5  6  7
8 9 10 11
   (-"1~+/)b        NB. sums of all rows but one
12 14 16 18
 8 10 12 14
 4  6  8 10
   (-"2~+/"1)b      NB. sums of all columns but one
 6  5  4  3
18 17 16 15
30 29 28 27
```

The rule is that the rank operand for the left verb should be one greater than that of the right verb.

Generalising 'retaining all but one'

Another tool which deals with 'retaining all but one' is the *suffix* adverb which eliminates `n` items from a list in all possible ways :

```
   (1+\.i.5);(2+\.i.5);(3+\.i.5)
┌───────┬─────┬───┐
│1 2 3 4│2 3 4│3 4│
│0 2 3 4│0 3 4│0 4│
│0 1 3 4│0 1 4│0 1│
│0 1 2 4│0 1 2│   │
│0 1 2 3│     │   │
└───────┴─────┴───┘
```

In the J line above + is monadic, which for real numbers is a 'do nothing' verb, that is the left arguments 1, 2 and 3 are not added to elements of `i.5` but are arguments of the derived verb `+\.` which indicate how many items are to be dropped progressively working from the left. The first case with 1 as left argument is thus the 'all but one' case.

An application of this is the calculation of minors of a determinant. Consider the rank 2 object `z` :

```
   z
3 6 7
9 3 2
9 7 7
   (1&(+\.))"2 z    NB. select 'all but' one rows
9 3 2
9 7 7

3 6 7
9 7 7

3 6 7
9 3 2
```

Now combine the structural verb *transpose* with the adverb *suffix* to switch rows and columns for all but one row at a time :

```
   transuff=.1&(|:\.)"2
   <"2 transuff z
┌───┬───┬───┐
│9 9│3 9│3 9│
│3 7│6 7│6 3│
│2 7│7 7│7 2│
└───┴───┴───┘
```

and the result is a list of the transposed planes of `(1&(+\.))"2 z`.

Do this twice using the *power* conjunction to generate three boxes within each of the above boxes; the row/column switch is fortuitously reversed and the minors of `z` obtained :

```
   <"2 transuff^:2 z
```

```
┌───┬───┬───┐
│3 2│9 2│9 3│
│7 7│9 7│9 7│
├───┼───┼───┤
│6 7│3 7│3 6│
│7 7│9 7│9 7│
├───┼───┼───┤
│6 7│3 7│3 6│
│3 2│9 2│9 3│
└───┴───┴───┘
```

Define

```
   minors=.transuff^:2      NB. minors unboxed
   det=.-/ .*               NB. determinant
```

The determinants of the minors are given by

```
   ]dz=.det every <"2 minors z
 7  45  36
_7 _42 _33
_9 _57 _45
```

This is verified by using the verb `det` dyadically :

```
      (det z);dz det |:z
┌─┬──────┐
│3│3  0 0│
│ │0 _3 0│
│ │0  0 3│
└─┴──────┘
```

It is often convenient to use **cofactors**, that is the signed determinants of minors. This requires multiplication by a matching matrix whose diagonals are alternately +1 and -1. One way of obtaining this matrix is :

```
   signs=.monad : '-_1+2*2|+/~i.#y
```

so that

```
   cof=.signs * det every @<"2@minors
   cof z
 7 _45  36
 7 _42  33
_9  57 _45
```

To summarise this section :

```
   transuff=.1&(|:\.)"2                    NB. transpose with suf-
fix
```

```
minors=.transuff^:2                     NB. minors unboxed
det=.-/ .*                              NB. determinant
signs=.monad :'-_1+2*2|+/~i.#y.'        NB. alternate 1,_1
cof=.signs * det every@<"2@minors       NB. cofactors
```

Reducing all but one

Rank again is at the heart of the matter, especially as typical experimental data is structured so that dimensions correspond to variables. However, J inherently structures higher rank data as lists, then lists of lists, lists of lists of lists and so on, which implies a nesting within dimensions which is not usually reflected in the 'real world' variables to which the dimensions correspond. For definiteness use `q` to demonstrate :

```
   ]q=.i.2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
```

The standard definition of *rank* is:

```
   rk=.#@$           NB. rank
```

`+/` *inserts* +'s between the list at the topmost level, thereby reducing rank by one, that is merging planes :

```
   +/q               NB. equivalent to +/""""n q for n>2
12 14 16 18
20 22 24 26
28 30 32 34
```

Explicit rank conjunctions allows such reduction to take place at any level in the rank hierarchy :

```
   (+/"1 q);(+/"2 q)      NB. merge cols ; merge rows
┌────────┬───────────┐
│ 6 22 38│12 15 18 21│
│54 70 86│48 51 54 57│
└────────┴───────────┘
```

Progressive reduction to the level of a rank 1 object is obtained using a recursive verb :

```
   sum=.sum@(+/)` ] @.(=&1@rk)  NB. sum down to rank 1
   sum q
60 66 72 78
```

The above values are readily verifiable as the column sums of `q`. To find other such sums, say row sums, transpose the data so as to bring the corresponding dimension to the top level. This suggests a general verb which takes the list `i.n` and performs the a set of `n` possible shifts necessary to bring each item in turn into the leading position :

```
   shifts=.|.each<  NB. all distinct shifts of a list
   shifts i.rk q
┌─────┬─────┬─────┐
│0 1 2│1 2 0│2 0 1│
└─────┴─────┴─────┘
```

If the argument to `shifts` is a list generated by `i.`, the result is left arguments to *transpose* which provide all the restructured forms of `q` to which `sum` can be applied. This in turn is determined by the rank of the data matrix so define

```
   targs=.shifts@(i.@rk)  NB. arguments for |:
   targs q
┌─────┬─────┬─────┐
│0 1 2│1 2 0│2 0 1│
└─────┴─────┴─────┘
```

The full set of transpositions to supply all possible sums by dimension is then

```
   transposes=.targs |:each <     NB. reqd. transposes
   transposes q
┌───────────┬─────┬────────┐
│ 0  1  2  3│ 0 12│ 0  4  8│
│ 4  5  6  7│ 1 13│12 16 20│
│ 8  9 10 11│ 2 14│        │
│           │ 3 15│ 1  5  9│
│12 13 14 15│     │13 17 21│
│16 17 18 19│ 4 16│        │
│20 21 22 23│ 5 17│ 2  6 10│
│           │ 6 18│14 18 22│
│           │ 7 19│        │
│           │     │ 3  7 11│
│           │ 8 20│15 19 23│
│           │ 9 21│        │
│           │10 22│        │
```

```
│          │11 23│        │
└──────────┴─────┴────────┘
```

These values are readily verifiable by summing the columns in the boxes above :

```
   sum each@transposes q
┌───────────┬──────┬─────────┐
│60 66 72 78│66 210│60 92 124│
└───────────┴──────┴─────────┘
```

To give these sums in dimension order, that is so that the `$each` of the result matches the shape of `q`, write

```
   allsums=.1&|.@(sum each@transposes)
   allsums q
┌──────┬─────────┬───────────┐
│66 210│60 92 124│60 66 72 78│
└──────┴─────────┴───────────┘
```

To summarise this section :

```
   rk=.#@$                          NB. rank
      shifts=.|.each <              NB. all shifts of i.n
   targs=.shifts@(i.@rk)            NB. arguments for |:
   transposes=.targs |:each <       NB. reqd. transpositions
   allsums=.1&|.@(sum each@transposes)
```

For comparison, here is a one-liner picked from the J Software forum some years ago which performs the same function by using the *power* conjunction `^:` to apply `+/` the requisite number of times with transpositions required between steps as in the version above :

```
   mt=.(<@(+/^:(<:@rk)@:|:)"0 _~i.@rk)
   mt q
┌──────┬─────────┬───────────┐
│66 210│60 92 124│60 66 72 78│
└──────┴─────────┴───────────┘
```

Subtotaling

It can be useful to be able to *append* such reductions to the original data as in :

```
  total=.,+/  NB. append totals for leading dimension
  sub=.3 :0
i=.0 [ r=.total y
while.(i<<:rk y)do.r=.total"i r [ i=.i+1 end.
)

   sub q
```

```
 0  1  2  3   6
 4  5  6  7  22
 8  9 10 11  38
12 15 18 21  66

12 13 14 15  54
16 17 18 19  70
20 21 22 23  86
48 51 54 57 210

12 14 16 18  60
20 22 24 26  92
28 30 32 34 124
60 66 72 78 276
```

Multi-statement lines using [as a separator as in `sub` allow something approaching the succinctness of tacit definition. However the individual statements are executed from the right since [itself is just another verb. It is easy to remember that it is [rather than] which is the separator, since, for example, it is 0 to the left which is assigned to `i` in the first line of `sub`. As far as the J interpreter is concerned it is really only one line which is executed; multiple lines are essentially just an orthographic device.

Code Summary

```
   minors=.transuff^:2            NB. minors unboxed
      transuff=.1&(|:\.)"2        NB. transpose with suffix
   det=.-/ .*                     NB. determinant
   cof=.signs * det every@<"2@minors NB. cofactors
      signs=.monad :'-_1+2*2|+/~i.#y.' NB. alternate 1,_1

   transposes=.targs |:each <     NB. reqd. transpositions
       targs=.shifts@(i.@rk)      NB. arguments for |:
          shifts=.|.each <        NB. all shifts of i.n
          rk=.#@$                 NB. rank

   allsums=.1&|.@(sum each@transposes)
      sum=.sum@(+/)`] @.(=&1@rk)  NB. sum down to rank 1

      total=.,+/    NB. append totals for leading dimension
   sub=.3 :0
i=.0 [ r=.total y
while.(i<<:rk y)do.r=.total"i r [ i=.i+1 end.
)
```

28. Have you a weight on your mind?

Principal Topics : ? (*deal*) {: (*take*) # (*copy*) I. (*interval index*) #. (*base*) random exponential, random Normal, Box-Muller formula, boundary values, grouping, rounding, frequency distributions.

Uniform Random numbers

A standard procedure for generate n random numbers in the range (0,1) with all numbers in that range being equi-probable is

```
   rnd=.?@#&0
   rnd 5
0.5377 0.06353 0.7059 0.5188 0.8832
```

Convert a list such as the above to random values in the range [a,b] is a matter of scaling, conveniently supplied by the *base* verb `#.` This transforms [a,b] into the list [(b-a),a] by

```
   relist=.(-/,{:)@|. NB. [a,b] -> [(b-a),a]
   run=.dyad : '>(rnd y)#.each<relist x'
                      NB. x=range [a,b], y=n
   5 11 run 5
5.544 10.69 5.442 8.004 7.305
```

Converting a `rnd n` sequence to a series of uniform random integers from 1 to 20 say is a simple exercise in multiplying and rounding down, for example :

```
   <.20*rnd 5
14 15 14 1 16
```

Weighted Random numbers

If the uniform, that is equi-probable, requirement is abandoned in favour of making some integers more probable than others then weights must applied. These can be specified by a list of values, for example by specifying m numbers (not necessarily integers) which define the relative frequencies of the integers in `i.m`. Weights are more tractable when they are expressed cumulatively and normalised to 1, hence

```
   cumwts=.+/\ % +/              NB. cumulative weights
   cumwts 1 4 3 2
0.1 0.5 0.8 1
```

The primitive `I.` (*interval index*) identifies the appropriate cell into which each item should be placed :

```
   rndw=.cumwts@[  I. rnd@]           NB. weighted random in-
tegers
   1 4 3 2 rndw 10
1 1 2 2 0 3 1 1 3 2
```

As an aside, the syntactic structure of `rndw` is a fork with separate transformations applied to the left and right tines.

Interestingly, reversing the arguments of `rndw` is equivalent to transposing the above binary table, and the result is the cumulative frequency distribution of the set of random drawings :

```
   cdrndw=.rnd@] I. cumwts@[      NB. cumulative distn of above
   1 4 3 2 cdrndw 10
1 5 8 10
```

which happens in the above drawing to match exactly the pattern of cumulated weights. A more general drawing might be

```
   1 4 3 2 cdrndw 100
10 52 84 100
```

still visibly confirming that the weights have been correctly applied.

Random Exponential values

Drawings from a random exponential distribution with a given mean are a simple extension of `rnd` :

```
   rne=.-@* ^.@rnd                NB. rand neg exp. x=mean, y=n
   5 rne 8
0.9258 16.82 14.64 3.177 1.994 24.33 4.793 13.53
   (mean=.+/%#)5 rne 10
5.985
```

Such values are commonly used in simulations where, if the probability of an arrival in a time interval is proportional to the length of the interval and successive arrivals are independent of each other, then statistically the negative exponential distribution is that followed by the gaps between successive arrivals.

Random Normal values

Now that rne and rno are available, random values from the standard Normal distribution are obtainable using a pseudo-random technique known as the **Box-Muller formula**. This involves two separate drawings of random numbers using rnd, one to furnish rne and the other used directly in the right hand multiplicand :

```
   rno=.3 : '(%:2 rne y)*2 o.(rnd y)*o.2'
NB. random Normal(0,1)
   rno 8
_0.6224 0.0922 0.9571 _0.3281 0.02999 1.28 _0.1247 0.02563
```

As an aside, 1 o. (sin) replacing 2 o. (cos) in the right hand multiplicand also delivers random Normal values.

A random drawing from a Normal distribution with a given mean and standard deviation is

```
   rnorm=.4 : '(rno y)#.every<|.x'
                        NB. rand Normal, x=mean,sd, y=n
   10 3 rnorm 8
11.18 10.37 10.55 12.97 7.146 11.75 10.61 11.83
```

Grouping and Rounding

When continuous variables are involved, it is often useful to group values by rounding :

```
   R=.<.@(0.5&+)           NB. round y to nearest integer
   round=.4 : 'R&.(%&x)y'  NB. round y to nearest x

   0.2 round 5 rne 10
7.2 0.2 0.8 6 2.2 5.2 1.4 0 1 2.6

   5 round 100 15 rnorm 10
110 70 30 145 115 105 105 75 100 115
```

Frequency distributions of random samples

Interval Index (I.) provides a means of obtaining indexes of items in intervals defined by the left argument which defines boundaries in ascending order, Items below the lowest boundary have index 0, and #x for those which exceed the highest boundary. Boundary items are assigned to the lower interval.

```
   sortu=.{~ /:                           NB. sort upwards
   fr=.dyad : '+/(=/~.)sortu x I.y'       NB. frequency distribu-
tion
    y
2.7 1.8 0.5 3.3 1.4 3.2 3.1 0
   1 1.5 2 2.5 3 fr y
```

```
2 1 1 1 3
```

The plausibility of the `rnorm` routine can be checked by, for example

```
   (67.5+5*i.14) fr 5 round 100 15 rnorm 100
3 1 2 9 7 10 10 14 14 13 5 4 4 3 1
```

This illustration confirms the general bell shape with values more than two standard deviations away from the mean accounting for about 5% of the total.

Code Summary

```
  rnd=.?@#&0                        NB. random uniform(0,1)
  run=.dyad : '(rnd y)#.each<relist x' NB. rand uniform
     relist=.(-/,{:)@|.
  cumwts=.+/\ % +/                  NB. cumulative weights
  rndw=.cumwts@[  I. rnd@]          NB. weighted random integers
  cdrndw=.rnd@] I. cumwts@[            NB. cumulative distn of
rndw

  rne=.-@* ^.@rnd                   NB. rand neg exp. x=mean, y=n
  rno=.monad : '(%:2 rne y)* 2 o.(rnd y)*o.2' NB. Box-Muller
formula
  rnorm=.4 : '(rno y)#.every<|.x' NB. rand Normal, x=mean,sd,
y=n
  wrnd=.cumwts@[  I. rnd@]          NB. weighted random integers
  cumwrnd=.rnd@] Slt cumwts@[       NB. cumulative distn of above
  round=.4 : 'R&.(%&x)y'            NB. round y to nearest x
      R=.<.@(0.5&+)                 NB. round y to nearest integer
  fr=.dyad : '+/(=/~.)sortu x I.y' NB. frequency distribution
     sortu=.{~ /:                   NB. sort upwards
```

29. Just say it in J - ANOVA

Principal Topics : # (*copy*) /. (*key*) ; (*raze*) ANOVA, between/within groups sums of squares, main effects, treatment/residual sums of squares, Latin square, orthogonality, interactions.

An early claim for APL was that the exactness demanded by expressing mathematics in an executable language could lead to significantly greater understanding of underlying algebra and algorithms than was achievable by conventional mathematical notation. Subsequent experience diluted this claim somewhat since the obscurity which even a few lines of code could generate outweighed the inherent discipline of using a programming language as a learning tool. Nevertheless there are some topics in which the expository power of APL, and now even more strongly J, trounces traditional methods. One of the most conspicuous examples of this is the manner in which sums of squares are partitioned as the first stage of the statistical technique known as Analysis of Variance.

The acronym ANOVA, which is at least as well known by its abbreviation as in its full form, is one of the curiosities of statistical terminology, since it is not a single technique, it is not primarily about variance, and it cannot properly be said to be a process of analysis. I surmise that many generations of students have tackled ANOVA as an arcane piece of algebra involving mysterious 'correction factors', seemingly arbitrary rules concerning denominators, and mantras about 'degrees of freedom', 'interactions' and 'error sums of squares'. Yet if ANOVA is seen as systematic set of transformations of data which all concerned portioning of sums of squares, then its rationale and the algebra itself become plain. I am not trying to underestimate the vast scope of the intellectually demanding science of Design of Experiments which ANOVA leads up to, but rather to suggest that its foundations might be more clearly introduced through the medium of J.

The insight to be obtained from using J assume competence in that language, so that as a starting point the followisng basic verbs can be taken for granted :

```
mean=.+/ % #          NB. Mean of a list
mdev=.-mean           NB. Deviations from mean
ssq=.+/@:*:           NB. Sum of squares of a list
ssqm=.ssq@mdev        NB. Sum of squares about the mean
```

One-way Analysis of Variance

One-way ANOVA is relevant where there are groups of numbers, not necessarily of equal size, but of sufficient general similarity for it to be interesting to speculate on whether the groups are distinguishably different, or whether any variability present is scattered across all the groups. In order to allow the calculations to be checked by mental arithmetic the values used in the following examples are integers. A set of groups of numbers is modelled by a list of lists such as

```
   data1=.1 2 3 6 8;4 8 3;3 4 9 5
   mean every data1
4 5 5.25
```

A common sense way of assessing whether this result shows real differences between groups is to hypothesise no variability within each group, and smooth out any such variability by replacing every value by its group mean. J obliges with the primitive verb *copy* which, operating as the centre tine of a fork, makes as many copies of the numbers as are necessary:

```
   ((#every) # (mean every))data1
4 4 4 4 4 5 5 5 5.25 5.25 5.25 5.25
```

(There are redundant parentheses in the above expression in order to make the fork structure stand out.) The sum of squares of these values about the overall mean is totally free of any within-group variation and so, rationally, is a sound measure of the between-group variation:

```
  bss=.ssqm@(#every # mean every)  NB. between group sum of sqq
  bss data1
3.91667
```

For variability within groups, the procedure is to sum the individual sums of squares about the mean within each group:

```
   wss=.+/@:(ssqm every)       NB. within group sum of sqq
   wss data1
68.75
```

Given the above results, any subsequent analysis of this particular data is clear cut, viz. the within group variability overwhelms the between group variability. As a happy consequence of algebra, which can be investigated separately, it happens that the sum of `bss` and `wss` necessarily equals the sum of squares about the mean which would have arisen with the same set of numbers if there had been no group divisions in the first place. Using *raze*, this is:

```
  tss=.ssqm@;             NB. Total sum of squares
  tss data1
72.6667
```

Hence

```
   aov1=.bss , wss , tss   NB. One-way ANOVA
   aov1 data1
3.91667 68.75 72.6667
```

Here is a consolidation of the verbs which describe this first stage in understanding ANOVA :

```
  bss=.ssqm@(#every # mean every)NB. Between sum of sqq
  wss=.+/@:(ssqm every)          NB. Within sum of sqq
  tss=.ssqm@;                    NB. Total sum of sqq
  aov1=.bss,wss,tss              NB. One-way ANOVA
```

Two-way Analysis of Variance

When data is classified in more than one way, e.g. by the rows and columns of a rectangular matrix, the rows can be boxed separately to give the between rows sums of squares of the rows, and the same process for the transposed matrix gives the between column sums of squares.

```
   data2
6 2 3
2 8 5
5 6 8
   rowssq=.bss@:(<"1)      NB. row sums of squares
   (rowssq data2),(rowssq |:data2)
10.6667 2
```

Now successively subtract row and column means from the matrix and sum the squares of the result :

```
   rowdevs=.mdev every@(<"1)
   ssq ,rowdevs |:rowdevs data2
29.3333
```

This value is the sum of the squares after any row and column effects have been removed. If an **interaction** effect between rows and columns is present, that is the occurrence in a particular cell of one of the experimental units is also dependent on its row and column combination, then 29.3333 is the sum of squares attributable to this factor. If an interaction assumption is not reasonable, 29.3333 must be accounted for as random variation and is called the **residual** sum of squares. Since the interaction assumption is stronger define

```
   iss=.monad : 'ssq,rowdevs |:rowdevs y'
   aov2=.(rowssq every @; |:) , iss , tss
   aov2 data2
10.6667 2 29.3333 42
```

As a check the first three items in the above must be equal to the last.

For square matrices it is possible to superimpose a further classification, oftene called treatments based on a so-called Latin square design such as

```
0 1 2
2 0 1
1 2 0
```

Using the ravel of this matrix as the basis of *box key* to obtain a between-treatments sum of squares :

```
   x=.0 1 2 2 0 1 1 2 0
   x </. ,data2
┌─────┬─────┬─────┐
│6 8 8│2 5 5│3 2 6│
└─────┴─────┴─────┘
   bss x </. ,data2
24.6667
```

which accounts for a further 24.6667 of the previous 29.3333 sum of squares, assuming these were considered as residual.

The property that the sums of row, column and treatment sums of squares is the total sum of squares depends on **orthogonality** in the design. For example an arbitrary arrangement of three 0s, three 1s, and three 2s such as

```
   x1=.1 0 2 2 0 1 0 1 2
```

does not possess this property.

Three-way Analysis of Variance

Where a 2-way experiment is replicated either by simple repetition, or by applying different treatments to each replicate, the data is best presented as a 3-dimensional matrix, for example `data3` which has 4 rows and 5 columns, and within each of the resulting 20 cells each of two treatments are applied :

```
   data3
130  34  20
150 136  25
138 174  96

155  40  70
188 122  70
110 120 104

 74  80  82
159 106  58
168 150  82

180  75  58
126 115  45
160 139  60
```

Consolidate the data by `+/data3`, apply `aov2` and divide by 4 to take account of the fact that each new cell is a sum of 4 original ones. The resulting sum of squares is less than that in the full data because `+/` has eliminated the variation due purely to repetition. A full ANOVA for `data3` is thus given by `aov2` adapted to `aov2r` (standing for ANOVA with repetition)

```
   aov2r=.monad :0
u=.}:(aov2 +/y)%{.$y
u,(t-+/,u),t=.tss y
)
   aov2r data3
10683.7 39118.7 9613.78 18230.8 77647
```

The five values in this list therefore stand for sums of squares for rows, columns, interaction, residual and total, and as before the sum of the first four must equal the fifth.

Now suppose that the several vales in each row/cell result from four different treatments, so that there are now three **main effects** (rows, columns and treatments) with the possibility of three interactions (RC, RT and CT). Averaging out the reduced effect is accomplished by +/

qualified by the three possible ranks, and divided by the appropriate *shape* element

```
   (aov2 +/data3)%4
10683.7 39118.7 9613.78 59416.2
   (aov2 +/"1 data3)%3
354.972 10683.7 5358.94 16397.6
   (aov2 +/"2 data3)%3
354.972 39118.7 3678.61 43152.3
```

All the values required are present but clearly there is some redundancy. The last but one values in the above lists represent the sums of squares due to interactions, the main effects appear twice over, and the residual sum of squares (or equivalently the 3-way interaction effect) is most easily found by subtracting all the main and 2-way interactions from the total sum of squares. All of this is sorted out in

```
   aov3=.monad :0
t=.}:"1(aov2 every (+/y);(+/"2 y);+/"1 y)%$y
t=.(0 1 3{,t),{:"1 t
t,(u-+/t),u=.tss,y
)
   aov3 data3
10683.7 39118.7 354.972 9613.78 3678.61 5358.94 8838.22 77647
```

which give in order row, column, treatment main effects, RC, CT and RT interactions, residual and total sums of squares. If required the residual sum of squares could be broken down further into three-way interactions by extending the existing technique, although from a substantive point of view such interactions are usually difficult to conceive.

The above development shows how the main J primitives match operations which are logically necessary steps for understanding the process of calculating values infor ANOVA tables. Each of the values in an ANOVA list bar the last provides a means of estimating the overall variance present in the data assuming that none of the effects are significant. If these estimates vary by too much (where 'too much' is determined by the so-called F-statistic) then the corresponding effect is deemed to be significant. This is the point at which raw calculation stops and statistical reasoning takes over.

Code Summary

Basic Statistical verbs

```
  mean=.+/ % #              NB. Mean of list
  mdev=.-mean               NB. Deviations from mean
  ssq=.+/@:*:               NB. Sum of squares of a list
  ssqm=.ssq@mdev            NB. Sum of squares about the mean
```

One-way Analysis of Variance (data is list of lists)

```
  aov1=.bss;wss;tss                NB. one-way ANOVA
    bss=.ssqm@(#every # mean every)  NB. between sum of sqq
    wss=.+/@:(ssqm every)          NB. within sum of sqq
    tss=.ssqm@;                    NB. total sum of sqq
```

Two-way Analysis of Variance (data is a numerical matrix)

```
   aov2=.monad : 0          NB. two-way ANOVA
u=.,rowssq every y;|:y
r=.u,(t-+/,u),t=.tss y
)
     rowssq=.bss@:(<"1)            NB. row sums of squares
     rowdevs=.mdev every@(<"1)    NB. remove row&col means
```

Three-way Analysis of Variance (data is a 3-D numeric array)

```
   aov3=.monad :0
t=.}:"1(aov2 every (+/y);(+/"2 y);+/"1 y)%$y
t=.(0 1 3{,t),{:"1 t
t,(u-+/t),u=.tss,y
)
```

30. Just what do they sell at C&A?

Principal Topics : A. (*anagram index / anagram*) C. (*cycle direct / permute*) i. (*index of*) a. (*alphabet*) ! (*factorial*), permutations, derangements, dihedral group, alternating group, parity.

A permutation shuffles items within a list. It can be described in two ways, either (1) by what is done in the course of the shuffle, or (2) as a display of its end result. Monadic `C.` transforms either of these descriptions into the other. Type (1) is always boxed so <7 3 6 4 is interpreted as "item 7 is replaced by item 3 which is replaced by item 4 which is replaced by item 7". The occurrence of 'item 7' at both the beginning and end of that description show that it is a **cycle**, and a general permutation is made up of several cycles which do not overlap. The result of carrying out such a series of actions is the **image** of the permutation, which unfortunately is also popularly referred to as a permutation. Denote by n the number of items available to be permuted (this is called the **order** of the permutation). If n=3, then <2 1 0 (type (1) definition) is a permutation whose image is 2 0 1 (type (2) definition). If n=4 the image of the same type (1) permutation is 2 0 1 3, that is the image depends on the order of a permutation. Images can be used as indexes to permute general objects using *from* ({), for example

```
   2 0 1{'ABC'
CAB
```

Permutations are in general <u>products</u> of disjoint cycles, and, as stated above monadic `C.` acts as a toggle between the two types of description.

```
   C.2 0;1 3
2 3 0 1
   C.2 3 0 1
┌───┬───┐
│2 0│3 1│
└───┴───┘
```

The one-to-one correspondence between cycles and images is guaranteed by making the highest index leftmost within each cycle and ordering the cycles in increasing value of leftmost elements. Thus the permutation 1 3;2 0 is identical to 2 0;3 1, although the latter is the preferred choice of representation.

The dyadic form of `C.` is called *permute*, and is very useful in shuffling items within lists as in

```
   ]ten=.(65+i.10){a.
ABCDEFGHIJ
   (4 3;7 1)C. ten
AHCEDFGBIJ
```

When `C.` is used monadically it is assumed that the order of the permutation is the maximum integer which appears explicitly in the argument

```
   C.4 3;7 1
0 7 2 4 3 5 6 1
   (C.4 3;7 1){ten
AHCEDFGB
```

Now consider the dyadic form of the verb `A.` which is called *anagram*. It allows all the permutation images of `i.3` can be listed in increasing order by

```
   (i.!3)A.i.3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

Replacing each image by its ordinal number reduces it to an integer from the set `i.!3` – think of this as an **atom**. If the order of n is large this can be a useful shorthand, so that for example `A.3 2 1 0` (the final permutation of order 4 in numerical sequence) is 23. Image numbering depends on the order n, so that the reverse process, that is moving from atimic number to image provides an obvious niche for dyadic `A.`:

```
   23 A.i.4
3 2 1 0
```

A more general way of looking at the dyadic form of `A.` is to view its left argument as an atomic number identifying a permutation to be applied to the smallest applicable sub-list at the right hand end of the right argument. For example `1 A.y` shuffles the last two items in a list, arguments 2 thru' 5 shuffle the final three items of permutations 6 thru' 23 shuffle the final four items and so on.

Permutations can be combined in the sense that 1 2 0 (A. number 3) applied to 1 0 2 (A.number 2) yields 0 2 1 (A. number 1).

```
   (3 A.t){2 A.t=.i.3
```

```
0 2 1
   A.(3 A.t){2 A.t=.i.3
1
```

Generalising this into a function and table :

```
   perm3=.dyad : 'A.(x A.t){y A.t=.i.3'
   3 perm3 2
1
   >perm3&>/~ i.6
0 1 2 3 4 5
1 0 3 2 5 4
2 4 0 5 1 3
3 5 1 4 0 2
4 2 5 0 3 1
5 3 4 1 2 0
```

This has the structure of a group table known as the dihedral group D3 (see E #26 “Working in Groups”) which is most commonaly observed in the symmetries of the equilateral triangle. 0, 3 and 4 correspond to rotations of 0, π/3 and 2π/3, and 1 2 and 5 to reflections in the perpendicular bisectors of the sides. Within D3 the subgroup formed by 0, 3 and 4 is the cyclic group C3.

A more general verb for combining permutations is

```
   perms=.dyad : 'A.(({.x) A.t){,(}.x)A.t=.i.y'
   11 5 perms 4
21
```

that is for permutations of order 4, the effect of permutation `A.11` following permutation `A.5` is permutation `A.21` (1 3 2 0 applied to 0 3 2 1 is 3 1 2 0 – see the table below.)

Inverses : getting back to where you started

The image permutation whose operation restores the natural numerical ordering of an image permutation p is given by /:p . So if the characters in ten appear in the order ICEDHFBAJG the instructions to restore them to natural order is

```
   C. /: ten i.'ICEDHFBAJG'
┌─┬─┬───────────────────┐
│3│5│9 8 0 7 4 2 1 6    │
└─┴─┴───────────────────┘
```

This operational permutation says : keep the fourth and sixth characters (i.e. D and F with indices 3 and 5) in place, the character in the

ninth position moves to the tenth, the character in the eighth position moves to the first and so on.

```
   C.C. /: ten i.t=.'ICEDHFBAJG'
7 6 1 3 2 5 9 4 0 8
   (C.C. /: ten i.t){t
ABCDEFGHIJ
```

A permutation consisting only of 2-cycles is self-inverse. For n=4 there are 10 of these, one containing no 2-cycles, 6 containing a single 2-cycle corresponding to the 6 ways in which a pair of items can be chosen from 4, and 3 containing two pairs of 2-cycles, viz. 3 2;1 0 , 3 1;2 0 and 3 0;2 1. There are 8 permutations with 3-cycles which form 4 mutually inverse pairs such as <3 0 2 and <3 2 0, and finally the number of 4-ccycles is 6, since a 4-cycle must have 3 as its first item and there are 6 ways in which the remaining 3 items can be permuted.

Permutations possess **parity**, that is they can be classified as odd or even according to whether they are obtainable as the result of an odd or an even number of successive two-item swaps from i.n. For example 0 1 2 3 -> 0 1 3 2 -> 3 1 0 2, so 3 0 1 2 is the image of an even permutation.

The table below classifies the permutations of order 4. In the properties column, O means Odd, S means self-inverse, and D indicates a derangement, that is a permutation in which every item has moved from its home position. The first 6 permutations have 0 in their first position, the next 6 have 1, and so on. However, apart from this labeling by A., number ordering does not reflect group structure. Subgroups of order 2 can be formed by combining 0 with any of the 9 non-identity self-inverse permutations. The following form subgroups of orders of orders 4, 8 and 12 :

{0,7,16,23} {0,2,21,23} (0,1,6,7} {0,5,14,16}
{0,7,16,23,6,1,17,22} {0,7,16,23,14,9,5,18]
{0,3,4,7,8,11,16,12,15,23,19,20}

of which the last is the set of even permutations, called the **alternating group.**

A .no	Image	Permutation	Properties	Inverse
0	0 1 2 3		S	
1	0 1 3 2	<3 2	O S	
2	0 2 1 3	<2 1	O S	
3	0 2 3 1	<3 1 2		4

```
 4  0 3 1 2  <3 2 1             3
 5  0 3 2 1  <3 1       O S
 6  1 0 2 3  <1 0       O S
 7  1 0 3 2  3 2;1 0      S D
 8  1 2 0 3  <2 0 1            12
 9  1 2 3 0  <3 0 1 2   O   D  18
10  1 3 0 2  <3 2 0 1   O   D  13
11  1 3 2 0  <3 0 1            19
12  2 0 1 3  <2 1 0             8
13  2 0 3 1  <3 1 0 2   O   D  10
14  2 1 0 3  <2 0       O S
15  2 1 3 0  <3 0 2            20
16  2 3 0 1  3 1;2 0      S D
17  2 3 1 0  <3 0 2 1   O   D  22
18  3 0 1 2  <3 2 1 0   O   D   9
19  3 0 2 1  <3 1 0            11
20  3 1 0 2  <3 2 0            15
21  3 1 2 0  <3 0       O S
22  3 2 0 1  <3 1 2 0   O   D  17
23  3 2 1 0  3 0;2 1      S D
```

Code Summary

For permutations described by their anagram numbers

```
perm3=.dyad : 'A.(x A.t){y A.t=.i.3'
            NB. application of 3-perms x then y
perms=.dyad : 'A.(({.x) A.t){,(}.x)A.t=.i.y'
            NB. application of y-perms 0{x then 1{x
```

31. A rippling good yarn

Principal Topics : /: (*grade up*) ^: (*power* conjunction) ripple shuffles

A 'ripple shuffle' of a deck of cards consists of dividing it into two halves (or as nearly as possible if there is an odd number). How many shuffles will it take for a deck of, say, 52 cards to return to its original state?

Suppose the cards are numbered consecutively from 0, and that there is an even number of cards. Then if all the even-numbered cards are placed in order on top of all the odd-numbered ones, a ripple shuffle would restore the original order. To put this in another way, a ripple shuffle is the inverse of 'all evens first' sort. Dyadic *grade up* performs by sorting its left argument to an order specified by its right argument, so if the latter is a list of alternating 0s and 1s of the same length as the former the effect is to obtain all the items of one parity followed by all the items of the other.

```
   irs=./: 0 1&($~)@#
   irs 'abcdef'
acebd
```

Use the *power* conjunction to do this repeatedly

```
   irs^:(i.7)i.10          NB. 7 inverse shuffles
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 1 3 5 7 9
0 4 8 3 7 2 6 1 5 9
0 8 7 6 5 4 3 2 1 9
0 7 5 3 1 8 6 4 2 9
0 5 1 6 2 7 3 8 4 9
0 1 2 3 4 5 6 7 8 9
```

Once the original order is restored, a succession of ripple shuffles is obtained by reading the rows of the above table from bottom to top. To count the number of shuffles required to restore the original order, the direction in which the table is read is immaterial, so define :

```
   countrs=.monad :0
r=.y [ i=.1
while. -.y-:irs r do.
  r=.irs r [ i=.>:i end. i
)
   countrs i.10
6
```

For a deck of 52 cards the number of ripple shuffles required to restore the original order is

```
   countrs i.52
8
```

A range of even card numbers can be explored some of whose results may be a little surprising at first sight :

```
   w,:>countrs each i. each w=.2+2*i.20
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
1 2 4 3  6 10 12  4  8 18  6 11 20 18 28  5 10 12 36 12
   w,:>countrs each i. each w=.22+2*i.20
22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60
 6 11 20 18 28  5 10 12 36 12 20 14 12 23 21  8 52 20 18 58
   w,:>countrs each i. each w=.64+4*i.15
64 68 72 76 80 84 88 92 96 100 104 108 112 116 120
 6 66 35 20 39 82 28 12 36  30  51 106  36  44  24
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
1 2 4 3  6 10 12  4  8 18  6 11 20 18 28  5 10 12 36 12
```

Intuitively it might be expected that numbers which are relatively rich in factors would have relatively short run lengths, however this is not in general the case. For example compare 60 with 92,

A variation on the ripple shuffle is to do a final exchange of the two half-decks by changing 0 1 to 1 0 in the verb `irs` and changing `irs` correspondingly in `countrs`. The results for the first few values are

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
2 4 3 6 10 12  4  8 18  6 11 20 18 28  5 10 12 36 12 20
```

The row above is the same as that of the corresponding row in the previous table, only shifted one place to the left. Thus if rs(n) is the run length for a pack of n cards without final half-deck exchange, and rss(n) is the run length with exchange, then rss(n)=rs(n+ 2).

Apart from the powers of two for which rs(n) is simply $\log_2 n$, the pattern of rs(n) is an irregular one with some abrupt changes. In general low values of rs(n) and rss(n) go together, although this is not always the case, e.g. rs(54)=rss(52)=52, that is the effect of the seemingly final switch of the two half-decks for a 52-card deck is that 52 runs rather than 8 are required for restoration.

Another way to address the problem of order-restoring shuffle run lengths is to observe the position of, say, the number 1 in successive shuffles in the table above, remembering to read it upwards!. Its successive positions are 1, 2, 4, 8, then it wraps round to position 7 which

= 16 in modulo 9 arithmetic. Then look at the progress of the number 2, which moves to position 4, 8, 7, 5, or 4,8,16,32 in modulo 9 arithmetic. Thus the numbers 1 and 2 will be restored in the same number of shuffles, and similarly for all the other numbers.

The problem of counting the number of shuffles needed to restore the original order is thus equivalent to that of obtaining a second 1 in the sequence `2^i.10` in modulo 9 arithmetic, thus

```
   9|2^i.13
1 2 4 8 7 5 1 2 4 8 7 5 1
```

shows the 6-cycle already observed with 10 cards. Since the number of shuffles to restore in general cannot exceed the number of cards, the number of shuffles comes from observing 1s in `(n-1)|2&^@>:@i.(n+1)`, leading to

```
   (<:|2&^@>:@i.)10
2 4 8 7 5 1 2 4 8 7
```

following which the number of shuffles for various cases is given by

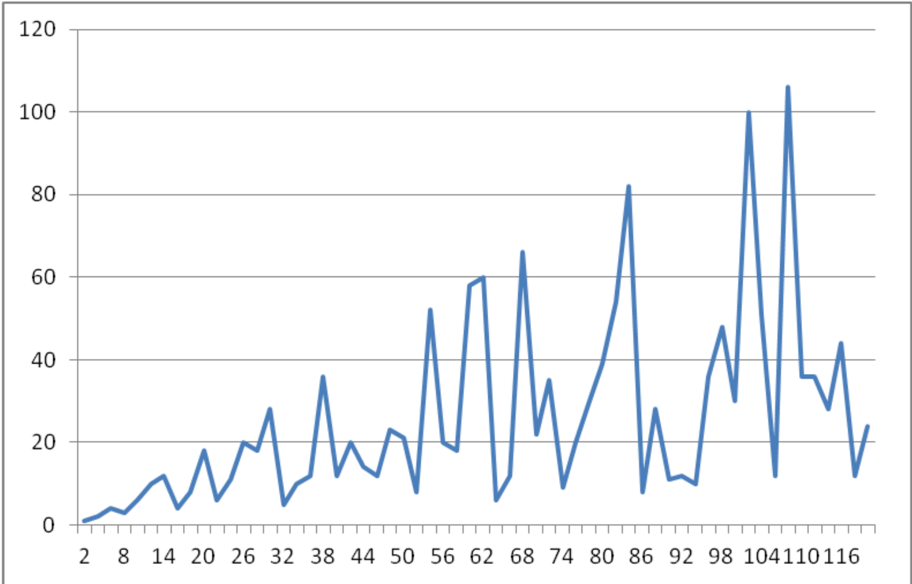
```
   v=.2&^@>:@i.
   count=.>:@i.&1@(<:|v)
   count 10
6
   w,:>count each w=.12+2*i.15
12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
10 12  4  8 18  6 11 20 18 28  5 10 12 36 12
   w,:>count each w=.44+4*i.15
44 48 52 56 60 64 68 72 76 80 84 88 92 96 100
14 23  8 20 61  6 69 35 20 39 85 28 12 36  30
```

A few of these values such as that for 60 are inconsistent with the previous table. This is because 2^{60} is rather a large number which requires extended precision for exact integer comparison. Thus in the event of discrepancy it is `counts` which delivers the correct values

To deal with the variation in which the two half-decks are exchanged at the end of the shuffle, simply replace the <: in the middle with >:

```
   counts=.>:@i.&1@(>:|v)
   counts 10
10
```

Here is a graph of rs(n) from n=2 to n=120



As noted earlier, the powers of 2 are easy to deal with, they are simply $\log_2 n$. But why do 22, 52, 74 an 92 have short cycles whereas 20,30,50 and 60 have long ones? And why are there regions of relatively short run lengths such as between 86 and 100? Any suggestions?

Code Summary

```
  irs=./: 0 1&($~)@#       NB.sorts list, odd-indexed items,
then even

  countrs=.monad :0 NB.exact count of sorts to restoration
r=.y [ i=.1
while. -.y-:irs r do.
  r=.irs r [ i=.>:i end. i
)
count=.>:@i.&1@(<:|v)      NB. count to restoration for n<60
   v=.2&^@>:@i.            NB. powers of 2 from 1 to y
counts=.>:@i.&1@(>:|v)     NB. ditto with half-list switching
```

32. So Easy a Child of 10…

Principal Topics : +. (*GCD)* /: (*grade up*) e. (*membership*) ^: (*power* conjunction) , -. (*not*= *1-)* *. *(LCM*) ~. (*nub)* q: (*prime factors*) C. (*cycle-direct*) ` (*gerund),* @. (*agenda)* ripple shuffle, clock (finite) arithmetic, totient, Euler's phi, tau, sigma, Fermat's little theorem, primeness factorisation, primitive roots.

Perfect shuffles just won't go away! Following E #31 ("A Rippling Good Yarn") on this subject, Gene MacDonnell, Roger Hui and Jeff Shallit made insightful comments which help cast the problem in a broader context. I shall endeavour to summarise their thoughts here, sauced with generous helpings of J!

To recap, a perfect or ripple shuffle of a deck of cards consists of dividing it into two halves (or as nearly as possible if there is an odd number of cards) and taking one card in turn from each half. A single shuffle of a given number of cards is given by

```
   sh=./:@$&0 1        NB. ripple shuffle of i.y
```

or for repeated shuffling, make the argument into a list (not necessarily numeric)

```
   rs=./:0 1&($~)@#     NB. ripple shuffle of y
```

Assume in what follows that both the word 'number'and the letters m, n and k denote 'a positive integer', while the letter p means 'a prime number' (including 1). A result called Fermat's Little Theorem, first formally proved by Euler in 1736, states that if (n,p) are relatively prime, then n^{p-1}=1 in modulo p arithmetic. **Relatively prime** says in words what GCD(m,n)=1 says in maths, or `1=m+.n` says in J, as in the verb:

```
   rps=.i.#~(e.&1(+.i.)) NB. relative primes of y
   rps 15
1 2 4 7 8 11 13 14
```

Modulo n arithmetic is what primary school children are familiar with as **clock arithmetic**, that is the arithmetic of a finite set of numbers `i.n` equally spaced around the rim of a clock. A J session can be set up to perform modulo n arithmetic by setting the modulus and defining an adverb such as `mod` :

```
   n=.7
   mod=.adverb : 'n&|@x'
   (6+mod 3),(*:mod 9)  NB. (9 mod 7),(9^2 mod 7)
2 4
```

Advancing a little (but only a little!) beyond primary school, every number possesses a **totient**, where tot(n) is the number of relatively prime numbers which are less than n. Thus tot(2) is 1, tot(3) and tot(4) are both 2 (the relatively prime number lists being 1,2 and 1,3 respectively), tot(5)=4 (all lower numbers) and so on. tot(n) is often written

$\phi(n)$, and called **Euler's phi**, or in J, `#@rps`. Were this mathematical function just a little more useful, it might well have found a place on calculator keyboards, or indeed as a J primitive, along with factorial, log, sin, etc., and the like.

However, it is not necessary to enumerate relatively prime numbers to find tot(n) since it is given by the closed formula

$$\phi(n) = n(1\text{-}1/p1)\ldots(1\text{-}1/pn)$$

where the p's are the unique prime factors of n

Totient can thus be regarded as an extension of `q:` which gives the **prime factorisation** of n:

```
   tot=.*/@,(-.@%)@(~.@q:) NB. totient (Euler's phi)
   (tot 10),(tot 51)
4 32
```

Neither set of parentheses is necessary in the above definition of `tot`, but they help to clarify how it works. `(-.@%)n` is 1 - 1/n, `(~.&.q:)` is the prime factor *nub*, and the comma makes the hook which multiplies in the factor n.

Euler generalised Fermat's Little Theorem to non-primes by proving that, provided m and n are relatively prime, $m^{tot(n)} = 1$ (mod n). For primes, all preceding numbers are relatively prime, so tot(p) = p-1 and Euler's and Fermat's theorems are equivalent in this case. Some other properties of the totient are simple to prove, viz.

tot(n) is even for all n>2 (this follows from the closed formula)
$tot(2^k) = 2^{k-1}$ (because every odd number less than 2^k is relatively prime)
tot(mn) = tot(m).tot(n) if m,n are relatively prime; and
= n.tot(m) when the prime factors of n are a subset of those of m.

In particular $tot(n^2) = n.tot(n)$. The third of the above properties can be described by saying that tot is a multiplicative function. Generalising the result to prime factor products, if $n=(p_1^{k1})(p_2^{k2})\ldots(p_v^{kv})$ then

$$tot\ (n) = */\ tot\ (p_1^{k1}),\ tot\ (p_2^{k2}),\ \ldots\ ,tot\ (p_v^{kv})$$

As an aside, the functions tau(n) and sigma(n) as defined below are also multiplicative functions.

```
   seldivs=.0&=@|~i.        NB. select divisors of y
   divs=.seldivs~#i.        NB. divisors of y excl y
   divs 12
1 2 3 4 6
   tau=.#@,divs             NB. tau=no. of divisors incl y
   sigma=.+/@,divs          NB. sigma=sum of divisors incl y
   (tau every 12 13 156);(sigma every 12 13 156)
┌──────┬─────────┐
│6 2 12│28 14 392│
└──────┴─────────┘
```

To illustrate the sort of possible uses for tot(n) and modulo n arithmetic, suppose that the last two digits of 3^{256} (which incidentally has 123 digits altogether) are required. tot(100) = 40 so the problem reduces to that of finding the last two digits of 3^{16} by e.g.

$(3^{16}) = (81)^4 = (-19)^4 = (361)^2 = 612 = 3721 = 21$

As a further aside, it is not hard to prove that tot(2n) = tot(n) if n is odd and equals 2.tot(n) if n is even, a result which it is pleasing to have J confirm by comparing matching columns in

```
   (5 6$tot every >:i.30);5 6$tot every 2*>:i.30
┌───────────────┬─────────────────┐
│ 1  1  2  2  4 2│ 1  2  2  4  4  4│
│ 6  4  6  4 10 4│ 6  8  6  8 10  8│
│12  6  8  8 16 6│12 12  8 16 16 12│
│18  8 12 10 22 8│18 16 12 20 22 16│
│20 12 18 12 28 8│20 24 18 24 28 16│
└───────────────┴─────────────────┘
```

As well as <u>confirming</u> results, J can also <u>suggest</u> results ahead of proof. For example, the result that tot(3n) = 3tot(n) for multiples of 3, and equals 2tot(n) otherwise is forecast with clarity by

```
   (5 6$tot every >:i.30);5 6$tot every 3*>:i.30
┌───────────────┬─────────────────┐
│ 1  1  2  2  4 2│ 2  2  6  4  8  6│
│ 6  4  6  4 10 4│12  8 18  8 20 12│
│12  6  8  8 16 6│24 12 24 16 32 18│
│18  8 12 10 22 8│36 16 36 20 44 24│
│20 12 18 12 28 8│40 24 54 24 56 24│
└───────────────┴─────────────────┘
```

Returning to the ripple shuffle problem, the number of shuffles required to restore an even numbered deck of n cards to its original order is the number of times 2 must be multiplied in modulo n-1 arithmetic in order to obtain 1. To obtain such a value, one way is simply to carry on multiplying and reducing modulo (n-1) until 1 is reached, an event which Euler's theorem guarantees is bound to happen. However there may be an earlier arrival at the target than that predicted by Euler's Theorem. For example tot(51) = 32, so that 32 shuffles will restore 52 cards to their original order.

However, if 2 is doubled repeatedly (note a good excuse for a gerund!) :

```
   n=.51                               NB. set modulus
   p2=.,$:@(+:mod@{:)`}.@.(1&e.)       NB. powers of 2
   p2 2
2 4 8 16 32 13 26 1
```

It transpires that a mere 8 steps are sufficient. 8 is called the multiplicative order of 2 (mo2 for short) in modulo 51 arithmetic, and Euler's theorem guarantees that mo2(n) is a divisor of tot(n), which is helpful in manual searches. mo2 is of course just `#p2` . As an alternative to redefining `p2` every time the modulus is reset, write

```
   mo2=.monad :0   NB. mult order of 2 for odd modulus y
r=.2
while.(1~:y.|r)do.r=.x:2*r end. [ 2^.r
)
   (mo2 13),(mo2 51)
12 8
```

Now revisit the ripple shuffle with an even number of cards, for example

```
   sh 10
0 2 4 6 8 1 3 5 7 9
```

It takes only a moment to see that 0 and 9 will remain in place in repeated shuffles, and that the second position will be occupied by successive powers of 2 in modulo 9 arithmetic. The number of shuffles to restore a pack with an even number of cards n is thus mo2(n-1).

Gene pointed out that another way to regard a shuffle such as `sh 10` is as a permutation of `i.10`, which can be expressed using `C.` as a combination of cycles:

```
   C. sh 10
┌─┬───┬───────────┬─┐
│0│6 3│8 7 5 1 2 4│9│
└─┴───┴───────────┴─┘
```

and if shuffling is continued until the original order is restored, the cycles emerge in the columns of these lists read as a matrix :

```
   rs^:(i.6)i.10    NB. all distinct shuffles of 10
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 1 3 5 7 9
0 4 8 3 7 2 6 1 5 9
0 8 7 6 5 4 3 2 1 9
0 7 5 3 1 8 6 4 2 9
0 5 1 6 2 7 3 8 4 9
```

This demonstrates clearly that if 1 is restored to its original position all the other numbers will obediently follow suit. Since <u>all</u> the cycles must return to the start point, the LCM of the lengths of the individual cycles determines the number of perfect shuffles to restore a deck of n cards :

```
   cyclecnt=.(#every)@(C.@sh)
   cyclecnt 10
1 2 6 1
   ns=.*./@:cyclecnt   NB. no. of restoring shuffles
   ns 52
8
```

If n is odd, `C.sh n` is the same as `C.sh n+1` only without the final one-element box:

```
   C. sh 9
┌─┬───┬───────────┐
│0│6 3│8 7 5 1 2 4│
└─┴───┴───────────┘
```

Thus ns(n) and ns(n-1) are identical in value to mo2(n-1) so that ns(n) is defined for <u>all</u> integers. The LCM of the `cyclecnt` of a product mn is the LCM of the `cyclecnts` of m and n separately, subject to GCD(m,n)=1. For example:

```
   cyclecnt EVERY 11 13 143
┌────┬────┬───────────────┐
│1 10│1 12│1 10 12 60 60  │
└────┴────┴───────────────┘
   ns every 11 13 143          NB. LCM(10,12)=60
10 12 60
```

Generalising the LCM property

$$mo2(n) = *./\ mo2(p_1^{k1}), mo2(p_2^{k2}),\ \ldots\ , mo2(p_v^{kv})$$

is identical in form to the analogous expression for tot above, only with `*.` (that is LCM) replacing `*` (multiply). The relationship between the notions of multiply and LCM is emphasised by the closeness of the <u>notation</u> in J. `mo2` of course is not a multiplicative function – perhaps it should be called an LCM-ic function!

Multiplicative order is a property of all relatively prime numbers less than the modulus. `mo10(n)`, where `mo10` is defined analogously to `mo2`, gives the period length of the recurrence in the decimal representation of `%n`, for example :

```
   (mo10 13),%13
6 0.076923076923
```

For shuffles where every third card is picked ns3 counts the number of shuffles to restore:

```
   sh3=./:@$&0 1 2            NB. shuffle with every 3rd card
   sh3 10
0 3 6 9 1 4 7 2 5 8
   C.sh3 10
┌─┬─────┬───────────┐
│0│7 2 6│9 8 5 4 1 3│
└─┴─────┴───────────┘

   cc3=.(#every)@C.@sh3
   ns3=.*./@:cc3              NB. #shuffles to restore
   ns3 every 10 11 12             NB. .. with 10,11 & 12 cards
6 5 5
```

Analogously with ns, ns3(3n) is identical in value to ns3(3n-1), as shown by :

```
   C.sh3 12
```

0	9 5 4 1 3	10 8 2 6 7	11

```
   C.sh3 11
```

0	9 5 4 1 3	10 8 2 6 7

although unlike ns, values of ns3(n) no longer coincide with those of mo3(n).

The above procedure can be extended to shuffles with picking at any regular interval, and all the previous discussion on shuffles can be condensed into

```
   shn=./:@$ i.                    NB.pick each yth out of x
   nsn=.*./@:(#each)@C.@shn        NB. #shuffles to restore
   (51 nsn 2),(10 nsn 3)
8 6
```

Multiplicative orders are a more general property than shuffle counts. Here is a table of totients and the first three multiplicative orders of the first few integers:

n:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
tot:	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8
mo2		2	_	4	_	3	_	6	_	10	_	12	_	4	_
mo3			2	4	_	6	2	_	4	5	_	3	6	_	4
mo5				_	2	6	2	6	_	5	2	4	6	_	4
tot²	1	1	1	2	1	2	2*	2	2	4	2*	4	2	4*	4*

n:	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
tot	16	6	18	8	12	10	22	8	20	12	18	12	28	8	30	16
mo2	8	_	18	_	6	_	11	_	20	_	18	_	28	_	5	_
mo3	16	_	18	4	_	5	11	_	20	3	_	6	28	_	30	8
mo5	16	6	9	_	6	5	22	2	_	4	18	6	14	_	3	8
tot²	8	2	6	4*	4*	4	10	4*	8	4	6	4*	12	4*	8	8*

If mo2(n) is equal in value to tot(n) it is called a **primitive root** of n. Roughly speaking, powers of primitive roots exhaust the full gamut of modulo n integers before repeating. Looking at the second and third rows in the table, 2 is a primitive root of some numbers such as 9 and 13, but not of others such as 7 and 17. The table also shows that 3 and 5 are primitive roots of 7. The final row is the totient of the totient which in the case of primes, is also the number of primitive roots. This is also the case for those non-primes such as 9 which possess primi-

tive roots, other non-primes such as 15 have no primitive roots, and are marked with an asterisk in the final row. There is no general formula for primitive roots, but for small numbers such as those given in the table, they are not hard to find, particularly if a computer with J is at hand. For example, 2 is a primitive root of 13 from the table, and the other three are to be found to be 6, 7 and 11 by observing that

```
   6^mod divs tot n=.13   NB. powers of 6 modulo 13
6 10 8 9 12
```

does not contain 1, and similarly for 7 and 11. Alternatively use lists to test all the candidate numbers simultaneously:

```
   (<>:i.12)^mod every >divs tot n=.13
                                 NB. primitive roots of 13
1  2 3  4  5  6  7  8 9 10 11 12
1  4 9  3 12 10 10 12 3  9  4  1
1  8 1 12  8  8  5  5 1 12  5 12
1  3 3  9  1  9  9  1 9  3  3  1
1 12 1  1 12 12 12 12 1  1 12  1
```

With a little more code all the primitive roots of primes can be extracted in one go:

```
   n=.13
   t#~-.1 e."1 |:(<t=.rps n)^mod every divs tot n
2 6 7 11
   n=.15
   t#~-.1 e."1 |:(<t=.rps n)^mod every divs tot n
(null list)
```

Although this discussion has led into the beginnings of number theory on the one hand and combinatoric analysis on the other, nevertheless a primary school child with outstanding numerical gifts could well appreciate all the notions in this article, if not perhaps the notations, and could, with at most the aid of a hand calculator, compute the above table of totients and multiplicative orders. Perhaps it is not a coincidence that the abbreviated form is tot(n)!

Code Summary

```
   mod=.adverb : 'n&|@x'
   sh=./:@$&0 1             NB. ripple shuffle of i.y
   rs=./:0 1&($~)@#         NB. ripple shuffle of y
   rps=.i.#~(e.&1(+.i.))    NB. relative primes of y
   tot=.*/@,(-.@%)@(~.@q:)  NB. totient (Euler's phi)
   tau=.#@,divs             NB. tau=no. of divisors incl y
   sigma=.+/@,divs          NB. sigma=sum of divisors incl y
     divs=.seldivs~#i.      NB. divisors of y excl y
       seldivs=.0&=@|~i.   NB. select divisors of y

   p2=.,$:@(+:mod@{:)`}.@.(1&e.)  NB. powers of 2

   mo2=.monad :0    NB. mult order of 2 for odd modulus y
r=.2
```

```
while.(1~:y|r)do.r=.x:2*r end. [ 2^.r
)
   ns=.*./@:cyclecnt         NB. no. of restoring shuffles
      cyclecnt=.(#every)@(C.@sh)
   ns3=.*./@:cc3              NB. #shuffles to restore
     cc3=.(#&>)@C.@sh3
     sh3=./:@$&0 1 2          NB. shuffle with every 3rd card
  nsn=.*./@:(#every)@C.@shn   NB. #shuffles to restore
    shn=./:@$ i.              NB. pick every yth out of x
```

33. Perming and Combing

Principal Topics : /: (*grade up*) A. (*anagram index*) {. (*take*) }: (*curtail*) }. (*drop*) ? (*deal*) |: (*transpose*), ` (*gerund*), @. (*agenda)* permutation, permutation list, Lehmer's algorithm, factorial digits, recursion, lexical ordering, Tompkins-Page ordering, parity, Johnson ordering, combination lists.

No, not an accidentally misdirected submission to "The Hair Stylist", but rather an extension of one of Gene McDonnell's token reduction examples (see Vector vol. 22 no.3) which involves generating systematic lists of permutations, where a permutation of order `n` is to be understood as a list of integers `i.n` in some order, for example `3 0 2 1 4` .

Permutation lists

A permutation list is a list of lists in which all of the `!n` possible permutations occur once and once only. The most interesting lists of this kind are those in which the ordering is in some way systematic, the most obvious being the lexical order which Gene discusses, and which is readily obtainable using `A.`(*anagram index*) as shown below. This ordering is also the result of Lehmer's algorithm, so `Llist` can be taken to denote either lexical or Lehmer.

```
   Llist=: i.@! A. i.          NB. Lehmer/lexical listing

   |:Llist 4
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
1 1 2 2 3 3 0 0 2 2 3 3 0 0 1 1 3 3 0 0 1 1 2 2
2 3 1 3 1 2 2 3 0 3 0 2 1 3 0 3 0 1 1 2 0 2 0 1
3 2 3 1 2 1 3 2 3 0 2 0 3 1 3 0 1 0 2 1 2 0 1 0
```

(Note : In many of the illustrations which follow, transposition appears as the leftmost verb in the J line for the purpose of compactness of display. In these cases individual permutations should thus be read as columns reading from top to bottom.)

Factorial Digit bases

Gene defines a factorial digit base (that is, number base in the J sense) as :

```
  fdb=.}:@(>:@i.@-)   NB. factorial digit base
  fdb 4
4 3 2
```

Each digit from `0` to `n-1` has a unique representation in this base, as shown by :

```
   fact=.fdb#:i.@!
   |:fact 4            NB. 0 to 23 as factorial dig-
its
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

A permutation can be transformed to corresponding factorial digits by taking its items from the left and, for all but the last, recording the number of smaller digits which appear to its right:

```
   nld=.+/@:({. > }.)       NB. number of lesser
digits
   ptof=.}.`(nld,ptof@}.)@.(1&<@#)
                            NB. perm -> fact digits
   ptof 3 0 2 1
3 0 1

   |:ptof every <"1 Llist 4
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

The inverse of `ptof` is also obtainable recursively, this time by joining each factorial digit to the immediately prior permutation list, copying any lesser digits and adding one to the others. To those who know J, this is probably described more clearly in J than in English!

```
   incgte=.] + <:      NB. increment if gtr or eq,
                       NB. else copy
   ftop=.(, -.)`({. , {. incgte ftop@}.) @.(1&<@#)
                       NB. fact digits -> perm
   |:ftop every <"1 fact 4
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
1 1 2 2 3 3 0 0 2 2 3 3 0 0 1 1 3 3 0 0 1 1 2 2
2 3 1 3 1 2 2 3 0 3 0 2 1 3 0 3 0 1 1 2 0 2 0 1
3 2 3 1 2 1 3 2 3 0 2 0 3 1 3 0 1 0 2 1 2 0 1 0
```

The above array demonstrates that ordering by factorial digits is equivalent to lexical ordering.

Factorial digits used with `ftop` resemble the left arguments of `b.` in as much as each digit in the appropriate range represents a function.

For example, a factorial digit of integers descending to 1 such as `3 2 1` represents *rotate* (`|.`), a string of all 1s represents *1-shift*, a string of 2s followed by a 0 represents *2-shift*, a string of 3s followed by two 0s represents *3-shfit*, and so on.

Other systematic orderings of permutations

Another systematic ordering for permutation lists involves using a recursive method to generate what is known as the Tompkins-Paige ordering, first published in 1960 :

```
   TPlist=.monad : 0
if.y>1 do.
  r=.,/(i.y)|."1 every<(TPlist<:y),.<:y
else. r=.1 $0 end.
)
  |:TPlist 4
0 1 1 0 2 2 1 0 2 2 0 1 2 2 0 1 1 0 3 3 3 3 3 3
1 0 2 2 0 1 2 2 0 1 1 0 3 3 3 3 3 3 0 1 1 0 2 2
2 2 0 1 1 0 3 3 3 3 3 3 0 1 1 0 2 2 1 0 2 2 0 1
3 3 3 3 3 3 0 1 1 0 2 2 1 0 2 2 0 1 2 2 0 1 1 0
```

In the above array, focussing on how the `3`s are blended with repetitions of the next lower order permutations list is the easiest way to see how the Tompkins-Paige ordering is formed. In time efficiency terms `TPlist` and `Llist` are broadly similar.

There are obvious ways in which orderings such as `Llist` and `TPlist` can be used to generate further systematic orderings, for example

```
  |:3 - TPlist 4
3 2 2 3 1 1 2 3 1 1 3 2 1 1 3 2 2 3 0 0 0 0 0 0
2 3 1 1 3 2 1 1 3 2 2 3 0 0 0 0 0 0 3 2 2 3 1 1
1 1 3 2 2 3 0 0 0 0 0 0 3 2 2 3 1 1 2 3 1 1 3 2
0 0 0 0 0 0 3 2 2 3 1 1 2 3 1 1 3 2 1 1 3 2 2 3
```

Another means of generating systematic orderings is to use `TPlist n` as an index to any one of its component permutations, for example :

```
   |:(TPlist 3){1 0 2
1 0 0 1 2 2
0 1 2 2 1 0
2 2 1 0 0 1
```

More generally, the component permutation can be randomised as in the following tacit verb train :

```
   rlist=.TPlist { ?@! { Llist   NB. list based on
random perm
   |:rlist 4
2 3 3 2 0 0 3 2 0 0 2 3 0 0 2 3 3 2 1 1 1 1 1 1
3 2 0 0 2 3 0 0 2 3 3 2 1 1 1 1 1 1 2 3 3 2 0 0
0 0 2 3 3 2 1 1 1 1 1 1 2 3 3 2 0 0 3 2 0 0 2 3
1 1 1 1 1 1 2 3 3 2 0 0 3 2 0 0 2 3 0 0 2 3 3 2
```

Parity of permutations

A basic property of permutations is that of **parity**, which means the oddness or evenness of the number of switches required to bring the permutation back to `i.n` . This property is related to factorial digit representations in the following way :

```
   parity=.2&|@+/@ptof                   NB. rem((sum
of fact digs),2)
   parity every<"1 TPlist 3
0 1 0 1 0 1
   parity every<"1 TPlist 4
0 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0
```

Parity can also be obtained by considering the cycle form of a permutation as given by `C.` which returns a list of boxed lists (cycles) whose lengths are given by `#eachC.` :

```
   C.t=.6 4 2 7 1 3 0 5
```

2	4 1	6 0	7 5 3

A cycle of even length has odd parity and a cycle of odd length has even parity so

```
ui
  #every C.t       NB. cycle parities are E O O E
1 2 2 3
```

Parities combine according to the 'not-equals' rule, so in the above case the overall parity is

```
   par=.~:/@:-.@(2&|)@(#every@C.)     NB. by cycles
   par t
0
```

Minimising switches between adjacent permutations

For `rlist 3`, the permutation list parity pattern is either `0 1 0 1 0 1` or `1 0 1 0 1 0` which means that parity alternates between successive permutations. For `rlist 4` the parity pattern is always one of

```
0 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0
```
or
```
1 0 1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1
```

where in both cases, there are some consecutive permutations with the same parity. In programs which loop extensively through permutations computed in advance, the efficiency gained by minimising the changes between successive permutations can sometimes be worth considering. In 1963 S.M. Johnson of the RAND Corporation showed how to achieve such an ordering. A recursive technique is based on taking a single permutation at the next lower level, and creating `n+1` further permutations by inserting `n` into all possible gaps starting either from the right or the left :

```
0 1 2 3                3 0 1 2
0 1 3 2                0 3 1 2
0 3 1 2                0 1 3 2
3 0 1 2                0 1 2 3
```

In array terms, the incoming value `n` is stitched to a skewed array of multiple copies of each of the next lower order permutations followed by a reverse skew which brings `n` onto the diagonal :

```
   rskew=:i.@-@# |.&> <"1        NB. right skew
   lskew=:i.@# |.&> <"1          NB. left skew
   mcopy=:$~ (,~ >:)@#           NB. multiple copies
   agr=:lskew@(rskew@mcopy@[ ,. ])  NB. start in-
serts from right
   agl=:rskew@(lskew@mcopy@[ ,. ])  NB. start in-
serts from left
```

Now choose between right and left on the basis of parity :

```
   ag=.agr`agl@.(parity@[)       NB. all gaps (right
or left)
```

and finally bring everything together in

```
Jlist=:monad : 0
if.y~:2 do.
   r=.,/(<"1 Jlist <:y)ag every <:y
else. r=.0 1,:1 0 end.
)
   |:Jlist 4
0 0 0 3 3 0 0 0 2 2 2 3 3 2 2 2 1 1 1 3 3 1 1 1
1 1 3 0 0 3 2 2 0 0 3 2 2 3 1 1 2 2 3 1 1 3 0 0
2 3 1 1 2 2 3 1 1 3 0 0 1 1 3 0 0 3 2 2 0 0 3 2
3 2 2 2 1 1 1 3 3 1 1 1 0 0 0 3 3 0 0 0 2 2 2 3
```

This ordering not only involves just one switch between consecutive permutations, but additionally all such switches are between immediately neighbouring elements. However, the time to compute a `Jlist` is orders of magnitude greater than that required for a `TPlist` or an `Llist`, and so as, noted above, this method only pays off when there is a considerable amount of looping through pre-computed permutation lists.

Combinations and permutations of r from n

To find ordered permutations of the n!/(n-r)! permutations of r items from n, make every possible selection (that is combination) of `r` elements from `i.n`, and then transform each of these into a list of its `!r` possible permutations using any of the permutation list methods. Alternatively use the following :

```
   perms=.4 :'~.x{.&.|: Llist y'
   |:2 perms 5                          NB. permutations of 2 
items from 5
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
1 2 3 4 0 2 3 4 0 1 3 4 0 1 2 4 0 1 2 3
```

Obtaining systematic combination lists is not quite as tidy a matter as finding permutation lists. One possible technique is to express `i.n` as binary numbers and select those whose digit sum is `r` :

```
   cnos=.i.@:(2&^)@]       NB. integers from 0 to 
2^n -1
   bins=.#:@cnos      NB. binary nos from 0 to 2^n-1
   mark=.|.@((= +/"1@bins) # cnos) { bins
                      NB. marks for bins with dDig-
itsum = x
   combs=.mark #"1 i.@]    NB. transform to combns
s
```

```
   |:2 combs 4
0 0 0 1 1 2
1 2 3 2 3 3
```

Starting with all combinations here is yet another way of obtaining a permutation list of r items from ::

```
   perm=.,/@(<@TPlist@[ {every<"1@combs)    NB. r
from n
   |:2 perm 4
0 1 0 2 0 3 1 2 1 3 2 3
1 0 2 0 3 0 2 1 3 1 3 2
```

An alternative method of obtaining combinations of r from n

The following technique is another way of obtaining the defining binary array. `mk` is `mark` with its component binary number lists in a different order. The method is based on the fact that the combinations of r from n consist of all the combinations of r from n-1, together with n added to all the combinations of r-1 out of n-1. In binary number terms :

r `mk` n is (r `mk` (n-1) *stitch*ed with 0) joined to ((r-1) `mk` (n-1) *stitched* with 1) .

For largish values the double recursion makes this method less appealing without adaptations which could obscure the fundamental principle. The double recursion also leads to the two stopping values embodied in the first two lines :

```
   mk=.dyad : 0
if. x=y do. r=.(1,y)$1 return. end.
if. x=1  do.r=.=i.y return. end.
r=.((x mk<:y),.0),((<:x)mk<:y),.1
)
   coms=.mk #"1 i.@]
   |:4 coms 6
0 0 0 0 1 0 0 0 1 0 0 1 0 1 2
1 1 1 2 2 1 1 2 2 1 2 2 3 3 3
2 2 3 3 3 2 3 3 3 4 4 4 4 4 4
3 4 4 4 4 5 5 5 5 5 5 5 5 5 5
```

… and finally a Bang !!

Anyone who has delved into such matters must know the phrase 'combinatorial explosion' which describes how soon space and time boundaries are exceeded for even quite modest values of the parameters. This article has outlined the basic arithmetic of permutations and combinations en masse for whose exposition J is particularly suitable. Skilful tinkering and use of parallel processing can extend these boundaries, usually at an understandable loss of some clarity – that's where programming ingenuity kicks in …

Code Summary

```
   Llist=: i.@! A. i.      NB. Lehmer/lexical listing
   fdb=.}:@(>:@i.@-)       NB. factorial digit base
   fact=.fdb#:i.@!
   ptof=.}.`(nld,ptof@}.)@.(1&<@#)      NB. perm -> fact digits
   nld=.+/@:({. > }.)      NB. number of lesser digits

   ftop=.(, -.)`({. , {. incgte ftop@}.) @.(1&<@#)
                           NB. fact digits -> perm
      incgte=.] + <:       NB. increment if gtr or eq, else
copy

   TPlist=.monad : 0       NB. Tompkins-Paige ordering of perms
if.y>1 do.
  r=.,/(i.y)|."1 every<(TPlist<:y),.<:y
else. r=.1 $0 end.
)
   rlist=.TPlist { ?@! { Llist   NB. list based on random perm

   combs=.mark #"1 i.@]    NB. transform to combns s
      mark=.|.@((= +/"1@bins) # cnos) { bins
                           NB. marks for bins with dDigitsum =
x
      bins=.#:@cnos        NB. binary nos from 0 to 2^n-1
      cnos=.i.@:(2&^)@]    NB. integers from 0 to 2^n -1
)
parity=.2&|@+/@ptof
par=.~:/@:-.@(2&|)@(#every@C.)   NB. by cycles

Jlist=:monad : 0           NB. Johnson ordering (minimizes
switches)
if.y~:2 do.
   r=.,/(<"1 Jlist <:y)ag every <:y
else. r=.0 1,:1 0 end.
)
  ag=.agr`agl@.(parity@[)             NB. all gaps (right or
left)
    agr=:lskew@(rskew@mcopy@[ ,. ])     NB. inserts from right
    agl=:rskew@(lskew@mcopy@[ ,. ])     NB. inserts from left
    rskew=:i.@-@# |.&> <"1              NB. right skew
    lskew=:i.@# |.&> <"1                NB. left skew
    mcopy=:$~ (,~ >:)@#                 NB. multiple copies

   perms=.4 :'~.x{.&.|: Llist y'     NB. permuttions of r
from n
```

```
   coms=.mk #"1 i.@]                    NB. combinations of r
from n
mk=.dyad : 0
if. x=y do. r=.(1,y)$1 return. end.
if. x=1  do.r=.=i.y return. end.
r=.((x mk<:y),.0),((<:x)mk<:y),.1
)
```

34. Combination Lists

Principal Topics : A. (*anagram index / anagram*), ! [(*left*)] (*right*) \ (*prefix / infix*), \. (*suffix / outfix*), ; (*raze*) ~ (*reflex*) !: (*foreign conjunction*), trace, time

Like R.E.Boss (see Vector Vol.24 Nos. 2 &3, pp. 75-88 "Generating Combinations in J efficiently on lists of combinations") - because, like him, I have been fascinated both by their patterns and by algorithms which generate them. In both APL and J systematic permutation lists are easier to generate than those for combinations. In J the former are available directly through the primitive `A.` so that

```
  A.1 2 0
3
```

says that `1 2 0` is permutation 3 (in index origin 0) of `i.3` in lexical order, a process which is reversed by dyadic `A.` :

```
   3 A. 0 1 2
1 2 0
```

A full list of such permutations is given by e.g.

```
   (i.@! A. i.)3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

Analogously with monadic `A.` any combination of r integers from `i.n` can be put into one-to-one correspondence with the counting integers by adding appropriate values of ${}^{k}C_{r}$,where the k's are the integers in the combination and r=1,2,.. , e.g. for the combination 1 3 4, ${}^{1}C_{1}$ + ${}^{3}C_{2}$ + ${}^{4}C_{3}$ = 8. Unlike permutations, the value of `n` need not appear in a combination, and so its number is independent of `n`. The following verb thus gives unique combination numbers :

```
   ctoi=:monad : '+/(>:i.#y)!y'  NB. combination to integer
   ctoi 1 3 4
8
```

A challenge is to produce `itoc` such that `8 itoc 3` is `1 3 4` .

The emphasis in R.E.Boss's article is on the pragmatic matter of how to generate combinations more efficiently using lines such as `[:`

`(,.&.><@:\.)/ >:@-~ [\i.@]`. On deconstruction this shows that orderly lists of combinations are a little closer to primitives in J than might at first sight be imagined. The key is the combination of *box* and *stitch*, *suffix*, for which a preliminary note on the 'fix' family of adverbs is in order, namely *prefix*(\ monadic), *suffix* (\. monadic), *infix*(\ dyadic) and *outfix* (\. dyadic). These have the general effect of making objects larger either by increasing rank as in

```
   (,\i.5);(,\.i.5)       NB. ravel prefix;ravel suffix
┌─────────┬─────────┐
│0 0 0 0 0│0 1 2 3 4│
│0 1 0 0 0│1 2 3 4 0│
│0 1 2 0 0│2 3 4 0 0│
│0 1 2 3 0│3 4 0 0 0│
│0 1 2 3 4│4 0 0 0 0│
└─────────┴─────────┘
```

or by repetition :

```
   <\.i.5                          NB. box suffix
┌─────────┬───────┬─────┬───┬─┐
│0 1 2 3 4│1 2 3 4│2 3 4│3 4│4│
└─────────┴───────┴─────┴───┴─┘
```

The dyadic form *infix* delivers overlapping `x`-lists and *outfix* delivers the result of progressively removing them :

```
    (2,\i.5);(2,\.i.5)
┌───┬─────┐
│0 1│2 3 4│
│1 2│0 3 4│
│2 3│0 1 4│
│3 4│0 1 2│
└───┴─────┘
```

The dyadic form *infix* delivers overlapping `x`-lists and *outfix* delivers the result of progressively removing them. An informal rule is that without dots (that is *prefix* and *infix*) things proceed from the left, with dots they do so from the right.

Combinations

A deconstruction of `comb2` in [1] for the orderly listing of combinations begins with

```
   |:3 ,\i. 6       NB. dyadic infix
0 1 2 3
1 2 3 4
2 3 4 5
```

Apply *box-ravel suffix* to the final row above :

```
      <@,\.2 3 4 5
┌───────┬─────┬───┬─┐
│2 3 4 5│3 4 5│4 5│5│
└───────┴─────┴───┴─┘
```

and then *stitch* the numbers in the penultimate row on an item by item basis :

```
   1 2 3 4 ,.each<@,\.2 3 4 5
┌───┬───┬───┬───┐
│1 2│2 3│3 4│4 5│
│1 3│2 4│3 5│   │
│1 4│2 5│   │   │
│1 5│   │   │   │
└───┴───┴───┴───┘
```

Call this form of *stitch*ing `Stitch` with an upper case `S` to distinguish it from the name of the primitive *stitch* :

```
   Stitch=.,.each <@;\.
```

so that the preceding display is obtained as

```
   1 2 3 4 Stitch 2 3 4 5
```

By applying this successively to the columns of `3 ,\i.6` , everything is in place to write a generalized verb for generating combinations of `x` from `y`. (The name `comb2` is chosen because this is the technique described by that name in Boss's article.)

```
   comb2=:dyad : 'z=.Stitch/|:x,\i.y'
   3 comb2 6
┌─────┬─────┬─────┬─────┐
│0 1 2│1 2 3│2 3 4│3 4 5│
│0 1 3│1 2 4│2 3 5│     │
│0 1 4│1 2 5│2 4 5│     │
│0 1 5│1 3 4│     │     │
│0 2 3│1 3 5│     │     │
│0 2 4│1 4 5│     │     │
│0 2 5│     │     │     │
│0 3 4│     │     │     │
│0 3 5│     │     │     │
│0 4 5│     │     │     │
└─────┴─────┴─────┴─────┘
```

A final ; (*raze*) could be use to transform the above into normal unboxed lists – retaining the boxes both helps appreciation of the structure, and also reduces the number of print lines required.

The integer representations of the combination list `3 comb2 6` in the order given above are

```
    ;ctoi each<"1 ;3 comb2 6
0 1 4 10 2 5 11 7 13 16 3 6 12 8 14 17 9 15 18 19
```

which shows that combinations generated by `comb2` are not in their 'natural' order.

The boxed result of `comb2` suggests that reduction could equally well have been applied from the right rather than the left by repeated use of the primitive verb *reverse* :.

```
    comb=.dyad : 'z=.|."1 each Stitch/|.|."1 |: x ,\ i. y'
   3 comb 6
┌─────┬─────┬─────┬─────┐
│3 4 5│2 3 4│1 2 3│0 1 2│
│2 4 5│1 3 4│0 2 3│     │
│1 4 5│0 3 4│0 1 3│     │
│0 4 5│1 2 4│     │     │
│2 3 5│0 2 4│     │     │
│1 3 5│0 1 4│     │     │
│0 3 5│     │     │     │
│1 2 5│     │     │     │
│0 2 5│     │     │     │
│0 1 5│     │     │     │
└─────┴─────┴─────┴─────┘
```

As a bonus `comb` delivers the combination list in reverse counting order :

```
    ;ctoi each<"1 ;3 comb 6
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Relationship to the Pascal Triangle

The number of boxes in `r comb n` is one more than `d=.n-r` and the numbers of combinations in the various boxes are directly derivable from the Pascal Triangle whose first few numbers are :

```
   !/~i.10
1 1 1 1 1  1  1  1  1   1
0 1 2 3 4  5  6  7  8   9
0 0 1 3 6 10 15 21 28  36
0 0 0 1 4 10 20 35 56  84
0 0 0 0 1  5 15 35 70 126
0 0 0 0 0  1  6 21 56 126
0 0 0 0 0  0  1  7 28  84
0 0 0 0 0  0  0  1  8  36
0 0 0 0 0  0  0  0  1   9
0 0 0 0 0  0  0  0  0   1
```

`3 comb2 6` and `3 comb 6` both consist of 20 combinations displayed in 4 boxes, the numbers of combinations in which are obtained from the right as the first four non-zero items in row 2 counting in origin 0. More generally `r comb n` has `d+1` boxes in which the numbers of combinations are given by the first `d+1` non-zero integers in row `r-1`. For example for `6 comb 9` read a total of 84 combinations from the Pascal triangle, then go one up along the diagonal and read 1+6+21+56 along the row.

When `r` is large relative to `n` it is more efficient to use complementary combinations as suggested by the symmetry of the Pascal triangle, for example

```
    time=.6!:2
   time each '2 comb 55';'53 comb2 55'
┌───────────┬─────────┐
│0.000388038│0.0130343│
└───────────┴─────────┘
   time '(<i.55)-."1 each 2 comb 55'
0.00245115
```

Defining an itoc verb

This challenge stated earlier requires the delivery of a unique combination, given an `r` and an integer `i` . If n is also given `i` must be in the range 0 .. ^nC_r -1. To do this, first find the largest k such that ^kC_r does not exceed `i`. Subtract ^kC_r from `i` and carry on repeating this process for (`i` – ^kC_r) and (r – 1). For example to find combination number 8 of `3 comb n`, $^5C_3$ = 10 which exceeds 8, but $^4C_3$ = 4 does not, so select 4 as rightmost element. 8 – 4 = 4 which exceeds $^3C_2$, so catenate 3 to the left of 4. 4 – 3 = 1 which is less than $^2C_1$ but equals $^1C_1$, so that the final digit is 1 and the required combination is 1 3 4. Here is this process expressed in J :

```
   lgstk=.dyad :0
i=.<:y                      NB. increase k up to lgst y!k <x
while. x>: y!>:i do. i=.>:i end.
)
   itoc=.dyad :0
x=.x-y!r=.x lgstk y
while. y>1 do.
   y=.<:y                            NB. decrement y
   r=.(x lgstk y),r         NB. catenate new value to left of r
   x=.x-y!{.r   end. r      NB. reduce x for next iteration
)
   8 itoc 3
1 3 4
```

Another Iterative Algorithm

`comb` and `comb2` are not the only methods for constructing combination lists. For example [1] starts by describing another such algorithm, and without worrying too much about the J details, tracing the iterative steps can be used to illustrate how alternative techniques reach the same goal by different routes.

```
   trace=.monad : 'y(1!:2)2'
   combi=.dyad : 0          NB. d is n-r for any given nCr
k=. i.>:d=.y-x              NB. k is a list of integers
z=.(d$<i.0 0),<i.1 0        NB. initially z is d+1 empty boxes
for_j. i.x do.              NB. loop thru items of i.x
trace z=. k Stitch >:each z  end.
)
   t=.3 combi 6
┌─┬─┬─┬─┐
│0│1│2│3│
└─┴─┴─┴─┘
┌───┬───┬───┬───┐
│0 1│1 2│2 3│3 4│
│0 2│1 3│2 4│   │
│0 3│1 4│   │   │
│0 4│   │   │   │
```

```
└───┴───┴───┴───┘
```

```
┌─────┬─────┬─────┬─────┐
│0 1 2│1 2 3│2 3 4│3 4 5│
│0 1 3│1 2 4│2 3 5│     │
│0 1 4│1 2 5│2 4 5│     │
│0 1 5│1 3 4│     │     │
│0 2 3│1 3 5│     │     │
│0 2 4│1 4 5│     │     │
│0 2 5│     │     │     │
│0 3 4│     │     │     │
│0 3 5│     │     │     │
│0 4 5│     │     │     │
└─────┴─────┴─────┴─────┘
```

Other methods are described in E #33 (“Perming and Combing”). The relative efficiencies of algorithms are parameter dependent. After removing `trace` from `combi` the following cases were tested informally :

```
   time each '9 comb 23';'9 comb2 23';'9 combi 23'
┌────────┬────────┬────────┐
│0.251543│0.173648│0.199808│
└────────┴────────┴────────┘
```

(n.b. `combi` had the trace removed for fair comparison)

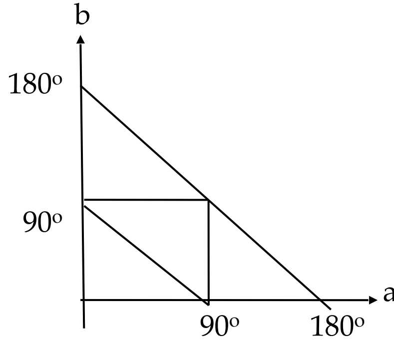
Code Summary

```
   ctoi=:monad : '+/(>:i.#y)!y'  NB. combination to integer
   Stitch=.,.each <@;\.    NB. extended stitch
   comb2=:dyad : 'z=.Stitch/|:x,\i.y' NB. list of combinations
   comb=.dyad : 'z=.|."1 each Stitch/|.|."1 |: x ,\ i. y'
   time=.6!:2              NB. timing J expressions

   itoc=.dyad :0           NB. integer to combination
x=.x-y!r=.x lgstk y
while. y>1 do.
   y=.<:y                  NB. decrement y
   r=.(x lgstk y),r        NB. catenate new value to left of r
   x=.x-y!{.r   end. r     NB. reduce x for next iteration
)
   lgstk=.dyad :0
i=.<:y                     NB. increase k up to lgst y!k <x
while. x>: y!>:i do. i=.>:i end.
)
   trace=.monad : 'y(1!:2)2'

   combi=.dyad : 0         NB. d is n-r for any given nCr
k=. i.>:d=.y-x             NB. k is a list of integers
z=.(d$<i.0 0),<i.1 0       NB. initially z is d+1 empty boxes
for_j. i.x do.             NB. loop thru items of i.x
trace z=. k Stitch >:each z  end.
)
```

35. How many Obtuse angled triangles are there?

Principal Topics : ?. (*roll*) o. (*circle functions*) \: (*grade down*) ~ (*reflex*), simulation, random numbers, random angles, accept-reject technique

In the world, that is. Lots, I know, but let me ask rather what <u>proportion</u> of all triangles in the world are obtuse-angled? Since any two of the angles, say a and b, of a triangle can be determined independently, the every possible triangle corresponds to a point within the large right-angled triangle in the diagram below



Acute-angled triangles must have $a+b>90^{o}$ and so their points all lie within the inner triangle, from which it follows that 75% of all triangles are obtuse-angled. Well let's ask J and see what he/she/it says. Assume that a point is represented by a 2-list of numbers such as `4 7`. The verb `py` calculates the square of the distance between two such points

```
   py=.(+/@:*:@:-/)"2       NB. square of dist between 2 points
   py 4 7,:1 3              NB. (3,4,5) triangle
25
```

A triangle is a 3-list of 2-lists (points). To calculate the squares of the three sides, take the points in pairs and apply `py` :

```
   sides=.(0 1;1 2;0 2)&({every)@<
   ]t=.3 2$_3 _2 _1 6 0 2
_3 _2
_1  6
 0  2
   <"2 sides t
┌─────┬────┬─────┐
│_3 _2│_1 6│_3 _2│
│_1  6│ 0 2│ 0  2│
```

A triangle is obtuse if the square of the largest side exceeds the sum of the squares of the other two, so define `ifbigone` which uses *reflexive grade down* to test whether the largest of a set of numbers is greater than the sum of all the others

```
   ifbigone=.({.>+/@}.)@\:~       NB. 1 if biggest > +/rest
   ifbigone 49 43 100
1
```

This leads naturally to

```
   obtest=.ifbigone@:((py"2)@sides) NB. 1 if obtuse
   obtest t
1
```

To generate random triangles centred on the origin and in the range (-y,+y), construct lists like `t` using an integer which determines coordinate resolution. This should be big enough to minimise the inaccuracy arising from the fact that `?` returns integers :

```
   randtri1=.-~?@>:@+:@(3 2&$)
   randtri1 100
 51 29
 36 58
_77 58
```

Everything is now in place to test a single random triangle with given resolution :

```
   trial1=.obtest@randtri1       NB. 1 if obtuse
   trial1 100
1
```

so the triangle in this particular trial was obtuse-angled. Here are a thousand trials at a higher resolution :

```
   +/trial1 each 1000$10000
722
```

... and a few more :

```
   +/trial1 every 1000$10000
720
   +/trial1 every 1000$10000
723
   +/trial1 every 1000$10000
748
```

By now you should be getting a feel for where the true value lies, somewhere between 72 and 75 per cent. Another way to conduct the simulation is to generate random angles rather than random sides. Start by writing a verb which generates a number of random values (angles) between 0 and 2π :

```
   rnd=.(%~?)@(1e9&(#~))   NB. y rand uniform nos. in (0,1)
   pi2=.o.2                NB. pi/2=6.28319
   randang=.(*&pi2)@rnd    NB. random angles in radians
   randang 3
5.66985 3.78605 2.69206
```

The sines and cosines of each angle generate a point within the unit circle, and so random angles can be transformed to random points within the unit circle by

```
   cs=.o.~/&2 1@randang
   cs 3
 0.930605 _0.366025
  _0.5352 _0.844725
_0.212795  0.977097
```

and then to random triangles in a circle with radius y by

```
   randtri2=.monad :'y*cs 3'
   randtri2 100
 99.0668  13.6299
_66.3328 _74.8329
_22.4049 _97.4578
```

All is in place for determining whether these triangles are obtuse or not, and for doing some more experiments

```
   trial2=.obtest@randtri2
   +/trial2 every 1000$10000
775
   +/trial2 every 1000$10000
739
   +/trial2 every 1000$10000
744
```

A little bit higher than our previous estimate, but close to the theoretical 75%. Maybe sampling from a square frame includes disproportionately 'skinny' acute-angled triangles which have one vertex tucked into one of the far corners. To test this conjecture, sample the coordinates within a square, but accept only those triangles all of whose points lie inside the largest inscribed circle. The final line of `randtri3` embodies recursively the 'accept-reject' technique which is common in simulation practice.

```
   ssq=.+/@:*:       NB. sum of squares
   randtri3=.monad :0
r=.randtri1 y
while. +./(*:y)<ssq"1 r do.
  r=.randtri3 y end.
)
```

As before everything is in place for the next experiment :

```
   trial3=.obtest@randtri3
   +/trial3 every 1000$10000
733
   +/trial3 every 1000$10000
721
   +/trial3 every 1000$10000
718
```

Not much different from `trial1` – ah well, that's one conjecture gone for a bang!

Another accept-reject technique for random triangles consists of generating the values of three random sides in order, and rejecting the result if the sides fail to form a triangle because one of them is bigger than the sum of the other two :

```
   randtri4=.monad : 0
r=.>:?3$y
while. ifbigone r do.
   r=.randtri4 y end.
)
```

Since `randtri4` returns lengths of sides the obtuseness test is applied to the squares of the sides :

```
   trial4=.ifbigone@:*:@randtri4
   +/trial4 every 1000$10000
561
   +/trial4 every 1000$10000
564
   +/trial4 every 1000$10000
566
```

Seems I've mislaid nearly 20% of the world's obtuse-angled triangles (careless of me!) Another possibility based on sides is to choose two sides at random, say these were 6 and 10. The range of acceptable values for a third side is then from 10-6 to 10+6, or more generally from m-n to m+n where m and n are the lengths of the sides in decreasing order. The third side can then be taken as (m-n) plus a number drawn at random from 0 to 2n:

```
   choose2=.\:~@(? , ?)    NB. two sides at random
   third=.,-/+?@>:@+:@{:   NB. join feasible random 3rd side
   rantri5=.third@:choose2
   trial5=.ifbigone@:*:@rantri5
   +/trial5 every 1000$10000
751
   +/trial5 every 1000$10000
746
```

That’s better, but let’s get back to the original question .   How many obtuse-angled triangles did I say there were? ...

Code Summary

```
   trial1=.obtest@randtri1        NB. first random trial
     obtest=.ifbigone@:((py"2)@sides) NB. 1 if obtuse
       ifbigone=.({.>+/@}.)@\:~   NB. 1 if biggest > +/rest
       py=.(+/@:*:@:-/)"2         NB. sq of dist between 2 pts
       sides=.(0 1;1 2;0 2)&({every)@< NB. sides from coords
     randtri1=.-~?@>:@+:@(3 2&$)  NB.
   trial2=.obtest@randtri2        NB. second random trial
     randtri2=.monad :'y*cs 3'
       cs=.o.~/&2 1@randang       NB. cos/sin of rand angle
         randang=.(*&pi2)@rnd     NB. random angles in radians
           pi2=.o.2               NB. pi/2=6.28319
           rnd=.(%~?)@(1e9&(#~))  NB. y rand uniform (0,1)nos.
   trial3=.obtest@randtri3        NB. third random trial

   randtri3=.monad :0             NB. accept-reject technique
r=.randtri1 y
while. +./(*:y)<ssq"1 r do.
  r=.randtri3 y end.
)
   ssq=.+/@:*:                    NB. sum of squares
   trial4=.ifbigone@:*:@randtri4  NB. fourth random trial

   randtri4=.monad : 0            NB. accept-reject technique
r=.>:?3$y
while. ifbigone r do.
   r=.randtri4 y end.
)
   trial5=.ifbigone@:*:@rantri5   NB. fifth random trial
     rantri5=.third@:choose2
       third=.,-/+?@>:@+:@{:      NB. join feasible third side
         choose2=.\:~@(? , ?)     NB. two sides at random
```

36. …the stylish part of Vector

Principal Topics : ? (*roll/deal*), a. (*alphabet*), ~(*passive* conjunction) E. (*member of interval*) # (*tally)* bridge hook, simulation, random words, random sentences

Browsing through back numbers the other day, it struck me that in spite of being in its eighteenth year, Vector has won no awards for outstanding literary merit, has received no Booker prize nominations, and features in no university reading lists for exemplary 20th century English prose. This situation demands remedy, and while the urge burned hot within me, I resolved to take some immediately necessary steps. I had heard recently that if one is allowed to oversee a couple of million monkeys equipped with an equal number of keyboards, it's a near cert that one of the little brutes will eventually outperform Shakespeare. So with my computer thus dedicated to monkey business, here is what emerged.

Start with the distinction between `7?10` and `7(?@#)10` which give 7 random numbers from the set 0,.., 9 without and with repetition respectively. In the former case the left argument must be no greater than the right. The latter phrase uses *copy* `(`dyadic `#)` to replicate the 10 seven times prior to *roll*ing.

It is easy using indexing and *tally* `(`monadic `#)` to extend randomisation to lists such as

```
   lets=.(97+i.26){a.      NB. lower case alphabetic chars.
   lets
abcdefghijklmnopqrstuvwxyz
```

To obtain a single random letter from `lets` say

```
   ran=.{~?@#
   ran lets
j
```

Suppose that you want the number of random numbers in a selection to be random. Two applications of ? are needed :

```
   10(?~?)7
3 7 6 8 9
```

The bridge hook `10(?~?)7` means `(?7)? 10` that is, the roles of the right and left arguments have been reversed by using the *passive* conjunction. Because there can be no repetitions, the right argument should be no greater than the left, since, depending on the luck of the draw you might hit

```
    7(?~?)10
|domain error: q    NB. looks like this ? 10 was >7
|   7    (q~q)10
```

If you want to exclude the possibility of an empty selection use `10(?~q)7` where

```
   q=.>:@?           NB. random integers from 1 to y
   6 q 7             NB. deal 6 random ints. from 1 to 7
6 2 4 7 5 1
```

A nice little palindromic fork `#(?~?)#` delivers the indices of a random selection of letters :

```
   (#(?~?)#)lets
6 17 20 10 15 8 9 22 13 18 19 23 7 24
```

and the addition of *from* makes this into a random selection from the list

```
   ({~#(?~?)#)lets
xprhwkf
```

In order to obtain the indices of, say, ten random letters with repetition say :

```
   (10&(?@#)@#)lets
10 15 19 22 8 4 19 18 22 1
```

and to convert these to indices use *from* as before :

```
   ({~10&(?@#)@#)lets
dpaxjnighg
```

The problem with this as a technique for random words is that words like this are not very beautiful and are almost certain to be unpronounceable. One possible strategy to overcome this might be to use as weights a list of rough relative frequencies of occurrence of the various letters in English :

```
   fr=.8 2 3 4 12 2 2 6 8 1 1 4 3 8 8 2 0 6 8 8 3 1 2 1 2 1
   #fr
26
   +/fr
106
```

The 16th letter ‘q’ has been given the value of 0 on account of its tiresome requirement to be followed by a ‘u’, a matter which can comfortably be left to a later refinement. A weighted letter list is then

```
   wlets=.fr#lets                NB. weighted letters
   #wlets                        NB. sum of weights
106
```

```
   ({~10&(?@#)@#)wlets
iedorthgwh
   ({~10&(?@#)@#)wlets
doakidsoas
```

At least the words look vaguely like words, and some (but only some) are just about pronounceable! Somewhat greater control of the patterns, as opposed to the mere frequencies, of vowels and consonants is desirable, so weight the vowels and the consonants separately:

```
   wv=.2 3 2 2 1#'aeiou'
   wc=.2 3 4 2 2 6 1 1 4 3 8 2 0 6 8 8 1 2 1 2 1#
                    (lets-.'aeiou')
```

Next determine some vowel/consonant patterns, and randomize each component in turn. Since a random drawing is made from each element of the letter pattern it is expedient to define

```
   rand=.ran every
   wpat=.wc;wv;wc;wv;wc
   rand wpat
terel
   wpat1=.wv;wc;wc;wv;wc
   rand wpat1
ettum
```

The patterns can be extended using `$` to allow for longer words, and at the same time the occurrence of unpleasing single letter words can be inhibited by using

```
   rint2=.>:@>:@?   NB. random integers from 2 to y+1
```

which is used to provide a random argument for *take*:

```
   (rint2 12){.rand 12$wpat      NB. word of at most 13 lets
pedetnebec
```

Everything is now in place to define a verb to produce random words:

```
   rw=.(rint2@[){.rand@$
   12 rw wpat
selimnehekja
   12 rw wpat1
apsaku
   12 rw wv;wc
eromoloner
```

Prefixes and suffixes help, and prototype lists of each are built into the verbs :

```
   pre=.(4#<''),'ex';'un';'in';'pre'
```

```
   suf=.(3#<''),'est';'ist';'s';'ed';'ism'
```

following which :

```
   rword=.dyad :'(>ran pre),(x rw y),>ran suf'
   10 rword wpat
johism
   10 rword wpat1
apmopuphed
```

What about random sentences? Different strategies are available. One possibility is simply to string along a random number of random words with as left argument the maximum word length before inflections, and the right argument the maximum sentence length :

```
   rsent1=.dyad :0
s=.'' [ i=.0 [ lim=.?y
while.i<lim do.
  s=.s,(x rword wpat),' ' [ i=.i+1
end.
)
   6 rsent1 9
intoto sots labist nime prediwoft extatotced sudirts
   7 rsent1 10
undusatbo presalos
```

and of course the global variable `wpat` is also available for tweaking. A bit short in any attributable meaning (my spellcheck also sees red!) but the likes of Kenneth Branagh might be able to put it across for a fee!

Another view is that a random sentence is a random drawing from an available list of words in the same way that a random word is a random drawing from an available random list of letters. In the case of a sentence the list is called a vocabulary, which can be divided into separate sub-lists of, for example, `nouns`, `verbs`, and `adjs`. Only the start of the definitions of these are shown, the rest should become apparent as matters proceed, and no doubt will provide invaluable physcoanalyitic evidence for future diagnosis of my various personality disorders :

```
   nouns=.'cats';'degree';'philosophy'; ..
   verbs=.'hopes';'keeps';'flies';'steps'; ..
   adjs=.'great';'lean';'fat';'blue'; ..
```

Define a sentence pattern analogous to the consonant/vowel patterns :

```
   spat=.(<adjs),(<nouns),(<verbs),(<adjs),(<nouns)
```

Parts of speech patterns are analogous to the vowel/consonant pattern for words :

```
   rand spat
┌────┬──────────┬─────┬───┬──────┐
│blue│philosophy│steps│fat│degree│
└────┴──────────┴─────┴───┴──────┘
```

Add a standard idiom for removing blanks :

```
   rdb=.#~-.@('  '&E.)
   rdb,(>rand spat),.' '
empty boat eats old feet
```

and here is the basis for some random sentences :

```
   rsent2=.rdb@,@:(>@(,&>&' '@:ran every))
   rsent2 spat
lean tradition treads creepy happiness
```

As with letter patterns, different parts of speech patterns add variety :

```
   spat1=.(<adjs),(<adjs),(<nouns),(<verbs),(<nouns)
   rsent2 spat1
sneaky greek catalyst steps faith
   rsent2 (<adj),(<nouns),(<verbs),(<nouns),(<verbs)
nice faith mates hope castigates
```

Chomsky, thou should'st be living at this hour!

Of course things are still in the early stages but there are enough controls in place to make incremental refinements, and I think I can claim to have given my monkeys a head start. Progress so far is thus promising, and I confidently expect Vector to be awarded a Nobel prize for Literature before too long. Perhaps our friends in the Swedish APL Association could show the spirit of brotherliness and pull whatever strings are necessary to accelerate this outcome.

And in case you are concerned that I may nothing left to say in future articles, don't worry, I have six million of them or so already written. And what is more they are all completely fresh, not a single hint of repetition anywhere!

Code Summary

The following are of general usefulness in creating random text

```
   lets=.(97+i.26){a.      NB. lower case alphabetic chars.
   ran=.{~?@#              NB. single random character
   rand=.ran every         NB. random drawings from patterns
```

Thereafter everyone so engaged will have their own style, and so the sequences below for first random words, and then random sentences are entirely personal!

```
   rword=.dyad :'(>ran pre),(x rw y),>ran suf  NB. random word
     rw=.(rint2@[){.rand@$
       rint2=.>:@>:@?      NB. random integers from 2 to y+1
     pre=.(4#<''),'ex';'un';'in';'pre'         NB. prefix
     suf=.(3#<''),'est';'ist';'s';'ed';'ism'   NB. suffix
   wpat=.wc;wv;wc;wv;wc    NB. word pattern; wc=cons, wv=vowel
     wv=.2 3 2 2 1#'aeiou'
     wc=.2 3 4 2 2 6 1 1 4 3 8 2 0 6 8 8 1 2 1 2
1#(lets-.'aeiou')

   rsent1=.dyad :0                NB. random sentence style 1
s=.'' [ i=.0 [ lim=.?y
while.i<lim do.
  s=.s,(x rword wpat),' ' [ i=.i+1
end.
)
   rsent2=.rdb@,@:(>@(,&>&' '@:rand)) NB. rand sentence style 2
     rdb=.#~-.@('  '&E.)               NB. remove blanks
   spat=.(<adjs),(<nouns),(<verbs),(<adjs),(<nouns)
     nouns=.'cats';'degree';'philosophy'; ..
     verbs=.'hopes';'keeps';'flies';'steps'; ..
     adjs=.'great';'lean';'fat';'blue'; ..
```

37. Greed : patterns for the collapse of Western capitalism

Principal Topics : t. (*Taylor coefficients*) p. *(polynomial)* ` *(gerund*), @. (*agenda)* {: *(tail)* }: *(curtail)* generating functions, binomial theorem, Fibonacci numbers, partitioning, recursion, greedy algorithm, distance tables

In a communication following "Fifty Ways to tell a Fib", (see E #42) Ken Iverson pointed out that

$$x/(1-x-x^2)$$

is a generating function for the Fibonacci numbers which in J terms is

```
   fibfn=.0 1&p. % 1 _1 _1&p.
```

Rewriting the generating function as

$$x(1-x(1+x))^{-1}$$

the binomial theorem gives the series expansion

$$x(1+x(1+x)+x^2(1+x)^2+x^3(1+x)^3+\dots$$

Write the coefficients of the various binomial coefficients as rows of a table :

x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9
1								
1	1							
	1	2	1					
		1	3	3	1			
			1	4	6	4	1	
				1	5	10	10	5
					…. . .	. .		

Adding down the columns should make it clear how the Fibonacci coefficients arise. But why do all this work when J will do it for you using the t. adverb to give the best fitting polynomial of given degree, say the 13th. ?:

```
   fibfn t. i.14
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

which has the same result as the expression in E #42 :

```
   f=.(,+/@(_2&{.))^:12(0 1)
```

Yet another interesting Fibonacci fact is that every positive integer can be written as the sum of a series of non-consecutive Fibonacci numbers, non-consecutive because every sum of a consecutive pair can immediately be replaced by the next higher Fibonacci number – this is the fundamental Fibonacci property. To obtain such a sum, start with a general verb which transforms any given number k into the 'highest value below' in a numeric list :

```
   hvb=.>./ @(>:#]) NB. highest value below
   100 hvb f
89
```

Define

```
   gap=.-hvb&f
   gap 100
11
```

Both the 89 and the 11 are required, the first to be stored, and the second to be processed further, so define a verb which partitions an integer in this fashion :

```
   Fgap=.(hvb,[-hvb)&f     NB. partition into fib+gap
   Fgap 100
89 11
```

Using `&` makes `f` into a pseudo-constant which would be changed only if a shorter or longer Fibonacci series was required. Also, the double computation of the verb `hvb` in `Fgap` is inherently displeasing – this can be circumvented by rewriting the slightly less transparent verb :

```
   Fptn=.{:@(>:#(],.-))    NB. Fibonacci partition
   100 Fptn f
89 11
```

following which `Fsum` remembers the left hand part of the list and processes the gap on the right :

```
   Fsum=.}:,Fptn&f@{:      NB. single step process
   Fsum 89 11
89 8 3
```

8 and 3 are both Fibonacci numbers, and so 100 = 89 + 8 + 3.

This process can be repeated as long as necessary using a recursive verb

```
   Fsumr=.($:@Fsum)`]@.(e.&f@{:)
```

```
   Fsumr 100
89 8 3
```

which says in effect "go on reducing the gap until you reach a Fibonacci number".

100 (= 89 + 8 + 3) can also be expressed in a binary form in which a 1 represents an integer in the reverse of the Fibonacci sequence

```
   Fnum=.|.@(2&}.@(f&e.)@Fsumr)
   Fnum 100
0 0 1 0 0 0 0 1 0 1 0 0
```

call such numbers Finary numbers, say - they will be revisited later.

A characteristic of the above algorithm is that at every step the Fibonacci number giving the smallest gap was grabbed, a characteristic of greedy behaviour as exemplified alike by small boys sharing cakes and fat-cat directors raiding their company takings.

The term **greedy algorithm** is used generically to describe a range of algorithms which share this general characteristic. It arises sufficiently often to merit description as a **pattern**, a word which has become a technical term in the vocabulary of Object Oriented programming to mean any common way of doing things - more general than an **idiom** or a **phrase**, but not large enough to be considered as a piece of software architecture.

Patterns seems a good way of describing general techniques which emerge intuitively in J programming through recurrent use and reinvention. For example, the greedy technique apparent in `Fsumr` above might also have emerged in programming a distance reducing route starting from a given town and visiting all other towns represented in a given distance table. To be specific suppose that a distance table for four towns is

```
0  1  5  3
1  0  7  6
5  7  0 10
3  6 10  0
```

The routing problem is not altered by subtracting all the non-zero entries in the table from 11 and making the objective maximization rather than minimization. So define `m` as

```
 0 10 6 8
10  0 4 5
```

```
        6   4  0  1
        8   5  1  0
```

Starting at town 0, initial greed says find the route which gives the highest reward from town 0. This is the route to town 1, from which the next greedy step is to go from town 1 to town 3 since 5>4, and the route is completed by visiting 2, thereby adding another 1 to the reward, a total of 16.

To preserve indexing it is prudent not to reduce the distance table at each recursive step but rather to reduce the reward to 0 for steps which proceed to towns already visited. Thus if the 'route so far' is 0 2 1, the next town is identified by its index as the one offering the highest reward in row 1 after avoiding revisits, which requires that items 0 and 2 in row 1 must first be amended to 0 by `0(0 2)}1{m`. The index of the next town is generated greedily by

```
   imax=.i.>./              NB. index of maximum value
   nextg=.dyad :'imax 0(}:x)}({:x){y'
                            NB. next town in greedy chain

   0 2 1 nextg m
3
```

As with `f` above, a distance table would be unlikely to have frequent changes and so it makes sense to build it in as a pseudo-constant `m` :

```
   optnextg=.nextg&m        NB. row index of optimum next town
   optnextg 0 2 1
3
   optnextgr=.]`($:@(],optnextg))@.(4&~:@#)
   optnextg r 0
0 1 3 2
```

A note should be made that a change in the distance matrix might require a change in the constant 4.

There is an important distinction between these two manifestations of greed. In the first, the representation of a positive integer as a sum of Fibonacci numbers can be proved to be unique, and so *any* algorithmic pattern would produce the same result. This is not true in the second example for which the route 0 3 2 1 has the value of 17, demonstrating that greed is not always the best policy!

It should also be stressed that it is recursion in the presence of *maximum* which makes these patterns greedy, not recursion alone. Simple recursion is a generalisation of the greedy pattern. This can be illus-

trated by venturing even further into Fibo-land and constructing a Finary adder. First assume that an ordinary binary adder is available :

```
   badd=.+&.#.             NB. binary addition
   1 0 1 badd 1 0 0
1 0 0 1
```

The main difference between binary and Finary numbers is that Finary numbers never contain consecutive 1s, and so if two consecutive 1s were ever to turn up in an intermediate calculation these would be immediately resolved into a single 1 at the level of the next higher digit. A verb to detect such a state of affairs is :

```
   find11=.0&,@(2&(*./\))  NB. find sublist 1 1
   b                       NB. representation of 148
1 0 1 0 1 1 0 1 0 1
   find11 b
0 0 0 0 0 1 0 0 0 0
```

Simple binary addition of these two lists makes the necessary adjustment :

```
   rep11=.badd find11             NB. replace with 1 0 0
   rep11 b                        NB. representation of 148
1 0 1 1 0 0 0 1 0 1
```

Having made one such replacement it is possible that, as in the case above, the new 1 produces a further pair of consecutive 1s, which in turn may generate a ripple effect through the whole Finary number. Adopting the recursive pattern again, define :

```
   rep11r=.]`($:@rep11)@.(+./@find11)
   rep11r b                       NB. representation of 148
1 0 0 0 0 0 0 0 1 0 1
```

Enough is now in place for a verb to add 1 to a given Finary number :

```
   Fadd1=.rep11r@(1&badd)
   Fadd1 1 0 1 0 1                NB. add 1 to Finary 12
1 0 0 0 0 0
```

To perform a general Finary addition, e.g. 1 0 1 0 Fplus 1 0 0 0 0 1, observe that the recursive pattern requires that the data is in the form of a single argument. One technique in the present case is to keep on applying `Fadd1` to one of the summands until the value in a counter matches the other, hence a suitable data format for the above sum is

```
   u=.0;1 0 1 0;1 0 0 0 0 1       NB. Finary 0(cntr),7,14
```

and a single step of the addition process is :

```
   Fplus1=.Fadd1&.>@(2&{.),{:     NB. Add 1 to each of 0,7
   Fplus1 u
┌─┬─────────┬───────────┐
│1│1 0 0 0 0│1 0 0 0 0 1│         NB. Finary 1(cntr),8,14
└─┴─────────┴───────────┘
```

Applying the general recursive pattern yet again leads to

```
   Fplus=.($:@Fplus1)`(>@(1&{))@.({.-:{:)
   Fplus u
1 0 0 0 0 0 0
```

The problems above may in themselves be academic, but the use of patterns themselves is a highly practical programming matter. Using patterns is matter of <u>design</u>; in the examples described above, the generic pattern can be informally described as fnr = fnr (fn), that is, a recursive verb defined as "itself applied to the result of a matching single step verb". Many, although certainly not all, programming problems at this level are susceptible to this design pattern. To use such a pattern, three questions must have clear answers :

1. Can I describe a single step verb whose result is the input to the next step?
2. How do I know when to stop the recursion?
3. What do I want to happen when it does?

What happens below the level of fnr = fnr (fn), for example whether code is tacit, explicit or a mixture, is a matter of programmer choice. For example, in the last case, an iterative design might have led to an implementation something like

```
   Fplus=.dyad :0
s=.0 [ r=.y
while.(-.s-:x) do.
r=.Fadd1 r [ s=.Fadd1 s end.
   1 0 1 0 Fplus 1 0 0 0 0 1          NB. 7 + 14
1 0 0 0 0 0 0
```

Finally for those of you whose sensible habit is always to skip to the end, thereby cutting out all the tedious stuff in the middle, this article has been all about Fibonacci, patterns and greed, but the greatest of these is … (reader to complete)

Code Summary

```
   fibfn=.0 1&p. % 1 _1 _1&p.     NB. Fib nos by generating fn
   f=.(,+/@(_2&{.))^:12(0 1)      NB. ditto by direct method
```

```
   Fnum=.|.@(2&}.@(f&e.)@Fsumr)

Fsumr=.($:@Fsum)`]@.(e.&f@{:)
      NB. partitions integer into sum of Fibonacci numbers
   Fsum=.}:,Fptn&f@{:      NB. single step process
      Fptn=.(hvb,[-hvb)&f  NB. partition into fib+gap
        hvb=.>./ @(>:#])   NB. highest value below

Fplus=.($:@Fplus1)`(>@(1&{))@.({.-:{:)  NB. addn of Finary nos
     Fplus1=.Fadd1 each@(2&{.),{:       NB. Add 1 to each
       Fadd1=.rep11r@(1&badd)
         rep11r=.]`($:@rep11)@.(+./@find11)
         Fadd1=.rep11r@(1&badd)
           rep11=.badd find11
              badd=.+&.#.               NB. binary addition
                 find11=.0&,@(2&(*./\))  NB. find list 1 1
```

Greedy Algorithm applied to distance table

```
   optnextgr=.]`($:@(],optnextg))@.(4&~:@#)
     optnextg=.nextg&m     NB. row index of optimum next town
          nextg=.dyad :'imax 0(}:x)}({:x){y'
                    NB. next town in greedy chain
        imax=.i.>./ NB. index of maximum value
```

38. Shortest Paths

Principal Topics: e. (*member in*) -: (*match*) ` (*gerund*) @. (*agenda)* networks, nearest neighbour, symmetric networks, random networks, weighted graphs, shortest path, critical path, slack, connectivity, reachability, feasibility, loops

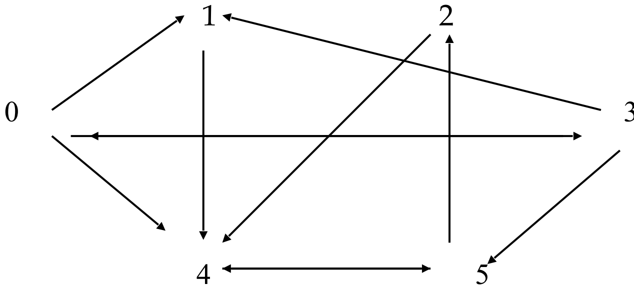
Finding shortest paths through networks is a basic technique in graph theory. Investigating these in J turns out to be a highly practical application of the verb `JE` (standing for Join Each) described in E #4 (“Parallel joins”). `JE` performs scalarised joins, that it is a derivative of *append* which behaves like a scalar verb with a result which is always a boxed list or a list of boxed lists.

```
box=.]`<@.(-:>)   NB. boxes if unboxed, else do nothing
JE=.,each &box    NB. scalarised join, results boxed
```

The following examples demonstrate how `JE` works :

```
   1 2 JE 3 4 5                    NB. two lists, append and box
┌─────────┐
│1 2 3 4 5│
└─────────┘
  (1 2;4 5 6) JE 7 8 9             NB. cf. 1 2 + 3
┌─────────┬───────────┐
│1 2 7 8 9│4 5 6 7 8 9│
└─────────┴───────────┘
  1 2 JE 4 5;7 8 9                 NB. cf. 1 + 2 3
┌───────┬─────────┐
│1 2 4 5│1 2 7 8 9│
└───────┴─────────┘
   (1 2;2 4 6) JE 4 5;7 8 9
┌───────┬───────────┐
│1 2 4 5│2 4 6 7 8 9│
└───────┴───────────┘
                                   NB. cf. 1 2 + 3 4
```

To begin the search for shortest paths, it helps to be specific by using a network in the form of a directed graph such as the following :



There are three reasonably obvious ways in which such a network can be represented. First it can be represented as a list of 2-lists each of which is a directed arc

```
arcs=.0 1;0 3;0 4;1 4;2 4;3 0;3 1;3 5;4 5;5 2;5 4
```

or this can be compacted into a list of n 'nearest neighbour' lists where n is the number of nodes :

```
n1=.1 3 4;4;4;0 1 5;5;2 4
```

or thirdly it can be represented as a matrix in which unconnected nodes are represented by _ :

```
   ]g1=.1(arcs)}6 6$_
_ 1 _ 1 1 _
_ _ _ _ 1 _
_ _ _ _ 1 _
1 1 _ _ _ 1
_ _ _ _ _ 1
_ _ 1 _ 1 _
```

If required the matrix in which unconnected nodes are represented by 0s is simply `%g1`.

The nearest neighbour form can be recovered from the matrix form by

```
   btoi=.# i.@#            NB. binary to list of 1-positions
   mton=.(btoi@:%)each@<"1 NB. matrix to nst neighbour
   mton g1
┌─────┬─┬─┬─────┬─┬───┐
│1 3 4│4│4│0 1 5│5│2 4│
└─────┴─┴─┴─────┴─┴───┘
```

and the reverse transformation is :

```
   ntom=.monad :'1(;(i.n),each each y)}(n,n=.#y)$_'
```

The rationale for using _ to represent unconnected nodes is partly visual, but more importantly, the following verb spm delivers the shortest path matrix (that is, the matrix whose values are the lengths of the shortest paths between all pairs of nodes) by using the 'minimum-dot-plus' inner product applied using the power conjunction as many times as there are nodes in the graph.

```
   spm=.monad : '<./y(<./ .+)^:(i.#y)y'
   spm g1
2 1 3 1 1 2
_ _ 3 _ 1 2
_ _ 3 _ 1 2
1 1 2 2 2 1
_ _ 2 _ 2 1
_ _ 1 _ 1 2
```

spm has the merit of working equally well for weighted graphs, that is if the non-infinity values in the base matrix are path-lengths rather than just ones.

```
   wts=.2 7 6 1 5 7 6 1 4 8 4
   h1=.wts(arcs)}6 6$0
   h1
0 9 0 7  6 0
0 0 0 0 11 0
0 0 0 0  5 0
7 6 0 0  0 1
0 0 0 0  0 4
0 0 8 0  4 0

   ind =. adverb : '(i.@$*x)@]'
   spm _(=&0 ind)}h1
14 2 15  7 3 7
 _ _ 13  _ 1 5
 _ _ 17  _ 5 9
 7 6  9 14 5 1
 _ _ 12  _ 8 4
 _ _  8  _ 4 8
```

For testing purposes, a random network, y square with a density x is obtainable by :

```
   randg=.dyad : '%(100*x)>?(y,y)$100'
   0.25 randg 8
_ _ _ _ _ 1 _ _
_ 1 1 _ _ 1 1 1
_ _ _ _ _ _ _ _
_ _ _ 1 1 1 _ 1
_ 1 _ _ 1 _ 1 _
_ 1 1 1 _ _ _ _
_ _ _ 1 _ 1 _ _
```

```
1 _ _ _ _ _ _ 1
```

and a random weighted matrix with weights in the range 1 to 100 by

```
   randh=.dyad : '(*~:&_)(>:?(y,y)$100)(*&.%)x randg y'
   0.25 randh 8
 0  0 61 0  0 0  0  0
 0  0  0 0  0 0  0 50
65 53  0 0 96 4  0  0
60 21  0 0  0 0  0  3
 0  0  0 0  0 0  0  0
14  0  0 0  0 0  0 23
 0  0  0 0 40 0  0  0
44  0  0 0  0 0 84  0
```

If symmetry is a requirement a suitable random matrix is generated by

```
    mksym=.(+|:) @ (*<:/~@(i.@#))
    %mksym% 0.6 randh 4
16 79  2  0
79 35  0  0
 2  0  7 14
 0  0 14 20
```

Before embarking on shortest path and critical paths routines, it is sensible to test for their possible existence. The **connectivity matrix** (a matrix which has a 1 for each pair of nodes for which a route exists) is a straightforward derivative of `spm`.

```
   connected=.~:&_@spm
   connected g1
1 1 1 1 1 1
0 0 1 0 1 1
0 0 1 0 1 1
1 1 1 1 1 1
0 0 1 0 1 1
0 0 1 0 1 1
```

The list of lists of **reachable** nodes from each node in order is given by

```
   reachable=.btoi each@<"1@connected
   ]u=.reachable g1
┌───────────┬─────┬─────┬───────────┬─────┬─────┐
│0 1 2 3 4 5│2 4 5│2 4 5│0 1 2 3 4 5│2 4 5│2 4 5│
└───────────┴─────┴─────┴───────────┴─────┴─────┘
```

which can be expanded to a list of all possible journeys of which the first few are demonstrated for

```
   feasible=.;@:((i.@#)(,each each)reachable)
```

```
   12{.u=.feasible g1
```

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

`ShtstPath` below returns the actual shortest path between any single pair of vertices of an unweighted graph . It begins with a line to check that such a path exists in the first place

```
   ShtstPath=:dyad : 0
if._=(<y){spm x do.'no path' return.end.
r=.{.y [ tgt=.{:y
while.(-.tgt e.tails r)do. r=.r extend x addnodes r end.
r=.>(tgt=tails r)#r
)
   tails=.{:every
   addnodes=.<@[rtoi each tails@]
   rtoi=.btoi@:(%@{~)
   extend=.;@:(JE"1 0 each)

    g1 ShtstPath 0 2
0 3 5 2
0 4 5 2
```

which can be speeded up by

```
   ShtstPath1=.dyad : 0
if._=(<y){spm x do.'no path' return.end.
p=.i.0 [ r=.{.y [ 'src tgt'=.y    NB. p=paths found
while.(-.tgt e.tails r)do.        NB. r=paths ongoing
  r=.(r-.p)-.(tails p)e.src [ p=.p,((tails r)e.tgt)#r
  r=.r extend x addnodes r end.   NB. np=newpaths
r=.>(tgt=tails r)#r               NB. reject incomplete paths
)
   g1 ShtstPath1 0 2
0 3 5 2
0 4 5 2
```

Next here is an algorithm which returns <u>all</u> paths between a pair of nodes in an <u>unweighted</u> matrix :

```
   AllPaths=.dyad : 0
if._=(<y){spm x do.'no path' return.end.
p=.i.0 [ r=.{.y [ 'src tgt'=.y          NB. initialise loop
while.(0~:#r)do.                        NB. r = incomplete
paths
  r=.r-.p [ p=.p,((tails r)e.tgt)#r     NB. p = complete paths
  r=.r extend x addnodes r              NB. add one node to
each r path
  r=.(#~((=&#~.)every))r                NB. remove loops
end. p
)
   g1 AllPaths 0 2
```

```
┌───────┬───────┬─────────┬───────────┐
│0 3 5 2│0 4 5 2│0 1 4 5 2│0 3 1 4 5 2│
└───────┴───────┴─────────┴───────────┘
```

In the case of a weighted matrix it may not be the case that the physically shortest path is the one with the lowest weighted value. To calculate the weighted value of a path use

```
      steps=.<"(1)@(2&(,/\))
   values=.+/@:([{~steps@])
   h1 values 0 3 5 2
16
```

Then to find a critical path, all that remains is to find that minimum value path from all paths :

```
   >(<h1) values each g1 AllPaths 0 2
16 18 15 26
   ((=<./)>(<h1) values each t)#t=.g1 AllPaths 0 2
┌─────────┐
│0 1 4 5 2│
└─────────┘
```

To build this into a critical path verb with a single argument `h1`, observe first that `g1` is derived from `h1` by

```
   unweight=.monad : '(%@~:&0)y'
   unweight h1
_ 1 _ 1 1 _
_ _ _ _ 1 _
_ _ _ _ 1 _
1 1 _ _ _ 1
_ _ _ _ _ 1
_ _ 1 _ 1 _

   CritPath=:dyad :0       NB. critical path
if._=(<y){spm x do.'no path' return.end.
t=.( unweight x)AllPaths y NB. all paths
v=.>(<x) values each t     NB. all paths evaluated
(((=<./)v)#t),<<./v        NB. minimum value path selected
)
   h1 CritPath 0 2         NB. critical path and value
┌─────────┬──┐
│0 1 4 5 2│15│
└─────────┴──┘
```

If there is more than one critical path, all are reported as in the following graph together with their common value :

```
   h2
 0 0  0  0  0 10  0
```

```
 2 0  6  0 16 12 13
 0 0  3 10  0  0  0
 0 0  1  0  0  0  0
10 0 19  0  3  5  0
14 0  0  9  0  0  1
 0 0  0  0  0  0 15
   h2 CritPath 1 6
┌───┬─────┬───────┬──┐
│1 6│1 5 6│1 0 5 6│13│
└───┴─────┴───────┴──┘
```

For a list of slack values associated with each path only a slight modification to `CritPath` is needed

```
   slack=.-~>./
   Slacks=.dyad :0
if._=(<y){spm x do.'no path' return.end.
t=.(unweight x)AllPaths y
>t JE each slack >(<x)values each t
)
    h1 Slacks 0 2
┌──────────┬─────────┬────────────┬─────────────┐
│0 3 5 2 10│0 4 5 2 8│0 1 4 5 2 11│0 3 1 4 5 2 0│
└──────────┴─────────┴────────────┴─────────────┘
```

Finally the line of `AllPaths` which was used to remove loops can be adapted in a relatively unsubtle way (changes are in bold type) to detect and report them :

```
   Loops=.dyad : 0
if._=(<y){spm x do.'no path' return.end.
l=.p=.i.0 [ r=.{.y [ 'src tgt'=.yNB. initialise loop
while.(0~:#r)do.                     NB. r = incomplete
paths
  r=.r-.p [ p=.p,((tails r)e.tgt)#r      NB. p = complete paths
  r=.r extend x addnodes r       NB. add one node to each r
path
  l=.l,(#~((~:&#~.)each))r       NB. remember loops
  r=.(#~((=&#~.)each))r          NB. remove loops
end. l
)
   g1 Loops 0 2
┌─────┬───────┬─────────┬─────────┬───────────┐
│0 3 0│0 4 5 4│0 1 4 5 4│0 3 5 4 5│0 3 1 4 5 4│
└─────┴───────┴─────────┴─────────┴───────────┘
```

Code Summary

Data representations of networks, properties and conversions

```
   arcs=.0 1;0 3;0 4;1 4;2 4;3 0;3 1;3 5;4 5;5 2;5 4
   nn=.1 3 4;4;4;0 1 5;5;2 4     NB. nearest neighbor
```

```
      btoi=.# i.@#                NB. binary list to list of 1-
positions
      mton=.(btoi@:%)each@<"1     NB. matrix to n'st neighbour
      ntom=.monad :'1(;(i.n),each each y)}(n,n=.#y)$_'
                                  NB. nerest neighbour to matrix
r
   g1=.1(arcs)}6 6$_                        NB. 1 = connected, _ =
not
   wts=.2 7 6 1 5 7 6 1 4 8 4
   h1=.wts(arcs)}6 6$0            NB. weighted g1
   randg=.dyad : '%(100*x)>?(y,y)$100' NB. random g matrix
   randh=.dyad : '(*~:&_)(>:?(y,y)$100)(*&.%)x randg y'
   mksym=.(+|:) @ (*<:/~@(i.@#)) NB. make symmetric matrix
   spm=.monad : '<./y(<./ .+)^:(i.#y)y'
             NB.shortest path between pairs of g matrix
   ind=.adverh : '(i.@$*x)@]'
             NB. spm _(=&0 ind)}h1 is shtst paths in h matrix
   connected=.~:&_@spm
   reachable=.btoi each@<"1@connected
   feasible=.;@:((i.@#)(,each each)reachable)
```

Algorithhms

```
   ShtstPath1=.dyad : 0
if._=(<y){spm x do.'no path' return.end.
p=.i.0 [ r=.{.y [ 'src tgt'=.y    NB. p=paths found
while.(-.tgt e.tails r)do.        NB. r=paths ongoing
  r=.(r-.p)-.(tails p)e.src [ p=.p,((tails r)e.tgt)#r
  r=.r extend x addnodes r end.   NB. np=newpaths
r=.>(tgt=tails r)#r               NB. reject incomplete paths
)
   tails=.{:&>                    NB. tails of list of lists
   extend=.;@:(JE"1 0 each        NB. list to scalar joins
     JE=.,each &box               NB. scalarised join
     box=.]`<@.(-:>) NB. boxes if unboxed, else do nothing
     addnodes=.<@[rtoi each tails@] NB. new nodes
        rtoi=.btoi@:(%@{~)        NB. btoi after _s into 0s

   AllPaths=.dyad : 0
if._=(<y){spm x do.'no path' return.end.
p=.i.0 [ r=.{.y [ 'src tgt'=.y    NB. initialise loop
while.(0~:#r)do.                  NB. r = incomplete paths
  r=.r-.p [ p=.p,((tails r)e.tgt)#r     NB. p = complete paths
  r=.r extend x addnodes r        NB. add one node to each r
path
  r=.(#~((=&#~.)every))r          NB. remove loops
end. p
)
   unweight=.monad : '(%@~:&0)y'
   values=.+/@:([{~steps@])
      steps=.<"(1)@(2&(,/\))

   CritPath=:dyad :0              NB. critical path
if._=(<y){spm x do.'no path' return.end.
t=.( unweight x)AllPaths y NB. all paths
v=.>(<x) values each t     NB. all paths evaluated
(((=<./)v)#t),<<./v        NB. minimum value path selected
)
```

```
   Slacks=.dyad :0
if._=(<y){spm x do.'no path' return.end.
t=.(unweight x)AllPaths y
>t JE each slack >(<x)values each t
)
   slack=.-~>./

   Loops=.dyad : 0
if._=(<y){spm x do.'no path' return.end.
l=.p=.i.0 [ r=.{.y [ 'src tgt'=.y       NB. initialise loop
while.(0~:#r)do.                  NB. r = incomplete paths
  r=.r-.p [ p=.p,((tails r)e.tgt)#r     NB. p = complete paths
  r=.r extend x addnodes r        NB. add one node to each r
path
  l=.l,(#~((~:&#~.)every))r             NB. remember loops
  r=.(#~((=&#~.)every))r          NB. remove loops
end.
)
```

39. The I-spy book of J

Principal Topics : [(*left*)] (*right*), ~ (*reflex*) ciphers, public/private keys, clock multiplication, inverse, multiplicative inverse, exponential ciphers, RSA ciphers

Dear J-ohn and J-anet,

How would you like to be a security man or woman when you grow up? It's a very important job these days on account of the enormous volumes of personal and business data which fly through cyberspace every microsecond. If this is a career which attracts you, have you considered telling your teacher what a very good medium J is for getting started in this area?

Here are a few things you should know before we find you a uniform. First you should appreciate that cryptographic systems divide broadly into two categories, namely those based on **transposition** and those based on **substitution**. (I'm afraid you will have to ask your Mummy or Daddy to explain what these big words mean). In practice many current coding systems involve both of these techniques. Your first lesson will concentrate on a subset of the second of these subdivisions, that is on those in which characters are first converted to numerals and then replaced by **ciphers**. The systems concerned work to a general pattern in which there are **keys** of two kinds, **public** and **private**. I make a public key freely available to anyone who wants to send me **enciphered** messages. The relative security of any cryptographic system is proportional to the time taken by the "enemy" (that is the hackers) to determine a **private** key which allows me, and only me, to **decipher** messages. I am aware of course that the enemy will **analyse** my messages in order to try and break the code by discovering my private key. This is the process which is known as **cryptanalysis**.

The basics of such methods are simple, and J is great for describing them. My first illustration concerns multiplicative codes which depend on the clock arithmetic which you do at school. As you know, the world of sums contains only positive integers and small ones at that. Should any of these accidentally get too big for their boots, they are simply trimmed down to size by taking away the clocksize. And should one of them stray into naughty negative regions then adding the clocksize (cs) an appropriate number of times is all that is needed to bring it back into the orderly region of i.cs .

The essence of clock multiplication is the remainder verb | applied to a table based on `i.`

```
   cmultab=.|*/~@i.       NB. clock multiplication table

   cmultab 5
0 0 0 0 0
0 1 2 3 4
0 2 4 1 3
0 3 1 4 2
0 4 3 2 1
```

(If you don't like the column and row of zeros just drop them:

```
  cmtab=.| */~@ }.@:i.
  cmtab 5
1 2 3 4
2 4 1 3
3 1 4 2
4 3 2 1   )
```

The **inverse** of a clock number x is that value y for which xy=1 in clock arithmetic, so that for cs=5 the tables above show that 1 and 4 are self-inverse, and that 2 and 3 are each the inverse of the other.

Now assign a different clocksize :

```
  cs=.26
```

From this you can guess that I have an alphabetic message in mind, with letters translated into numbers in the obvious way, A=1, B=2, etc. Encryption consists of multiplying each message number by the public key, (that is one of the numbers between 1 and 25 which has no common factor with 26) and then simplifying this by clock arithmetic

```
  enc=.cs&|@*             NB. x=key, y=number
  5 enc 22 5 3 20 15 18  NB. encrypt "VECTOR"
6 25 15 22 23 12
```

To decipher this coded message I need to repeat this process, only now using the inverse of 5. The restriction put on the key in parentheses above guarantees that such a number will exist and be unique. To find the **multiplicative inverse** of a key with respect to cs, multiply the key by all the integers in the field and perform clock arithmetic using the key. The inverse is the index of whichever of these values is one:

```
   minv=.i.&1@(]|(*i.))    NB. syntax is 'key minv field'
   5 minv 26
21
```

Decipherment of ‘VECTOR’ as coded above is simply a further encryption using the inverse key:

```
  21 enc 6 25 15 22 23 12
22 5 3 20 15 18
```

I make 5 known to all my correspondents as a public key, and hence implicitly to the enemy, who is presumed to be smart enough to work out the broad method, but needs to know cs in order to discover the inverse key which turns code back into plain text. It would not be a very difficult exercise to find these quantities using the parameters above, however by using a much larger cs, and redefining the encrypting verb so that letters are dealt with in blocks, a modest degree of security can be achieved:

```
   cs=.2752
   enc=.cs&|@*

   379 enc 2205 320 1518   NB. key=379
1839 192 154
   379 minv cs             NB. multiplicative inverse of 379
1779
   1779 enc 1839 192 154   NB. decipher coded message
2205 320 1518
```

One step which could be made towards greater security is to use an *exponential* cipher rather than a multiplicative one, that is instead of multiplying the code by the key, it is raised to the *power* of the key. Uniqueness of inverse requires that cs be a **prime number**. Otherwise the only change in J terms is from * to ^ in enc :

```
  cs=.29
  15(cs&|@^)22 5 3 20 15 18
0 10 11 0 0 0
```

The zeros in the above indicate that there is a problem, namely that numbers such as 15^{22} are very large and exceed the capacity of the computer. This is easily solved since

(i) exponentiation is just repeated multiplication, and

(ii) multiplication in clock arithmetic follows the rules of multiplication in ordinary arithmetic, that is if a and b are the values of A,B and C when reduced to clock integers, then ab, if necessary reduced to a clock integer, is equal to the clock integer reduction of AB. (In mathematical terminology a=A(mod n) and b=B(mod n) implies that ab=AB(mod n)). So define clock multiplication:

```
  mul=.cs&|@*
```

and insert this into *key* replicates of the code:

```
  eenc=.mul/@#
  5 eenc &> 22 5 3 20 15 18       NB. encrypt “VECTOR”, key=5
13 22 11 24 10 15
```

The & conjunction (`&>` is equivalent to `every`) is necessary due to the non-scalar nature of the verb mul .

For the purposes of decipherment, mathematics dictates that the inverse key is the multiplicative inverse of one less than `cs`:

```
   5 minv 28                      NB. multiplicative inv. of 5
17
   17 eenc&> 13 22 11 24 10 15    NB. decipher message
22 5 3 20 15 18
```

This improves security a bit, but not by an enormous amount since an enemy with computers at his disposal would not take long to work out cs and hence, given that my key is public knowledge, to work out inv cs. A cryptographer’s Holy Grail is to find a way in which to make his key completely public so that anyone can send him messages, while at the same time making the rule for computing the decipherment key so complex that the enemy has little hope of finding it, however massive the computing power he has available. This remained an open problem in the world of cryptography until 1977 when a major breakthrough was achieved through the invention by of the so-called RSA ciphers at the Massachusetts Institute of Technology. These are named after the initials of their inventors R.L. Rivest, A. Shamir and L. Adelman.   Their idea was that cs should be the <u>product</u> of two primes, say 3551=53*67, following which any public key must then be coprime to (53-1)*(67-1)=3432, which is also the number used to calculate the multiplicative inverse. Choosing 191 as the key, and resetting the eenc verb gives:

```
   cs=.3551
   eenc=.(3551&|@*)/@#

   191 eenc every 2205 320 1518   NB. encipher “VECTOR”, key=191
489 2774 2274
   191 minv 3432                  NB. multiplicative inv. of 191
575
   575 eenc every 489 2774 2274   NB. decipher message
2205 320 1518
```

Of course the primes used in the above illustration are very small. In practice two very large prime numbers, say of the order of 10^{200},

would be chosen. Factoring products of this size is a very hard problem given the present state of the mathematical and computational arts, and so what is available to me is a public key which I can broadcast to everybody, but for which, provided I keep the two prime factors a secret, I have a private key which, ensures that only I can decipher my incoming messages.

Now, dear J-anet and J-ohn, just think how little programming you have had to do to take you from the clock arithmetic which you love to techniques which are the basis of the day-by-day encryption of millions of business and financial transmissions. Will it by any chance be one of you who cracks the factoring algorithm? (For the answer, see Vector volume 99 no. 4).

Code Summary

(Note : clocksize `cs` must be dynamically reset to an integer value).

```
cmultab=.|*/~@i.          NB. clock multiplication table
cmtab=.| */~@ }.@:i.      NB. ditto dropping zeros
enc=.cs&|@*               NB. x=key, y=number
minv=.i.&1@(]|(*i.))      NB. multiplicative inverse
mul=.cs&|@*               NB. clock multiplication
eenc=.mul/@#              NB. exponential encode
```

40. The I-spy book of J part 2

Principal Topics : `|.` (*shift*), `,` (*append*) `,.` (*stitch*) `A.` (*anagram index*), `^:` (*power* conjunction), `a.` (*alphabet*) `/:` (*grade up*) encryption, decryption, keywords, Vigenère table

Dear J-ohn and J-anet,

Now that you know a little bit about encryption and decryption, and understand the difference between public and private keys I would like to introduce you to an even simpler code which was invented by a sixteenth century French diplomat called Vigenère. The idea of a Vigenère table is that people sharing the code share a table of alphabetic characters, and for the purposes of this demonstration we will use an alphabet of just 6 characters 'ABCDES'. To make a Vigenère table choose an anagram of these 6 characters

```
      ]tkey=.(6?6){'ABCDES'
AECSDB
```

(One way to create such an anagram which makes memorization easier is to choose a keyword, say 'BED', which leads to a `tkey`= 'BEDACS' in which the characters not in the keyword are *append*ed in alphabetical order.) However it is formed, `tkey` forms the row and column headers for a table in which the first row is the tkey reversed, and the remaining rows are shifted progressively to the left :

```
   makevtab=.(1&|.)~i.@#
   (' ',tkey),.'|',.tkey,makevtab |.tkey
 |AECSDB
A|BDSCEA
E|DSCEAB
C|SCEABD
S|CEABDS
D|EABDSC
B|ABDSCE
```

This table is used for both encryption and decryption. Entering ED into the table as an example yields the letter B. Messages are converted into character pairs by the parties agreeing a message keyword. This is written by expanding it as often as necessary beneath the message, so that if the message is "Abe's Dad's a cad", and the keyword is 'SAD' write

ABESDADSACAD
SADSADSADSAD

Using the columns as indices to the Vigenère table gives the coded message

ADDEBBCABDSE

Now repeat this procedure for the coded message

ADDEBBCABDSE
. SADSADSADSAD

… and back comes the original message

ABESDADSACAD

Now we will go through all this in your favourite programming language

```
   ]vtab=.makevtab |.tkey
BDSCEA
DSCEAB
SCEABD
CEABDS
EABDSC
ABDSCE

   msg=.'ABESDADSACAD'
   msg,:'SAD'($~#)msg
ABESDADSACAD
SADSADSADSAD

   encrypt=.dyad :'(<"1 tkey i.y,.(#y)$x){vtab'
   'SAD' encrypt msg
CAABEEDCEABS
    'SAD' encrypt^:2 msg
ABESDADSACAD
```

For reassurance, try it with another keyword and another message :

```
   'BEADS' encrypt 'SEASBAD'
SSBDSAA
   'BEADS' encrypt^:2 'SEASBAD'
SEASBAD
```

Moving up to a full alphabet define `alph`, reassigning `tkey` and `vtab` and reexecuting (but not redefining) encrypt.

```
   alph=.' ',(65+i.26){a.
   tkey=.(27?27){alph
```

```
    vtab=.makevtab |.tkey=.525 A. alph
   encrypt=.dyad :'(<"1 tkey i.y,.(#y)$x){vtab'

    'VECTOR' encrypt 'QUICK A LAZY DOG'
NVNC HCYKEONDQHU
    'VECTOR' encrypt^:2 'QUICK A LAZY DOG'
QUICK A LAZY DOG  tkey=.(6?6){'ABCDES'
```

A refinement which you might like if you are to use a full alphabetic code is to have rows and columns of the Vigenère table in alphabetical order. To illustrate how to construct such a visual table, we revert to the 6-alphabet case

```
  order=.({~/:)&tkey
   ]tkey=.(6?6){'ABCDES'
SCADEB
   ok=.order tkey
ABCDES
   (' ',tkey),.'|',.tkey,makevtab |.tkey
 |SCADEB
S|BEDACS
C|EDACSB
A|DACSBE
D|ACSBED
E|CSBEDA
B|SBEDAC
   ' ',ok),.'|',.ok,(order&|:@order) vtab
 |ABCDES
A|CEASBD
B|ECBDAS
C|ABDCSE
D|SDCBEA
E|BASEDC
S|DSEACB
```

from which it is easy to confirm that these two tables are equivalent.

A Vigenère cipher can be made more secure by performing a final alphabetic shift using a further key called **shift key**. The price of this extra security is that the encryption algorithm is no longer reciprocal. However inverting the shift is simply a matter of applying minus to the shift key.

```
   shift=.dyad : '((#alph)|x+alph i.y){alph'
   skey=.3
   ]x=.skey shift 'SAD' encrypt 'QUICK A LAZY DOG'
TXPGQYI MIEDJAJC
   'SAD' encrypt (-skey)shift x
QUICK A LAZY DOG
```

Code Summary

```
makevtab=.(1&|.)~i.@#           NB. make Vigenere table
   vtab=.makevtab |.tkey        NB. vtab is a Vig.table
alph=.' ',(65+i.26){a.          NB. alphabet
tkey=.(27?27){alph              NB. anagram of alph

encrypt=.dyad :'(<"1 tkey i.y,.(#y)$x){vtab'
order=.({~/:)&tkey              NB. rows in alpha order
shift=.dyad : '((#alph)|x+alph i.y){alph'
                                NB. for extra security
```

41. Suffer the little children... to bring along their equations

Principal Topics : i. (*index of*) +. (*GCD*) -: (*halve*) {. (*head*) . congruences, clock arithmetic, inverses in finite arithmetic, GCD, simultaneous linear congruences, Chinese remainder problem, quadratic congruences, quadratic residues, square roots in finite arithmetic.

Early articles on APL sometimes speculated on how to do a work-around if one of the APL function keys was broken. Analogously one of the properties of 'clock arithmetic' as taught in the early stages of primary schools is that division is a disallowed (broken key!) operation. This reflects the historical fact that the concepts of division and fractions came relatively late in mankind's mathematical development. Despite their arithmetical skills and geometrical sophistication, the early Greek and Roman mathematicians had only crude notions of division into parts, and it was not until the invention of place value in the 11th. century that fractions in the sense we know them today became part of the earliest stages of elementary arithmetic.

J adverbs provide a natural means for realising the concepts of finite arithmetic, for example the derived verb +mod is addition in modulo arithmetic :

```
   mod=.adverb :'7&|@x'    NB. specify modulus, say 7

   3+mod 6                 NB. add 3 and 6 (mod 7)
2
   3*mod 6                 NB. multiply 3 and 6 (mod 7)
4
```

Graduating from clock arithmetic to 'clock algebra' is where some at least of the little children might begin to suffer, since division in the conventional sense is no longer an option for solving e.g. 2x=3. This is known in clock arithmetic as a **congruence** rather an equation. In modulo 5 arithmetic, it is not too difficult to spot that x=4 is a solution.

```
    mod=.1 : '17&|@x'              NB. set modulus

   (17*mod i.23)i.4                NB. locate 4 in multiples of
17
7
```

This is readily confirmed by multiplying 17 times 7 = 119 = 4 in modulo 23 arithmetic. The second line in the above J sequence suggests a

general technique for solving linear equations (congruences) of the form ax + b = 0 by first defining inverse as

```
   inv=.dyad : '(x|y*i.x)i.1'
   23 inv 17
19
```

(tacit definition enthusiasts may want to write this as

```
  inv=.i.&1@([|]*i.@[),
```

although arguably the above version is more expressive.) It is easy to confirm that in modulo p arithmetic (p prime), all integers in 1, … p-1 have an inverse, for example :

```
   iota=.>:@i.      NB. integers from 1 to y
   13|t*13 inv every t=.iota 12
1 1 1 1 1 1 1 1 1 1 1 1
```

A first try at a solution of the linear equation ax + b = 0 is then

```
   lsol=.dyad : 'x|(x inv {.y)*(-{:y)'
   23 lsol 17 _4
7
```

However, inverses exist only for numbers which are relatively prime to the modulus. In clock arithmetic terms 17x=4 is an invitation to find how many chunks of 17 steps are needed to arrive at 4, to which the answer is 3. But for 2x=5 in modulo 6 arithmetic no solution exists because 2 and 6 have a common factor, which means that only some of the clock points are reachable. On the other hand 2x=4 has two solutions, x=2, the 'obvious' one, and also x=2+3=5. To generalise this, the number of solutions of ax=b in modulo n arithmetic is either none if b is not a multiple of GCD(a,n), otherwise it is GCD(a,n), in which case these solutions are found by adding {n/GCD(a,n)} successively GCD(a,n) times to the solution of ax=b after a, b and the modulus n have all been divided by GCD(a,n) :

```
   linsol=.dyad : 0        NB. linear equation solver
if.1=gcd=.x+.{.y           NB. if a and n are co-prime ..
   do.x lsol y
elseif.0=gcd|{:y           NB. if gcd(a,n) divides b ..
   do.(m lsol y%gcd)+(m=.x%gcd)*i.gcd
end.                       NB. otherwise null result
)
   21 linsol 6 _15         NB. solns of 6x=15 in mod 21 arith
6 13 20
```

The divide symbol (`%`) appearing in the long line of `linsol` reflects the 'cancellation' of ax=b to its prime form, and does not conflict with the absence of division in clock algebra. ax=b has now been solved with complete generality.

Simultaneous Linear Congruences

Unlike ordinary simultaneous equations where, barring degeneracies, the number of equations must exactly equal the number of variables for there to be a unique solution, there is no limit to the number of simultaneous congruences for which a solution can be sought. Further, a theorem called the Chinese Remainder Theorem (so called because such results were known in China from about 100 A.D.) guarantees that provided the various moduli are coprime, then a set of simultaneous equations such as

x=0 (mod 2), x=1 (mod 5), x=2 (mod 7)

has a solution which is unique modulo the product of moduli.

The algorithm for obtaining such a solution consist of multiplying three lists :

(1) a list of the b's as in ax=b;
(2) a list of products of the a's omitting one at a time; and
(3) the inverses of the products in (2) relative to their matching moduli,

and then multiplying the items of the resulting list modulo the product of moduli. This is described as readily, and certainly more unambiguously, in J with as input

Left arg (x) : a list of moduli – n.b. these must be coprime;
Right arg (y) : a matching list of pairs of coefficients as for `linsol`.

```
   simlsol=.dyad : 0            NB. simultaneous congruences
r1=.>-@{:every y
r2=.(%~*/)x
r3=.>x linsol every <"1 r2,._1
r=.(*/x)|+/r1*r2*r3
)
```

The solution of the above set of congruences is :

```
   2 5 7 simlsol 1 0;1 _1;1 _2
```

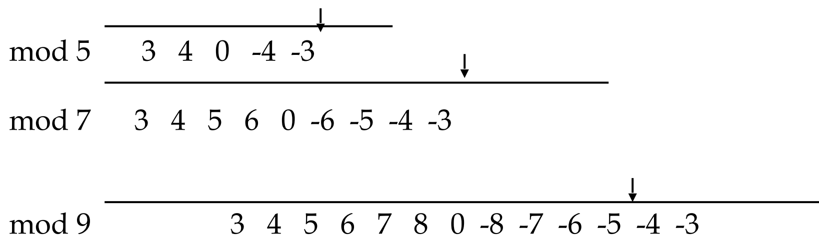
```
16
```

For those who like puzzles `simlsol` lends itself to solutions of problems such as what is the smallest integer divisible by 7 whose remainders on division by 2,3,4,5 and 6 are 1,2,3,4 and 5? what is the smallest integer divisible by 7 whose remainders on division by 2,3,4,5 and 6 are all 1?

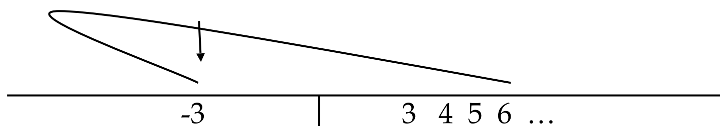
```
   4 3 5 7 simlsol 1 _3;1 _2;1 _4;1 0
119
   4 3 5 7 simlsol 1 _1;1 _1;1 _1;1 0
301
```

Quadratic Congruences

Here the simplest case is x^2=a. In ordinary arithmetic there are two solutions, namely ± the square root of a, and it is useful to picture how finite arithmetics converge towards normal arithmetic as the modulus increases towards infinity. Represent say 3 up to -3 by points on number lines corresponding to arithmetics with successively larger moduli :



Eventually the arrow ‘goes off to infinity’ and returns ‘on the other side of zero’ as -3 in the conventional sense :



Returning to the problem of solving $x^2=a$ in modulo n arithmetic, Since only integers are admissible, there can only be solutions if a is one of those integers in 1,..,n-1 which are squares in modulo n, and since $k^2 = (-k)^2$ it is only necessary to consider the range 1,.. ½(n-1) in order to establish all such squares. These are called **quadratic residues** in finite arithmetics, and are obtained as :

```
   qres=.|*:@:(iota@(-:@<:))      NB. quadratic residues
   qres 13                        NB. squares modulo 13
1 4 9 3 12 10
   qres 17                        NB. squares modulo 17
1 4 9 16 8 2 15 13
   qres 29                        NB. squares modulo 29
1 4 9 16 25 7 20 6 23 13 5 28 24 22
```

so, for example, in modulo 13 arithmetic, the pairs of square roots of 3, 12 and 10 are (4,13-4), (5,13-5) and (6,13-6), i.e. (4,9), (5,8) and (6,7) respectively, and a must be one of the six values `qres 13` if the equation $x^2=a$ is to have a solution. One such solution is then `(qres n)i.a` so, for example one solution of $x^2=5$ in modulo 29 arithmetic is

```
   1+(qres 29)i.5
11
```

and the other is 18, which is confirmed by observing that both 121 and 324 equal 5 in modulo 29 arithmetic. This leads to the following defin-

ition of a verb which delivers a 'single square root' verb in finite arithmetic :

```
   sqrt=.>:@(qres@[i.])  NB. sqrt of y in modulo x
   13 sqrt 12
5
```

This can readily be generalised to find any root :

```
   res=.[ | iota@(<:@[) ^ ] NB.generalised residue
   13 res 3                 NB.cubes in modulo 13
1 8 1 12 8 8 5 5 1 12 5 12

   iall=.>:@(= # i.@#@[)    NB.iota all (origin 1)

   root=.(({.res{:)@[)iall] NB.all kth. roots, e.g. …
   13 3 root 12            NB.cube roots of 12 mod 13
4 10 12
```

Read the above lines as "in modulo 13 arithmetic, the 3-roots (i.e. cube roots) of 12 are 4, 10 and 12"

The suite of verbs `res`, `iota`, `iall` and `root` makes `qres` and `sqrt` redundant, and allows the solution of any equation $x^n=a$. Where no solution exists a null result is returned.

Now turn to the solution of the more general quadratic $ax^2 + bx + c = 0$. In ordinary arithmetic, the solution is found by the technique of completing the square to give the standard formula $(-b\pm\sqrt{(b^2-4ac)})/2a$ with 2a as the denominator. With division disallowed, the trick is to multiply the left hand side by 4a to obtain a leading term $(2a)^2x^2$ and factorise

$$4a(ax^2 + bx + c) \text{ as } (2ax + b)^2 - (b^2 - 4ac).$$

Then write $d = b^2 - 4ac$ and $y = 2ax + b$, so that $ax^2 + bx + c = 0$ becomes $y^2=d$ which has already been solved provided that d is one of the quadratic residues. If not, there are no solutions. So define the verb `disc` standing for 'discriminant' to compute $b^2 - 4ac$ in the ordinary way, so that the discriminant of e.g. $5x^2 - 6x + 2$ is -4 :

```
   disc =.(*:@(1&{))-4&*@{.*{:
   disc 5 _6 2
_4
```

`disc`, like other verbs, can be modified with the adverb `mod` using the current modulus `n` :

```
   disc mod 5 _6 2                NB. n is currently 13
9
```

so that the first step in the solution is

```
   13 2 root disc mod 5 _6 2      NB. sq roots of 9 mod 13
3 10
```

All that remains is to transform these two solutions in y's back to x's, specifically solve 10x – 6 = 3 and 10x – 6 = 10 (the latter being equivalent to 10x – 6 = –3), that is the two linear equations 10x – 9 = 0 and 10x – 16 = 0 for which a technique is already available :

```
   13 linsol every 10 _9;10 _16
10 12
```

solutions which are confirmed by

```
   10 12 #.mod every< 5 _6 2
0 0
```

The technique is consolidated in the verb

```
   qsol=.dyad : 0                 NB. quadratic solver
t=.(n,2)root disc mod y [ n=:{.x
n linsol&><"1(2 1*}:y)+"1(0,&>t)
)
   13 qsol 5 _6 2
12 10
```

and confirmation is obtained by

```
   (13 qsol 5 _6 2)#.mod&>< 5 _6 2
0 0
```

Not all quadratics have genuine solutions, and the simplest way to proceed is to execute `qsol` regardless but disregard any solutions which fail the confirmation test above. This leads to a general quadratic solver :

```
   quadsol=.dyad : 0       NB. general quadratic solver
t=.(n=:x)qsol y
if.0 0-:t#.mod&><y do.t
else. i.0 end.
)
   13 quadsol 5 _6 2
12 10
   13 quadsol 5 6 _2       NB. change of coefficients
```
(null result)

the results of which can be checked by

```
   (13 quadsol 5 _6 2)#.mod&>< 5 _6 2
0 0
   (13 quadsol 5 6 _2)#.mod&>< 5 6 _2
```
(null result)

Thus a single session of algebra has provided solutions for all linear and quadratic equations in countless algebras, and all with quite a modest amount of suffering!

Code Summary

(Note : The modulus at any point depends dynamically on the setting of the session variable `n` as in the definition of `mod`).

Finite Arithmetic

```
   mod=.adverb :'n&|@x'      NB. specify modulus, e.g. by n=.7
   inv=.dyad : '(x|y*i.x)i.1' NB. multilipcative inverse
```

Linear Equation Solver

```
   linsol=.dyad : 0        NB. linear equation solver
if.1=gcd=.x+.{.y           NB. if a and n are co-prime ..
   do.x lsol y
elseif.0=gcd|{:y           NB. if gcd(a,n) divides b ..
   do.(m lsol y%gcd)+(m=.x%gcd)*i.gcd
end.                       NB. otherwise null result
)
  lsol=.dyad : 'x|(x inv {.y)*(-{:y)'
```

Simultaneous Linear Congruences

```
   simlsol=.dyad : 0             NB. simultaneous congruences
r1=.>-@{:every y
r2=.(%~*/)x
r3=.>x linsol every <"1 r2,._1
r=.(*/x)|+/r1*r2*r3
)
```

Quadratic Residues

```
   qres=.|*:@:(iota@(-:@<:))      NB. quadratic residues
   sqrt=.>:@(qres@[i.])           NB. sqrt of y in modulo x
```

Quadratic Equation Solver

```
iota=.>:@i.

   quadsol=.dyad : 0                    NB. general quadratic
solver
t=.(n=:x)qsol y
```

```
if.0 0-:t#.mod&><y do.t
else. i.0 end.
)
   qsol=.dyad : 0                NB. quadratic solver
t=.(n,2)root disc mod y [ n=:{.x
n linsol&><"1(2 1*}:y)+"1(0,&>t)
)
disc =.(*:@(1&{))-4&*@{.*{:
  root=.(({.res{:)@[)iall]             NB.all kth. roots
   res=.[ | iota@(<:@[) ^ ]       NB.generalised residue
       iota=.>:@i.                NB. integers 1 to y
       iall=.>:@(= # i.@#@[)      NB.iota all (origin 1)
        disc =.(*:@(1&{))-4&*@{.*{:
```

42. Fifty ways to tell a fib

Principal Topics : ^: (*power* conjunction), " (*rank* conjunction \ (*prefix / infix*) /. (*oblique*) ~ (*passive / reflex*) Fibonacci numbers, Lucas numbers, binomial coefficients, Pascal triangle, Binet formula, continued fractions.

The Fibonacci series is a bit like fly-paper or goose-grass, once it begins sticking to you, it is terribly hard to get rid of it. I will start by defining the first 14 terms which should be enough to observe the patterns which evolve. (Note : because I am treating only a finite part of the series there will be end effects which could be resolved by topping and tailing, but this usually serves only to obscure what is important, so please just ignore end effects.)

```
   f=.(,+/@(_2&{.))^:12(0 1)
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

Incrementing the cumulative sums and differences still leaves us in Fibonacci-land :

```
   >:+/\f                     NB. 2|.f
1 2 3 5 8 13 21 34 55 89 144 233 377 610
   >:-/\f                     NB. alternating differences(6)
1 0 1 _1 2 _3 5 _8 13 _21 34 _55 89 _144
```

Here is a selector verb which takes a binary pattern as left argument and extends it as a mask for the right argument. Its first use is to select odd and even items in `f` :

```
   sel=.($~#)#]
   ]od=.0 1 sel f                    NB. odd items in f
1 2 5 13 34 89 233
   ]ev=.1 0 sel f                    NB. even items in f
0 1 3 8 21 55 144
```

Cumulating either odds or evens leads to the other :

```
   +/\od                             NB. 1|.ev    (4)
1 3 8 21 55 144 377
   >:+/\ev                           NB. 1|.od    (5)
1 2 5 13 34 89 233
```

and the Fibonacci trade mark is still there if we do the following :

```
   +/\*:f                            NB. scan the squares of f (9)
0 1 2 6 15 40 104 273 714 1870 4895 12816 33552 87841

   (*1&|.)f                          NB. multiply adjacent terms
0 1 2 6 15 40 104 273 714 1870 4895 12816 33552 0
   2+/\f                             NB. result is 2|.f
1 2 3 5 8 13 21 34 55 89 144 233 377
   2-~/\f
1 0 1 1 2 3 5 8 13 21 34 55 89
   _2+/\f                            NB. same as }.ev
1 3 8 21 55 144 377
```

```
   2*/\f                          NB. products in pairs
0 1 2 6 15 40 104 273 714 1870 4895 12816 33552
   +/\2*/\f                       NB. cum sums of prods in pairs
0 1 3 9 24 64 168 441 1155 3025 7920 20736 54288
```

The product pairs of consecutive items in the Fibonacci series generate another derived series with some interest in its own right :

```
   2*/\ f
0 1 2 6 15 40 104 273 714 1870 4895 12816 33552
```

Its cumulative series is :

```
   ]cpp=.+/\2*/\f                 NB. cum product pairs
0 1 3 9 24 64 168 441 1155 3025 7920 20736 54288
```

Compare :

```
   0 1 sel cpp                    NB. odd items in ccp
1 9 64 441 3025 20736
   ev^2
0 1 9 64 441 3025 20736

   1 0 sel cpp                    NB. even items in cpp
0 3 24 168 1155 7920 54288
   1 0 sel(*2&|.)f                NB. prods of pairs but one
0 3 24 168 1155 7920 0

   2+/\2*/\f                      NB. add adj product pairs
1 3 8 21 55 144 377 987 2584 6765 17711 46368
   (2*/\f)+1|.2*/\f               NB. 1|.ev   (as above)
1 3 8 21 55 144 377 987 2584 6765 17711 46368 33552
```

At this point switch to a lesson in J style :

```
   ((+ 1&|.)@(2&(*/\)))f          NB. better style for above

   pp=.2&(*/\)                    NB. third version
   ((+1&|.)@pp)f
1 3 8 21 55 144 377 987 2584 6765 17711 46368 33552
```

Add products in pairs and you get the evens

```
   1 |.ev
1 3 8 21 55 144 0
```

Subtract succesive products in pairs and you get the odds which are also the squares of f :

```
   2-~/\2*/\f               NB. cf.0 1 sel+/\2*/\f
1 1 4 9 25 64 169 441 1156 3025 7921 20736
```

What about succesive sums of three or more ? :

```
   -:3+/\f                        NB. 2|.f
1 2 3 5 8 13 21 34 55 89 144 233
```

```
   4+/\f
4 7 11 18 29 47 76 123 199 322 521
```

These numbers which arise from applying the Fibonacci rule starting with 1 3 are known as the Lucas numbers, and are generated directly by

```
   ]f=.(,+/@(_2&{.))^:10(1 3)     NB. Lucas numbers
1 3 4 7 11 18 29 47 76 123 199 322 521 843

   |>:-/\f                        NB. _1|f
1 0 1 1 2 3 5 8 13 21 34 55 89 144
   |2-/\f                         NB. _1|.f
1 0 1 1 2 3 5 8 13 21 34 55 89
   -:3-/\f                        NB. f
0 1 1 2 3 5 8 13 21 34 55 89

   |4-/\f                         NB. _1|.Lucas nos.
2 1 3 4 7 11 18 29 47 76 123

   }.*:f                          NB. f squared
1 1 4 9 25 64 169 441 1156 3025 7921 20736 54289
   (*2&|.)f                       NB. f * 2|.f
0 2 3 10 24 65 168 442 1155 3026 7920 20737 0 233

   v1=.1&|.@:*:      NB. shift squares
   v1 f
1 1 4 9 25 64 169 441 1156 3025 7921 20736 54289 0
   v2=.*2&|.         NB. prods next but one (hook)
   v2 f
0 2 3 10 24 65 168 442 1155 3026 7920 20737 0 233
   (v1-v2)f          NB. differences
1 _1 1 _1 1 _1 1 _1 1 _1 1 _1 54289 _233


  +//.!/~i.10        NB. oblique sums of Pascal tri.
1 1 2 3 5 8 13 21 34 55 88 133 176 189 155 92 37 9 1


   +/\0 0 1 sel f   NB. cum sum of 3rd 6th, 9th …
1 6 27 116
   -:<:0 1 0 sel f  NB. half of u sub(3n+2) -1
0 1 6 27 116
```

Relationship with binomial coefficients

```
   +//.(!/~)i.10
1 1 2 3 5 8 13 21 34 55 88 133 176 189 155 92 37 9 1
```

As an aside the following gives the first four Pascal triangles as a rank 3 array :

```
   a=.(!/~)\i.4

   +/"2 a NB. add cols of successive Pascal tria.
1 0 0 0
1 2 0 0
1 2 4 0
```

```
1 2 4 8

   ]gs=.(1&{::@p.)1 1 _1   NB. Binet''s formula
1.61803 _0.618034
   */gs
_1

   ]c=.0 1%.1,:gs                  NB. solve for closed form
0.447214 _0.447214

   +/"1 c *"1(<gs)^&> i.10
1.11022e_16 1 1 2 3 5 8 13 21 34 55 89
```

or equivalently, because the larger root rapidly dominates :

```
   rnd=.<.@(0.5&+)
   rnd(({.gs)^i.15)%%:5
0 1 1 2 3 5 8 13 21 34 55 89 144 233 37
```

It also allows the generation of indefinitely large Fibonacci numbers :

```
   (9!:11)16

   c +/ .*gs^78
8944394323791464
```

Divisibility properties :
Demonstration that even Fibonacci numbers have indices divisible by 3, all the others are odd :

```
   1 0 0 sel f
0 2 8 34 144
   0 1 1 sel f
1 1 3 5 13 21 55 89 233
```

This can be extended by generalising the selection verb to provide complementary selections :

```
   dpsel=.(0&=;0&~:)@:i.
   dpsel 4
┌───────┬───────┐
│1 0 0 0│0 1 1 1│
└───────┴───────┘
   (dpsel 4)sel&.><f
┌──────────┬─────────────────────────┐
│0 3 21 144│1 1 2 5 8 13 34 55 89 233│
└──────────┴─────────────────────────┘
   (dpsel 5)sel&.><f
┌──────┬──────────────────────────────┐
│0 5 55│1 1 2 3 8 13 21 34 89 144 233 │
└──────┴──────────────────────────────┘
   (dpsel 6)sel&.><f
┌───────┬─────────────────────────────┐
│0 8 144│1 1 2 3 5 13 21 34 55 89 233 │
└───────┴─────────────────────────────┘
```

```
   F=.(,+/@(_2&{.))^:60(0 1)
   17((=&0@:|)#])F
0 34 2584 196418 14930352 1134903170 86267571272
```

Continued fractions

Continued fractions are defined by

```
   cf=.1&+@%
   cf^:(i.11)1
1 2 1.5 1.6667 1.6 1.625 1.6154 1.619 1.6176 1.6182 1.618
```

The relationship to the Fibonacci numbers should be clear from

```
   (}.f)*cf^:(i.13)1
1 2 3 5 8 13 21 34 55 89 144 233 377
```

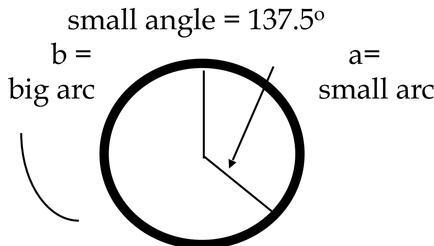
Code Summary

```
   f=.(,+/@(_2&{.))^:12(0 1)      NB. first 12 Fib numbers
   od=.0 1 sel f                  NB. odd items in f
   ev=.1 0 sel f                  NB. even items in f
      sel=.($~#)#]
   cpp=.+/\2*/\f                  NB. cum product pairs
   pp=.2&(*/\)                    NB. product pairs
   gs=.(1&{::@p.)1 1 _1           NB. Binet''s formula
   rnd=.<.@(0.5&+)                NB. round to integer
   dpsel=.(0&=;0&~:)@:i.          NB. complementary selections
   cf=.1&+@%                      NB. continued fractions
```

43. Seeds, Cones and Sunflowers

Principal Topics : Fibonacci sequence, golden ratio, golden angle, spirals

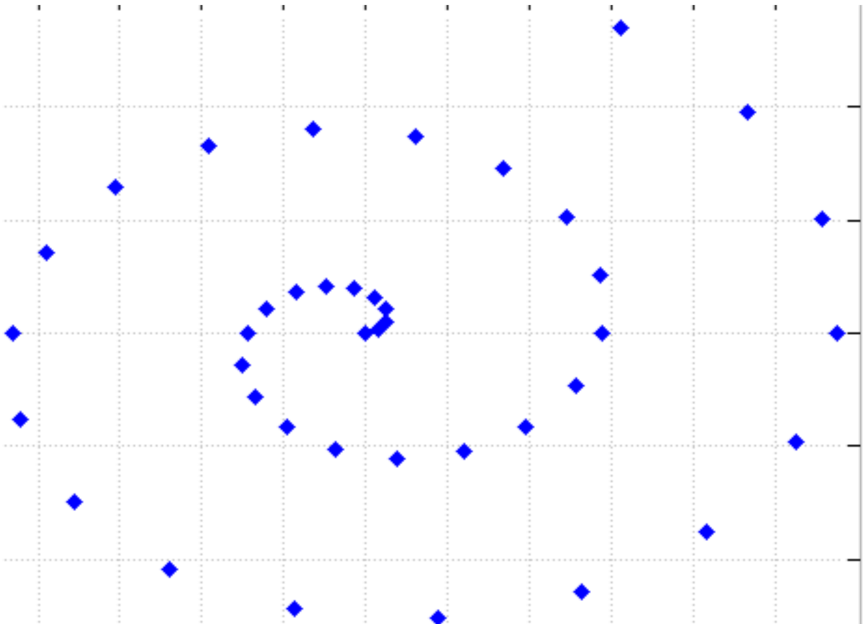
It is well known that the Fibonacci sequence 0,1,1,2,3,5,8,13,21,34, ... in which each number is the sum of its two predecessors exhibits itself in nature in e.g. the arrangements of seeds in the head of a sunflower and the pattern of scales on a cone. In architecture and design it is widely claimed that the most 'pleasing' form of rectangle is one in which the ratio of its side approximates to 0.61818... which is the limiting value of the ratio of consecutive values of terms in the series, viz. 2/3, 3/5, 5/8, and so on. If a and b are the lengths of the shortest and longest sides respectively, this limiting value, known as the *golden ratio*, is that value of a/b, which is also equal to the ratio b/(a+b). An equivalent problem is that of dividing the circumference of a circle into two arcs a and b with a/b=a/(a+b). This is attained when the radius to the dividing point is at an angle, analogously called the *golden angle*, which is very close to 137.5 ^{o} degrees (more accurately 137.5078 ^{o} but for practical purposes 137.5 ^{o} is adequate.) This can readily be checked by observing that 137.5/222.5 = 222.5/360 = 0.618 to 3 significant figure precision.



Why is the golden angle so important in natural growth? Start by looking from above directly down the axis of a growing shoot such as a cone, and projecting the growth points onto a two dimensional plane spiralling outwards, with each new scale emerging at a regular angular displacement. *Primordia* is a generic noun used to cover the growth embryos characteristic of many plants such as leaves, sepals, florets, etc. It often happens that the angular displacement of successive primordia is the golden angle of 137.5^{o}. Its effect is observable in the series of spirals going in opposite directions which are striking in both the seeds of sunflowers and the scales of cones such as the spruce cone below in which the junction of a steep and a gentler spiral is highlighted.

Use J to display and experiment with spirals :

```
   evry=.4 : '(0=y|(i.@#)x)#x'

   dtor=.180%~o.                  NB. degrees to ra-
dians

spiral=.dyad :0
  ang=.dtor 360| y*i=.i.5*x^2          NB. primor-
dial angles in rads
  t1=.i*"(1)1 2 o.every<ang       NB. converted to x-
y coords
'marker; labels 0' plot <"1 t1 evry"(1)x    NB. plot
every xth.
)

   8 spiral 137.5
```



What causes these spirals to be so readily observable? Consider that every third primordium is a member of a spiral sequence, so that, subtracting out multiples of 360°, the third, sixth, ninth, twelfth and fifteenth primordia develop at angles of

```
   360|137.5*(i.20)evry 3
0 52.5 105 157.5 210 262.5
```

degrees forming what will be called a 3-spiral. The difference of 52.5 between successive terms makes the spiral easy to spot by eye. Analogously a 5-spiral begins with the fifth, tenth and fifteenth primordium. These values are

```
   360|137.5*(i.20)evry5
0 327.5 295 262.5
```

and the differences of 32.5 are again easy for the eye to spot. Since the fifth angle in the 3-spiral and the third angle in the 5-spiral both correspond to the 15th primordium these values necessarily coincide.

Now extend the 5-spiral to eight terms and the 8-spiral to five terms

```
   360|137.5*(i.45)evry 5
0 327.5 295 262.5 230 197.5 165 132.5 100
```

```
   360|137.5*(i.48)evry 8
0 20 40 60 80 100
```

Arguing as above, these necessarily coincide in their final items which correspond to the 40th primordium. The difference of 20 between successive terms of the 8-spiral make it easy to pick out like the 3- and 5-spirals.

Now use J to generate a plots of these spirals, extending `spiral` to show a pair of spirals.

```
   spi2=.dyad :0
ang=.dtor 360| y*i=.i.2*>:*/x
x1=.(2 o.ang)*i  [ y1=.(1 o.ang)*i
x3=.x1 evry{:x  [ x2=.x1 evry{.x
y3=.y1 evry{:x  [ y2=.y1 evry{.x
'marker; labels 0'plot (x2,:x3);(y2,:y3)
)
```

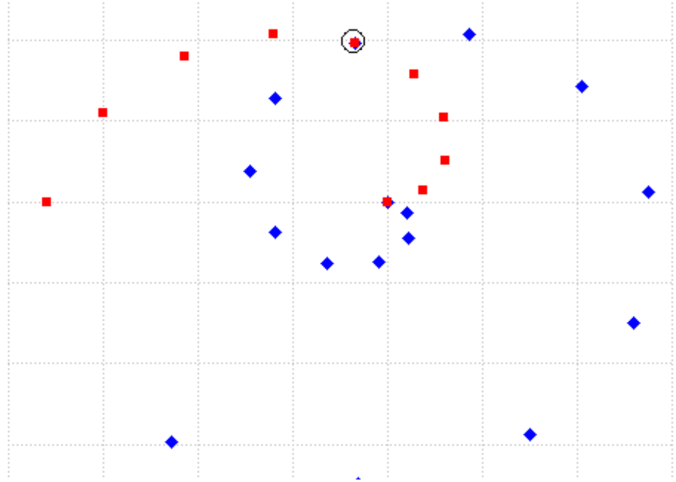
(Note: There is a temptation akin to the 'one-liner' APL phenomenon to practise some rank artistry to the above, viz.

```
  Evry=.4 :'(+./0=y|every<(i.@#)x)#x'
spirals=.dyad :0
  ang=.dtor 360 |y*i=.i.>:*/x         NB. primor-
dial angles in rads
  t1=.i*"(1)1 2 o.every<ang      NB. convert to
cartesian coords
'marker; labels 0' plot <"1 t1 Evry"(1)x
)
```

but for me the relative clarity of the first version makes it preferable.)

The plot below shows a 5-spiral radiating out anticlockwise from the centre and an 8-spiral radiating clockwise with the two meeting at the 40th primordium at the circled point at the top centre of the plot.

```
    5 8 spi2 137.5
```



The area bounded by the two paths to the circled point is a two-dimensional representation of spiral patterns which stand out on the pine cone below :

Next consider the first eight terms of the 13-spiral which are

```
   360|137.5*(i.105)evry 13
0 347.5 335 322.5 310 297.5 285 272.5 260
```

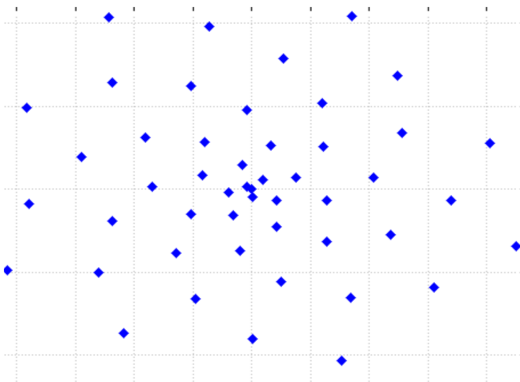
Following the reasoning above, 260 must be the value of the thirteenth term of the 8-spiral. The small value, -12.5, of the differences in the 13-spiral, make it readily observable. Also while the differences for the 3- and 5-spirals are positive, those for the 8- and 13-spirals are negative. This means that when successive spirals in the 3, 5, 8, 13

sequence meet at their common point, the 104th primordium, this will be the junction of two spirals going in opposite directions.

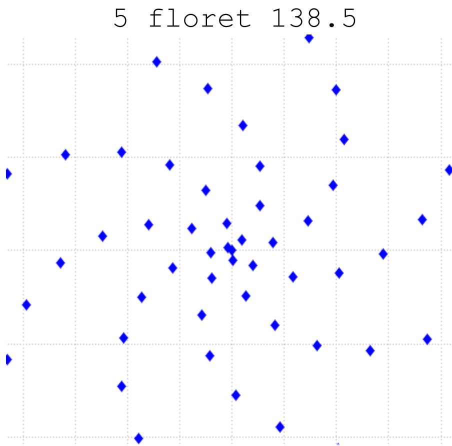
Fibonacci numbers are instantly recognisable in the above discussion. These Fibonacci based spirals are the only ones where the differences are sufficiently small for the spirals to be eye-catching. For example the differences for the 4-, 6-, 7- and 9-spirals are -170, 105, -117.5, and 157.5. In the context of the cone this would mean that after at most two scales, the observer would have to turn the cone over repeatedly to trace the spiral. Also with higher order Fibonacci numbers the differences shrink towards zero, for example for the 233-spiral the difference is less than one degree. This corresponds to the manner in which the Fibonacci sequence itself converges to the golden ratio. Of course not many plants manage to generate 233 primordia on a single axis! The golden angle has a a further property, namely that it is that angle which brings about the most efficient use of the circular plane of the flower-head, in other words which packs the seeds together as closely as possible.

The verb `floret` and plot below illustrates how the seeds position themselves given a primordial angle of 137.5°:

```
floret=.dyad :0
  ang=.dtor 360 | y*i=.i.10*>./x
  x1=.(2 o.ang)*i  [ y1=.(1 o.ang)*i
  'marker; labels 0'plot x1;y1
)
   5 floret 137.5
```



If the primordial angle were to change to 138.5 ^{o} spaces between the spirals begin to be apparent :



If nature had chosen 135 ^{o} all the seeds would eventually be laid on the radial zero axis, leaving most of the flower-head as empty space!

The above discussion demonstrates why simpler spiral sequences are readily observable in scales, petals, flower-heads and so on, but does not address the more fundamental question of *why* primordia pop out at the golden angle with such regularity. Amazingly, although everything that has gone before has been known and observed for centuries, it is only within the last thirty years or so that mathematicians have developed *explanatory* models which show that these patterns are *inevitable* if it assumed that developing primordia compete for space behave like identical atoms emitting mutually repelling electrical charges. Each sunflower seed for exampIe behaves in its own self-interest, and the result is an equilibrium state which requires complex mathematics to work out from first principles. It seems a logical conclusion that every sunflower seed emerges as if had solved all these equations in the moment it bursts out from the bud. Or was the formula solved once and for all back in distant aeons and transmitted through the plant's DNA, so that each seed knows at birth exactly the spot to go to on the flower-head? Nature constantly stretches man's powers of wonderment. The patterns of 3. 5, 8, 13 etc. are elementary manifestations of much deeper matters which are only beginning to be understood. How nature must scorn mankind's as yet primitively simple mind; thIs is not just intelligent design, but super-, even super-super-intelligent design. Every glance at a sunflower, cone or floret should thus bring out profound sense of humility in the observer!

Code Summary

```
   evry=.4 : '(0=y|(i.@#)x)#x'
   dtor=.180%~o.                    NB. degrees to ra-
dians
spiral=.dyad :0
  ang=.dtor 360| y*i=.i.5*x^2          NB. primor-
dial angles in rads
  t1=.i*"(1)1 2 o.every<ang      NB. converted to x-
y coords
'marker; labels 0' plot <"1 t1 evry"(1)x    NB. plot
every xth.
)
spi2=.dyad :0
  ang=.dtor 360| y*i=.i.2*>:*/x
  x1=.(2 o.ang)*i  [ y1=.(1 o.ang)*i
  x3=.x1 evry{:x  [ x2=.x1 evry{.x
  y3=.y1 evry{:x  [ y2=.y1 evry{.x
'marker; labels 0'plot (x2,:x3);(y2,:y3)
)
floret=.dyad :0
  ang=.dtor 360 | y*i=.i.10*>./x
```

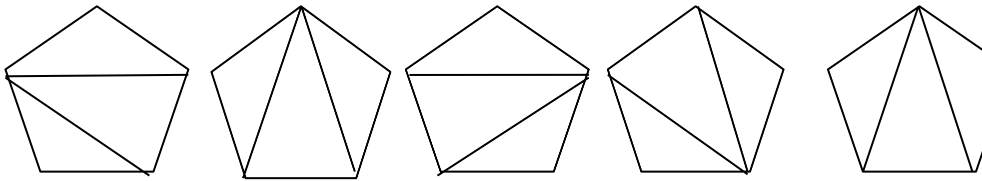
```
  x1=.(2 o.ang)*i  [ y1=.(1 o.ang)*i
  'marker; labels 0'plot x1;y1
)
```

44. Catalan Numbers

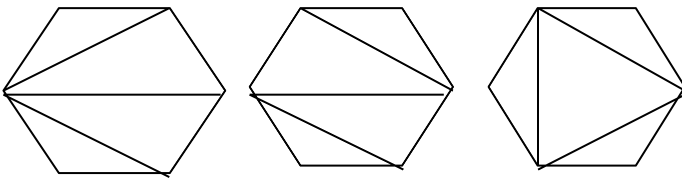
Principal Topics : Catalan numbers, permutations, combinations, Manhattan diagram, binary trees

The Catalan numbers, although not as universally well-known as the Fibonacci sequence, arise in a surprisingly disparate number of counting situations. They have nothing to do with Catalonia – they were known to the Chinese, but first discovered in Europe by Euler, and named after Eugène Catalan who elaborated on them in a paper in the 1830s. First here are a few problems for which they are relevant :

Some counting problems
(1) Given a polygon with n sides, into how many triangles can its area be divided by connecting vertices with non-crossing line segments. If n=4, the answer is readily seen to be 2, if n=5, the answer is 5 obtained by drawing two internal rays from each of the vertices in turn :



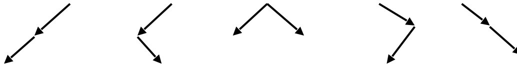
The n=6 case is a little more complex and just three of the 14 possibilities are shown.



(2) Given a list of n numbers to be totalled, how many different ways of subtotalling are there, for example for an n-list with n=4 the possibilities are :

((a+b)+c)+d (a+(b+c))+d (a+b)+(c+d) a+((b+c)+d) a+(b+(c+d))

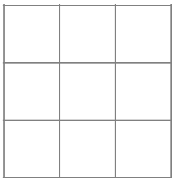
(3) How many binary trees are there with n branches which do not end in a leaf. For example for n=2 the possible trees are (n.b. leaves when reached are not shown) :



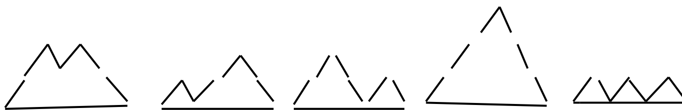
(4) How many ways can n pairs of parentheses be written in such a way that no right parenthesis appears before its matching left parenthesis, for example with n=3 pairs

((( )))   (( )) ( )   (( ) ( ))   ( ) (( ))   ( ) ( ) ( )

(5) In an n by n 'Manhattan diagram' how many different <u>progressive</u> routes are there from bottom left to top right which do not cross the diagonal but may be on either side of it. One such route is



(6) How many different 'mountain ranges can be formed given n slopes, e.g. for n=6 the possibilities are



The Catalan Sequence

The Catalan numbers can be evaluated by a simple formula (2n!)/[(n+1)! n!], and so the first twelve Catalan numbers along with their indices are

```
   cat=.monad : '(!+:y)%(*/! every y,y+1)'
   |:(i.12),.cat every i.12
0 1 2 3  4  5   6   7    8    9    10    11
1 1 2 5 14 42 132 429 1430 4862 16796 58786
```

The ratio of the (n+1)th number to the nth is 2(2n+1)/(n+2) which for large values of n approaches 4.

The procedure for obtaining the next Catalan number is similar to, but slightly more elaborate than, that for the Fibonacci sequence. Based on the hook `(,:|.)` the rule is 'take a sequence and its reverse, multiply corresponding pairs and add' :

```
    ]u=.(,:|.)t=.cat every i.5
 1 1 2 5 14
14 5 2 1  1
   +/*/u
42
```

Another interesting property of Catalan sequences involves determinants and is illustrated by the sequence below.

```
   cat every i.4
1 1 2 5

   (i.4)+every <i.4     NB. construct a matrix of indices
...
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6

   box=.<"1
   (cat every) every box (<i.4)+every i.4
1  1  2   5
1  2  5  14
2  5 14  42
5 14 42 132
```

So define :

```
   catmat=.monad : 'cat every every box (<i.y)+every i.y
   det=.-/ .*
   det catmat 4
1
```

Now generalise :

```
   catdet=.monad : 'det catmat y'
   catdet every i.12
1 1 1 1 1 1 1 1 1 1 0.999957 0.99893
```

The last two items reflect the rapid growth in size of the Catalan numbers as the sequence progresses.

Relation to Permutations and Combinations
J conveniently gives a list of all permutations of a given order by

```
   allperms=.i.@! A. i.
   |:allperms 4
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
1 1 2 2 3 3 0 0 2 2 3 3 0 0 1 1 3 3 0 0 1 1 2 2
2 3 1 3 1 2 2 3 0 3 0 2 1 3 0 3 0 1 1 2 0 2 0 1
3 2 3 1 2 1 3 2 3 0 2 0 3 1 3 0 1 0 2 1 2 0 1 0
```

(Note : the transpose `|:` is applied for compactness of display.)

Catalan permutations are those in which there are no sub-lists of length 3. A first step in extracting these is

```
   incseq=.*./@(_1&}.)@:(< 1&|.)    NB. 1 if list y
strictly increasing
   incseq every 2 3 4 5;2 5 4 3
1 0
```

The items which form a three-sequence in a permutation need not be consecutive, so for any permutation, all possible 3-lists must be obtained, e.g. 1 4 2 3 contains the strictly increasing sub-list 1 2 3. Testing for all 3-lists within a given permutation therefore requires a verb to give all combinations of r items out of n.

```
   cnos=.i.@:(2&^)@]          NB. integers from 0 to 2^n
-1
   bins=.#:@cnos              NB. binary nos from 0 to
2^n-1
   mark=.|.@((= +/"1@bins) # cnos) { bins  NB. 1=include
in combn
   combs=.mark #"1 i.@]       NB. transform to combina-
tions
   |:3 combs 4
0 0 0 1
1 1 2 2
2 3 3 3
```

The ‘no ascending-3-list’ is then given by

```
   tri=.monad : '-.+./incseq every box (box 3 combs #y)
{ every <y'
   tri every 1 4 2 3;3 4 1 2
0 1
   |:(tri every box t)#t=.allperms 3            NB. 5
columns
0 1 1 2 2
```

```
2 0 2 0 1
1 2 0 1 0
   |:(tri every box t)#t=. allperms 4     NB. 14 columns
0 1 1 1 2 2 2 2 2 3 3 3 3 3
3 0 3 3 0 1 1 3 3 0 1 1 2 2
2 3 0 2 3 0 3 0 1 2 0 2 0 1
1 2 2 0 1 3 0 1 0 1 2 0 1 0
```

Returning to the six problems
The solutions are now seen to be

(1) cat(n-2), (2) cat(n-1), (3) cat(n+1), (4) cat(n), (5) 2cat(n), (6) cat(n/2).

The correspondences between (2) and (3), and between (4), (5) and (6) are relatively easy to observe. In the case of (4) write L for left parenthesis, R for right parenthesis. The number of valid pairings is thus the number of possible words such as LLRRLRLLLRLRRR starting with L in which n= the equal numbers of Ls and Rs. To count these introduce a 'false' L as first character to ensure validity of the bracketing represented. The number of words in the enhanced set is thus the number of ways in which n like items can be chosen from 2n+1 which is ${}^{2n+1}C_n$ = ((2n+1)!)/[((n+1)!)(n!)] divided by (2n+1), since the first character is pre-determined. This simplifies to the formula ${}^{2n}C_n$/(n +1) as given above.

Code Summary

```
   cat=.monad : '(!+:y)%(*/! every y,y+1)'       NB. Cata-
lan numbers
   allperms=.i.@! A. i.                          NB. List
of all perms
   perms=.dyad :'~.x{.&.|: allperms y'           NB. All
y-perms from x
   combs=.mark #"1 i.@]                          NB. All
y-combns from x
     mark=.|.@((= +/"1@bins) # cnos) { bins      NB. 1=in-
clude in combn
       cnos=.i.@:(2&^)@]                   NB. integers
from 0 to 2^n -1
       bins=.#:@cnos                       NB. binary nos
from 0 to 2^n-1
   incseq=.*./@(_1&}.)@:(< 1&|.)           NB. 1 if list
strictly increasing
   tri=.monad : '-.+./incseq every box (box 3 combs #y)
{ every <y'
                                           NB. Catalan
perms of order y
       box=.<"1
   catmat=.monad :'cat every every box (<i.y)+every i.y
                                           NB. Catalan ma-
trix ...
   catdet=.monad :'det catmat y'           NB. ... and its
determinant
      det=.-/ .*
```

45. A partial solution to a partial problem

Principal Topics p. (*polynomial*), %. (*matrix inverse*), /. (*oblique*) ~ (*reflex*) -. (*less*) polynomial quotients, polynomial multiplication, polynomial factors, roots of equations

A seemingly innocuous post on the J Forum asked if anyone had a general routine for resolving partial fractions. Given that the heavy power-horses of `p.` and `%.` are already present in J, it seemed to require just small extensions of these to solve the problem, and it may be that this is indeed the case. Nevertheless I found this to be quite a tricky exercise, and as the title above suggests, the path to a general partial fraction algorithm as given here is not quite complete. However, some of the J features which turn up on the way are interesting in their own right, notably the use of explicit rank.

First, the basic problem is relatively simple, namely that of rewriting a quotient of polynomials such as f(x)/g(x) in which f(x) is of lower order than g(x), in a form such as

$$k_1/(a_1-x) + k_2/(a_2-x) + \ldots$$

where $a_1, a_2, ..$ are the roots of g(x). Initially, assume that these are distinct.

Start by defining a **polynomial** as a list of coefficients in ascending power order which is the meaning implicit in the right argument of `p.`, so that the algebraic function $f(x) = 2x^2+5x - 3$ is represented by the polynomial `_3 5 2`. The value returned by monadic `p.` is

```
   p. _3 5 2
┌─┬──────┐
│2│_3 0.5│
└─┴──────┘
```

that is the roots `_3 0.5` along with the highest order coefficient 2 which helps distinguish f(x) from say $g(x) = 6x^2+15x - 9$, which has the same roots.

To model partial fractions, an initial decision has to be made between using boxes or lists. For example, the partial fraction $(1+x)/(1+3x+2x^2)$ could be represented either by a list of boxes

```
   1 1;1 3 2
```

or by a list of polynomials

```
   1  1  0
   1  3  2
```

My general principle is to use lists wherever possible unless data-inherent heterogeneity makes lists too burdensome. Since the contents of boxes are sealed by definition, the only available operations without opening are the relatively simple ones of joining and shaping. With lists, operations specifically appropriate to the data types are fully available, subject to the constraints of structural rectangularity which may force the insertion of fill characters. These can sometimes be benign, as in the case of polynomials where adding a couple of zeros on the right of, say, the polynomial 1 0 2 merely adds two non-contributory terms $0x^3 + 0x^4$ to the function $1 + 2x^2$.

Using this representation of polynomials, a fraction such as f(x)/g(x) is a 2-list of polynomials, for example (4-x)/(1 + $2x^2$) is the 2-list

```
4 _1 0
1  0 2
```

Further, the model is readily extendible by representing a number of such polynomial fractions as a list of 2-lists of lists, for example {2/(4+x)} – {6/(1+5x)} is

```
    t0
 2   0
 4   1

_6   0
 1   5
```

Next, the dyadic verb `pmult` (polynomial multiply) multiplies two polynomials and is often cited as an illustration of the adverb *oblique* /.

```
   pmult=.+//.@(*/)        NB. (dyad) polynomial multiplication
  4 _1 pmult 1 0 2
4 _1 8 _2
```

(Assume in what follows that defined verbs are monadic unless there is a specific comment to the contrary.)

A useful first step in developing a partial fraction algorithm is to develop an inverse verb, that is one which combines basic (that is 2 by 2) partial fractions into a single composite partial fraction.

```
   cp=.(+/@(pmult"1|.)),:pmult&(1&{)    NB. combine basic
p.frac'ns
   cp/t0
_22  4 0
  4 21 5
```

The roots and multiplier of the denominator of a partial fraction are obtained by

```
   pfd=.p.@(1&{)     NB. partial fraction denominator
   of=.>@{pfd        NB. (dyad) pick from p. 0=multiplier,
1=roots
```

Thus if `u1` represents the partial fraction $(11x + 8)/(3 - 2x - 8x^2)$:

```
   u1
11  8  0
 3 _2 _8
   0 of u1           NB. multiplier
_8
   1 of u1           NB. roots of denominator
_0.75 0.5
```

Although having `p.` return a multiplier (which is already part of the input data) as well as roots adds complexity to its result, this approach is well justified by considering that the factorisation of an expression such as $3 – 5x - 2x^2$ is not unique – it could be the "obvious" factorisation of $(1 - 2x)(3 + x)$ or it could be $(0.5 - x)(6 + 2x)$, or $(2 - 4x)(1.5 + 0.5x)$ and so on. Thus in factorising a polynomial into linear factors (which is always possible because every nth. order polynomial has exactly n roots, allowing for possible complex values), the multiplier delivered by `p.` must be applied arbitrarily to one of the factors. The choice made here is to apply it to the first. A general verb for multiplying the head only of a list is most clearly expressed as an explicit function :

```
   mhead=.dyad :'(x*{.y),}.y'    NB. x*head of y,tail unchanged
```

following which roots are changed into factors by `rtof`, which reverses signs and catenates 1s, and adjusted by the multiplier with `facs`:

```
   rtof=.,.&1@-@(1&of)           NB. turn roots into factors ..
   facs=.(0&of)mhead rtof        NB. ..and adjust for multipli-
er
```

If there are just two roots, everything is in place to find the factors :

```
      facs u1                    NB. factors
  _6 _8
_0.5  1
```

.. and then the partial fraction coefficients :

```
   11 8 %.|:facs u1 NB. coefficients
_1.5 _4
```

so that the resolution is {-4/(-6 - 8x)} – {1.5/(-0.5 + x)} or equivalently {2/(3 + 4x)} + {3/(1 - 2x)}

More generally, the numerator coefficients are equated to the combined coefficients of the denominator polynomial factors following multiplication with one factor at a time omitted. This gives two nice examples of the use of explicit rank, one to exclude each item in turn in the list of factors, then `pmult` is used at rank 2 to multiply these factors together. For example, given the fraction

$$(23 + 55x + 8x^2)/(3 + 13x - 18x^2 - 40x^3)$$

to resolve, define:

```
   u2              NB. partial fraction
23 55   8   0
 3 13 _18 _40

   facs u2         NB. basic polynomial factors of u2
 _30 _40
_0.5   1
 0.2   1
```

Now use *less* which for `x-.y` includes all items of x except those which are cells of y :

```
   minors=.-."2 1~
   minors facs u2
_0.5   1
 0.2   1

 _30 _40
 0.2   1

 _30 _40
_0.5   1
```

and 'polynomial multiply' within each pair of factors

```
   mat=.pmult/"2@minors@facs NB. lin eqns for pf coeffs
   mat u2
_0.1 _0.3   1
  _6  _38 _40
  15  _10 _40
```

At this point the power of *matrix divide* comes into play and is consolidated in a verb :

```
   pfc=.}:@(0&{) %.|:@mat  NB. partial fraction coefficients
   pfc u2
_20 _1.5 0.8
```

which gives the partial fraction coefficients which correspond to `facs u2` in order.

Finally the verb `form` brings coefficients and factors together in the basic partial fraction representation :

```
   form=.(,:~,)"1 0   NB. (dyad) merge factors and coefficients

   pf=.facs form pfc  NB. partial fractions
   pf u2
 _20   0
 _30 _40

_1.5   0
_0.5   1

 0.8   0
 0.2   1
```

It is easy to confirm that `cp/pf u2` is identical to `u2` and similarly for `u1`. As further confirmation using the first example `pf cp/t0` is the same as `u0` within constant factors :

```
   pf cp/t0
  10 0
  20 5

_1.2 0
 0.2 1
```

The problem of distinct real denominator roots has thus been fully dealt with, which leaves two matters outstanding, namely complex roots, and repeated roots.

For complex roots, take as an example $1/(1+x^5)$ represented by

```
   u3=.2 6$1 0 0 0 0 0 1 0 0 0 0 1
   u3
1 0 0 0 0 0
1 0 0 0 0 1
```

pf operates as for real roots

```
    pf u3
_0.161803j_0.117557 0
_0.809017j_0.587785 1

 _0.161803j0.117557 0
 _0.809017j0.587785 1

                0.2 0
                  1 1

0.0618034j_0.190211 0
 0.309017j_0.951057 1

 0.0618034j0.190211 0
  0.309017j0.951057 1
```

If the implementation dependent assumption is made that imaginary complex pairs occur together, there is no problem in combining basic fractions into partial fractions with real coefficients as in :

```
   cp/2{.pf u3            NB. a quadratic partial fraction
0.4 _0.323607 0
  1  _1.61803 1

   cp/_2{.pf u3           NB. another quadratic partial frac-
tion
0.4 0.123607 0
  1 0.618034 1
```

It is a straightforward matter to write a verb which post-processes the results of `pf` by combining each complex fraction with its neighbour.

Next, multiple roots, say the resolution of $(6+8x+3x^2)/(1+x)^3$ where the appropriate partial fraction structure is $k_1/(1+x)^3 + k_2/(1+x)^2 + k_3/(1+x)$. This is a relatively easy application of the binomial coefficients and matrix divide :

```
   bc=.!/~@i.       NB. binomial coefficients
   bc 3
1 1 1
0 1 2
0 0 1
```

and the relevant coefficients are found by

```
   6 8 3 %.bc 3
1 2 3
```

that is, $(6+8x+3x^2)/(1+x)^3 = 1/(1+x)^3 + 2(1+x)^2 + 3(1+x)$.

More generally the binomial coefficients for the polynomial a b are found by

```
   bco=.dyad :0      NB. (dyad) coeffs. of (polynomial y.)^x.
r=.(,1),:y [ i=.2
while. i<x do.
  r=.r,({:r)pmult y [ i=.i+1  end.
```

so for a partial fraction $(4 + 18x + 9x^2)/(2 + 3x)^3$, the first step is the matrix

```
   3 bco 2 3
1  0 0
2  3 0
4 12 9
```

followed by matrix divide to obtain the coefficients :

```
   4 18 9%.|:3 bco 2 3
_4 2 1
```

that is $(4 + 18x + 9x^2)/(2 + 3x)^3 =$
$$\{-4/(2 + 3x)^3\} + \{2/(2 + 3x)^2\} + \{1/(2 + 3x)\}$$

Notice that `bco` depends only on `pmult` and not on any of the factorisation verbs. If the denominator is multiplied by a further factor, say resolve $(4 + 14x + 27x^2 + 18x^2)/(x+1)(2 + 3x)^3$ into partial fractions, each of the polynomials listed in `3 bco 2 3` must be multiplied by the new factor (hence `pmult"1`), and a third order set of binomial coefficients added :

```
   m1=.((3 bco 2 3)pmult"1(1 1)),{:4 bco 2 3
   m1
1  1  0  0
2  5  3  0
4 16 21  9
8 36 54 27

4 14 27 18 %.|:m1
4 _2 _1 1
```

that is the resolution is

$$\{4/(2 + 3x)^3\} - \{2/(2 + 3x)^2\} - \{1/(2 + 3x)\} + \{1/(1 + x)\} .$$

At the start I said that the journey was not complete, but at least a staging post has been reached from which it is just a matter of conscientious programming to achieve a general partial fraction algorithm.

Code Summary

```
pmult=.+//.@(*/)            NB. (dyad) polynomial multiplication
cp=.(+/@(pmult"1|.)),:pmult&(1&{) NB. combine partial fractions
pfd=.p.@(1&{)               NB. partial frtn denominator
rtof=.,.&1@-@(1&of)         NB. convert roots to factors ..
facs=.(0&of)mhead rtof      NB. .. and adjust for multiplier
   of=.>@{pfd               NB. pick from denom 0=mult, 1=roots
   mhead=.dyad :'(x*{.y),}.y'     NB. x*head of y,tail unchanged
minors=.-."2 1~             NB. uses dyadic -. (less)
mat=.pmult/"2@minors@facs   NB. lin. eqns for pfractn. coeffs
pfc=.}:@(0&{) %.|:@mat      NB. partial fraction coefficients
form=.(,:~,)"1 0            NB. merge factors and coefficients
pf=.facs form pfc           NB. construct partial fraction
bc=.!/~@i.                  NB. binomial coefficients

   bco=.dyad :0     NB. (dyad) coeffs. of (polynomial y.)^x.
r=.(,1),:y [ i=.2
while. i<x do.
r=.r,({:r)pmult y [ i=.i+1  end.
)
```

46. Tables and Geometry

Principal Topics = (*self classify*) ` (*tie*) `: (*evoke gerund*) ,. (*stitch*) " (*rank* conjunction) j. (*imaginary*) +. (*real / imaginary*)· tables, identity matrix, inner product, *apb* notation, upper/lower triangular matrices,

The identity table of any order can be formed in many ways, of which the simplest is as a fork `i.=/i.` This also spawns definitions of upper triangular tables with or without the diagonal

```
   <"2 (i.=/i.)`(i.<:/i.)`(i.<:/i.)/.3 3 3
┌─────┬─────┬─────┐
│1 0 0│1 1 1│1 1 1│
│0 1 0│0 1 1│0 1 1│
│0 0 1│0 0 1│0 0 1│
└─────┴─────┴─────┘
```

and lower triangular tables follow in an obvious way.

Another even shorter definition for the identity table is `=@i` . Superficially this looks like *equals* but in fact it is the monadic verb *self-classify* which is at the heart of the matter. When its argument contains no duplicates it is just the equals table of the argument with itself.

Plane Transformations

Start with `id` and `di`

```
   id=.i. =/ i.             NB. identity matrix
   id 3
1 0 0
0 1 0
0 0 1
   di=.<"1@(i. ,. i.)       NB. coeffs of diagonal
   di 3
┌───┬───┬───┐
│0 0│1 1│2 2│
└───┴───┴───┘
```

so a scaling matrix for enlarging 2 units in the x direction and 3 in the direction is

```
    2 3(di 2)}id 2
2 0
0 3
```

Define a pennant by suitable (x,y) coordinates :

```
   ]pen=. |:>0 0;5 5;4 5;4 4
0 5 4 4
```

```
0 5 5 4
```

... and here are the coordinates of this scaling

```
   mp=.+/ .*
   (2 3(di 2)}id 2) mp pen
0 10  8  8
0 15 15 12
```

To translate `pen` two units to the right and three units down

```
  2 _3+"0 1 pen
 2 7 6 6
_3 2 2 1
```

The isomorphic plane transformations whose combinations cover all possible such transformations are scaling, translating and rotating.

Rotating in 2D

```
   cs=.+.@^@j. NB. obtains cos y and sin y
   pi=.1p1
   cs pi%3
0.5 0.866025
```

Use `cs` to obtain the rotation matrix $\begin{pmatrix} \cos y & -\sin y \\ \sin y & \cos y \end{pmatrix}$

```
   rot2=.(cs&-)`(|.&cs)`:0 NB. evoke gerund

   rot2 pi%3
     0.5 _0.866025
0.866025       0.5

   (rot2 pi%3)+/ .*1 1
_0.366025 1.36603
```

Another way of performing this rotation is to use complex numbers based on the identity e^{iy}=cosy+isiny

```
   r2=.* ^@j.
   1j1 r2 pi%3
_0.366025j1.36603
```

Here is its rotation of the pennant through an anti-clockwise angle of π/2

```
  (rot2 pi%2)+/ .*pen
0 _5 _5 _4
0  5  4  4
```

... or using the alternative method

```
   q=.0 5j5 4j5 4 4
   q (r2 every) pi%2
0 _5j5 _5j4 2.449e_16j4 2.449e_16j4
```

Scaling and translation in 3D extends naturally into three dimensions:

```
   2 4 3(di 3)}id 3
2 0 0
0 4 0
0 0 3
   (2 4 3(di 3)}id 3)mp 2 1 3
4 4 9
```

To translate four points in 3D by 2 x-units -3 y-units and 4 z-units :

```
   2 _3 4 +"(0 1)pen,1
 2 7 6 6
_3 2 2 1
 5 5 5 5
```

In 3D there are infinitely many possible axes of rotation of which the three main ones are about axes at right angles to the Oxy, Ozx and Oyz planes, for which purpose the result of `rot2 y` has to be placed by amendment into the relevant 2 by 2 block of `id 3` . For example the relevant coordinate pairs for the Oxy rotation are

```
   <"(1)2 2#: i.4
┌───┬───┬───┬───┐
│0 0│0 1│1 0│1 1│
└───┴───┴───┴───┘
```

and those for Ozx and Oyz are obtained by doubling and incrementing respectively. Rather than using a 3D solid object, the illustrations show the effect of rotating a ray joining the origin to the point with coordinates (2,1,3).

```
   rxy=.monad :'(,rot2 y)(<"(1)2 2#: i.4)}id 3'

   rxy pi%3
  0.5 _0.866 0
0.866    0.5 0
    0      0 1
   (rxy pi%2) mp 2 1 3
_1 2 3

   rzx=.monad :'(,rot2 y)(<"(1)2*2 2#: i.4)}id 3'
   (rzx pi%2) mp 2 1 3
_3 1 2
```

```
    ryz=.monad :'(,rot2 y)(<"(1)>:2 2#: i.4)}id 3'
   (ryz pi%2) mp 2 1 3
2 _3 1
```

Code Summary

```
id=.i. =/ i.         NB. identity matrix
di=.<"1@(i. ,. i.)   NB. coeffs of diagonal
mp=.+/ .*            NB. matrix product
cs=.+.@^@j.          NB. obtains cos y and sin y
pi=.1p1
rot2=.(cs&-)`(|.&cs)`:0    NB. 2D rotation matrix
r2=.* ^@j.                 NB. ditto as complex list
```

3D rotations about major axes

```
rxy=.monad :'(,rot2 y)(<"(1)2 2#: i.4)}id 3'

rzx=.monad :'(,rot2 y)(<"(1)2*2 2#: i.4)}id 3'
ryz=.monad :'(,rot2 y)(<"(1)>:2 2#: i.4)}id 3'
```

47. Musical J-ers

> Principal Topics : \: (*grade down*) ~ (*passive*) \ (*infix*) tonic, dominant, octave, intervals, clock arithmetic, ciccle of fifths, celestial harmonies, 12 note chromatic scale, diatonic scale, Pythagorean tuning, just intonation, mean tone temperament, equal temperament, cents, Pythagorean comma, Wolf fifth. syntonic comma, frequencies.

Every now and then when struggling with some abstruse aspects of a technical subject, I get the feeling that it would have been so helpful if the experts and specialists had told me about it in J. One such area is that of temperament in music, and what follows is my attempt to make that case for using J to enlighten things in this field.

To start at the beginning, Pythagoras, he of the hypotenuse, also had strong ideas about music. He realised that a vibrating string when suddenly stopped at its middle point produces a note melodically identical to the original, only, in modern terminology, an octave higher. Call the melodic value of both these notes the tonic, so that advancing an octave allows us to 'listen' to the fraction 1/2. If the string is now stopped its 2/3 point the result is another note called the dominant which, when sounded at the same time as the tonic, produces a pleasant sound combination.

Simple fractions sound nice

From this starting point two separate experiments proceed. In the first the string is stopped at other fractional points with small integer numerators and denominators. Since the octave represents a full melodic circle which is repeated at 1/4 then 1/8 and so on, there is little point in considering stops other than those which lie between 1/2 and 1. The next 'interesting' stop is thus at 3/4, followed by others at 3/5, 4/5 and 5/6. At this point all fractions with components of 6 or less have been exhausted, and in all cases pleasing sound combinations with the tonic are obtained. This experiment has incidentally provided a means of 'hearing' the following range of fractions : 1/2 2/3 3/4 4/5 5/6 which are defined by the hook

```
    (%>:)1 2 3 4 5  NB. octave, 5th, 4th, 3rd, minor 3rd
0.5 0.667 0.75 0.8 0.833
```

annotated above with **intervals** names which describe distances from the tonic. It is important to distinguish between notes and intervals, which is akin to observing the gaps in the fence rather than the fence

posts. The piano tuner tweaks strings and the organ tuner adjusts pipes to produce **notes**, but what the listener hears is primarily intervals.

Advancing to higher integers in the above sequence, any fraction involving a 7, that is 4/7, 5/7, 6/7 and 7/8 produces distinctly unpleasant sound combinations. As for 8s, there is just one ratio which has not been investigated, namely 5/8 which also sounds nice and corresponds to the interval called a minor third. All of the intervals 1/2 2/3 3/4 3/5 4/5 5/6 5/8 are "pure" in the sense that they can be related to pleasant sounds which arise from the <u>physical</u> properties embodied in a bowed string, or resonating pipe.

Calculating a circle of fifths

The second experiment involves intervals rather than notes, and consists of finding where the stop should be for the dominant of the dominant. The 'obvious' answer is 2/3 of 2/3 = 4/9, but this is outside the range 1/2 to 1. However, the first experiment showed that doubling the fraction lowers the note by an octave but makes no difference to its melodic quality of the note, so make the second stop position at 8/9. Then repeat the experiment to find the dominant of the dominant of the dominant at 2/3 of 8/9 = 16/27 which does not need doubling since it is already in the range 1/2. To continue this process use J to develop a compound verb "multiply-by-2/3-and-double-if-outside-1/2-to-1". This situation is reminiscent of clock arithmetic as practised in the early stages of primary school. For example in arithmetic modulo 5 adding 4 and 2 makes 1, multiplying 4 and 2 makes 3, notions which are captured in J by

```
   cadd=.5&|@+       NB. clock add
   4 cadd 2
1
   cmult=.5&|@*      NB. clock multiply
   4 cmult 2
3
```

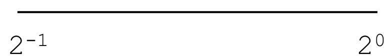
The whole infinite gamut of integers is thereby compressed into the set {0,1,2,3,4}. Another example of compressing an infinite into a finite one is the expression of numbers in scientific notation. Using logarithms, the fine detail of a real number is compressed into the range 1 (inclusive) to 10 (non-inclusive), while the exponent defines the wider territory within which the number lies. Again J can explain how to do this. If a number is expressed as v e x then

```
   x=.<.@(10&^.)     NB. exponent
```

```
   x 2675
3
   v=.%10&^@x       NB. value
   v 2675
2.675
```

In pictorial terms `v` compresses numbers into the space

Now return to the musical experiment with its 'special' multiplication in which e.g. $(2/3)^2 = 8/9$, so that the result always remains in the range 1/2 to 1. This compression region can be drawn as



– call the process "musical arithmetic" – which helps write the analogous verbs

```
   x2=.<.@>:@(2&^.)
   x2 1r8 2r3 4r9          NB. 2-exponent
_2 0 _1
   v2=.%2&^@x2             NB. value
   v2 1r8 2r3 4r9
0.5 0.666667 0.888889
```

In musical arithmetic the "nice" sounds as defined above, (or as Pythagoras would have more grandiosely called them "celestial harmonies") are inverses according to the following plan :

2/3 : fifth	1/(2/3) = 3/4 fourth
4/5 : major third	1/(4/5) = 5/8 minor sixth
3/5 : major sixth	1/(3/5) = 5/6 minor third

So applying the *power conjunction* to the "musical multiplication" verb to extend the Pythagorean progression of fifths :

```
   v2 2r3^>:i.12    NB. successive fifths
0.667 0.889 0.593 0.79 0.527 0.702
             0.936 0.624 0.832 0.555 0.74 0.987
```

After twelve applications, a value 0.987, is obtained which is not too far from 1 representing the tonic. Given that the numerators are powers of 2 and the denominators are powers of 3, there can never be any question of solving $(2/3)^k = 1/2$ exactly, and so it is reasonably satis-

fying to get as close as 0.987/2 in twelve steps. Musically this means that jumping by intervals of a fifth twelve times take us through a cycle of sounds which then repeats itself after an adjustment to make the octave pure. The natural place to make this adjustment is at the final step, but it could be made at any intermediate step or indeed spread across several steps. The above series is related to the familiar notes on a piano keyboard as follows

```
   v2 2r3^>:i.12    NB. progression of fifths
0.667 0.889 0.593 0.79 0.527
G     D     A     E    B
             0.702 0.936 0.624 0.832 0.555 0.74 0.987
               F#    C#    G#    D#    A#    F    C
```

The experiment is not over yet because the next question is what would happen if the above exercise was repeated for 3/4 rather than 2/3. The primary interval in this case is called a fourth, and the result is

```
   v2 3r4^>:i.12    NB. progression of fourths
0.75 0.563 0.844 0.633 0.949 0.712 0.534 0.801 0.601 0.901
0.676 0.507
 F    Bb    Eb    Ab    Db    F#    B     E     A     D     G
C
```

that is, the same notes only in reverse order, and finishing on the low octave, value 1/2, rather than the high octave at 1.

The twelve notes thus identified comprise the 12-note chromatic scale which has underpinned most Western music since around 1600. However, the first of the above lists shows that if the stop positions on the string are obtained by successive fifths (a scheme which is called **Pythagorean tuning**, although Pythagoras himself would only have recognised the first few steps), then by the time F comes round its value will be slightly different from the 'pure' value of 3/4. Likewise the second list shows that tuning by successive fourths G will also be a shade impure. Similar considerations apply to the other notes, which in turn means that the intervals will differ from the intervals identified in the first experiment. More importantly, the adjustment noted above which is needed to make the octave pure at the twelfth and final step is called a **comma**, or more specifically a **Pythagorean comma**, and sometimes again the **comma of Didymus**.

The Problem of D

So far six of the eight notes of the diatonic scale (that is the white notes on the piano in the scale of C) have been given places in the

scheme of things, the two remaining being D and B. Since these are symmetrically placed at either end of the octave, a discussion of one is automatically a discussion of the other, so focus on D. D is not consonant with C, so there is no physical "right" fraction for it, rather there are two candidates. The first comes from considering the fact that D is one whole tone removed from C, and there is already a whole tone represented, namely F – G, whose ratio is (2/3) / (3/4) = 8/9. The second candidate arises from the fact that in order to make D – A a pure fifth D must be set at (3/5) / (2/3) = 9/10. (A harpsichord with two such D keys was in fact built in Holland in 1639, but did not prove particularly popular for obvious reasons!) The diagram below shows D set to 8/9, and 9/16 as the symmetrical consistent choice for B which makes high C equal to 8/9ths. of B.

1/2	9/16	3/5	5/8	2/3	3/4	4/5	5/6	(8/9)	1
0.5	0.567	0.6	0.675	0.667	0.75	0.8	0.833	(0.889)	1
C	B	A	G#	G	F	E	Eb	(D)	C

The Note in the middle

A full octave in the chromatic scale (that is including two tonics) has 13 notes, and thus 12 intervals, and a middle note, namely F#. Where does it appear in the above table? The answer is that it doesn't because 'half-way' on a multiplicative scale means the $1/\sqrt{2}$ position, so playing the interval C – F# on the piano is a way of hearing the square root of 2! Moreover a glance at the progression series above shows that under musical multiplication both $(2/3)^6$ and $(3/4)^6$ are approximations to $1/\sqrt{2}$, one being about 0.005 above and the other the same amount below. On either side of the middle, interval of a fifth, C – G, consists of seven semi-tones, whereas a fourth, C – F, consists of five semitones in which respect fifths and fourths are mirror images of each other, explaining incidentally why each of the two progression series is the reverse of the other.

Adjusting the scale

The following is another copy of the 1/2 to 1 region in which the fractions are labelled with interval names rather than notes (dim stands for "diminished")

1/2	9/16	3/5	5/8	2/3	3/4	4/5	5/6	8/9	1

C	B	A	G#	G	F	E	Eb	D	C
oct	7th	6th	dim 6th	5th	4th	3rd	dim 3rd	2nd	

Under this scheme the whole tones D – E and G – A have values (4/5) / (8/9) and (3/5) / (2/3), both of which are equal to 9/10, which was the alternative candidate for D. This means that there are two types of whole tone in this scale, so that, for example, the first three notes of "Three Blind Mice" become a melodic progression of unequal steps. Also the ratios for the main consonant intervals, obtained by dividing the value of the second note by that of the first, are

perfect fifths				major sixths		major third
F - C	G - D	D - A	A - E	F - D	G - E	F - A
2/3	2/3	27/40	2/3	16/27	3/5	4/5

The values of 2/3, 4/5 and 3/5 are consistent with those for the tonic C, but clearly compromises must be made on account of the introduction of D which would need to be 9/10 to make D - A a pure fifth. Similarly in the key of G# the major third is G# – C, ratio 1/(5/8) = 4/5 and the fifth is G# – Eb = (5/6) / (5/8) = 2/3 both of which are pure. However, in the key of E the major third E – G# has the ratio (5/8)/(4/5) = 25/32 or 0.781 which is just a touch impure. And so one could go on. Once a set of strings is tuned for pure concordances in key C, compromises must be made, not just for melodies and harmonies in the key of C, but also for melodies played in other keys. How best to make such compromises has engaged the minds of musicians since medieval times, and is the subject of a fascinating little book called "Temperament - the idea that solved Music's Greatest Riddle" by Stuart Isacoff. (The word "temperament" was first used in this context round about 1500, and means, according to Chambers "a system of compromise in tuning".) The history of the debates on temperament is complex; but broadly, the D problem gave rise to two solution streams, one called **just intonation** which tolerated differences in whole tone values as a price worth paying for purity of major thirds, the other called **mean tone temperament** which is based on making whole tones uniform. "Just" should be thought of as being derived from "adjustment", of which there has already been hints in the preceding section. The notion of making adjustments to organ pipes or strings on keyboards may well date as early as the late 14th. century, and it is important to bear in mind that Renaissance instruments were much less full-blooded than their modern counterparts, so that both players and listeners would have been more sensitive to variations in tuning than audiences are today.

Equal Temperament

In the mid 16th century the concept of the equal-tempered scale emerged in which each of the twelve semi-tone intervals are equal on a multiplicative scale. This has the merit that music can be freely transposed into other keys, but at a cost of losing the 'purity' of the Pythagorean ratios in <u>any</u> key. On the other hand tuning for Pythagorean perfection in, say, the key of C, means that transposing outwards to remote keys on the circle of fifths results in increasingly unpleasant harmonies. Equal temperament tuning first found favour among lute players for whom other forms of tuning necessitated the undesirable feature of having frets at unequal distances for different strings. Again, J can clarify and quantify what musicians and musical historians mean when they talk about this topic. In an equal-tempered system (and also a well-tempered system which is a subtly different but more sophisticated variation of it), the common ratio of the series of semi-tone values must be the twelfth root of 2, and so the stop ratios going up the scale are given by

```
   ]r=.2^-%12 NB. r is 1/12th root of 2
0.944
   r^i.13
1 0.944 0.891 0.841 0.794
      0.749 0.707 0.667 0.63 0.595 0.561 0.53 0.5
```

The following series is the corresponding ordered version of the ratios for Pythagorean tuning :

```
   \:~1, v2 2r3)^i.12      NB. equal-tempered stop positions
1 0.936 0.889 0.832 0.79
      0.74 0.702 0.667 0.624 0.593 0.555 0.527 0.5
```

For the purposes of comparing tuning systems it is useful to convert from a multiplicative scale ranging from 1 to 1/2 going up the scale to a.n additive one from 0 to 1200 in which each semitone interval is represented by 100 in an equal-temperament system. Pictorially this conversion is

2^{-1}	$2^{-(1/2)}$	$2^{-(1/12)}$	2^{0}
		to	
1200	600	100	0

Musicians call the unit which divides an octave into 1200 parts a **cent**, and J readily provides the means of conversion

```
   cent=.1200&*@(2&^.)@%   NB. convert stop positions to cents
```

```
   cent %12 4 3 2 1.5%:2   NB. equal tempered C# Eb F F# G#
100 300 400 600 800

   cent r^i.13             NB. 12-tone equal-tempered scale
0 100 200 300 400 500 600 700 800 900 1000 1100 1200
```

It is informative to see how Pythagorean tuning measures up on a cent scale. First observe the cent values of pure fifths, thirds and sixths which are respectively :

```
   rnd=.<.@(0.5&+)                 NB. round to nearest integer
   rnd cent v2 2r3 4r5 3r5         NB. 5ths, maj 3rds. 6ths.
702 386 884
```

The values of the three complementary intervals in the diatonic scale (fourths, minor sixths and minor thirds) are just 1200 minus these values, as confirmed by

```
   rnd cent v2 3r4 5r8 5r6         NB. 4ths, min 3rds. 6ths.
498 814 316
```

Next establish the cent values of all twelve points on the Pythagorean chromatic scale

```
   rnd cent \:~v2 2r3^i.13         NB. Pyth tuning to cents
23 114 204 318 408 522 612 702 816 906 1020 1110 1200
 C  C#   D   Eb  E    F   F#   G   G#   A   Bb    B    C
```

The initial value of 23 represents the comma, that is the extent to which tuning by repeated fifths 'misses"' the octave whose purity in <u>any</u> tuning system is sacrosanct. Since an octave is also the sum of three major thirds and of four minor thirds, there are also discrepancies in these of 1200 – (3x386.6) = 41 below and (4x316) - 1200 = 63 above respectively, which are also commas of a sort, although the unqualified use of 'comma' means the measure of around 23 cents. This comma is incidentally also the interval by which the fourth C - F is impure within a well-tempered system.

The sequence of fifths in the row above, that is C, G, D, A, E etc., shows a progressive overshoot of 2 cents at each step cumulating into the comma which, if corrected at the final step, requires this step to be curtailed to 702-23 = 679 cents. Two notes at this interval produce a discordant sound known since mediaeval times as the **wolf fifth**, presumably because of its supposed likeness to the braying of a wolf.

Another way of looking at the last row of figures above is to calculate their differences :

```
   2-~/\0 114 204 318 408 522 612 702 816 906 1020 1110 1200
 114 90 114 90 114 90 90 114 90 114 90 90
C      D      E   F     G      A      B  C
```

This shows that every semitone is worth one of two values, namely 90 cents or 114 cents. If now the two semi-tones in the diatonic scale (that is the C scale without any black notes) are equalised at 90 by interchanging the 114 and 90 between E and F#, then the five whole tones in the diatonic scale are also equalised at 204 cents. (5x204) + (2x90) = 1200, which confirms the purity of the octave. The difference of 12 cents between some semitones and others is perceptible only to the most highly trained ears.

As an aside, it might be supposed that since fourths are the mirror images of fifths that tuning by successive fourths would be broadly similar. Try it on the computer, but not on your piano !

```
   rnd cent \:~v2 3r4^i.13
90 180 294 384 498 588 678 792 882 996 1086 1177 1200
```

Just Intonation

Just intonation systems are based on the notion that the major third is somehow a more "beautiful" consonance than the "fifth" (think of songs in which soprano and alto voices proceed in a blend in parallel thirds). Some form of this may have been in the mind of Ptolemy in the second century A.D., hence the occasional use of the term **Ptolemaic tuning** as a synonym. The starting point is that the Pythagorean system has a major disadvantage in that the major thirds (C - E, F - A and G - B) all have values of 408 cents whereas purity requires that they should have a value of

```
   cent v2 4r5      NB. size of equal-tempered major third
386.3
```

One way around this is to accept D as 204 cents as above, corresponding to a harmonic value of 8/9, but make the next tone D - E equal to 386 - 204 = 182 cents, a reduction of a comma. Reducing the interval G - A by the same amount simultaneously adjusts both the F – A and G – B thirds to the pure value of 386. This leaves the two diatonic semitones to take up the slack of 44 so each becomes 90 + 22 = 112. Now consider the chromatic notes. D – F# is currently 182 + 112 + 114 = 408, in excess by a comma. Switching the semi-tone values between F and G, and changing them slightly from 114/90 to 112/92 makes D

– F# pure, as are also F# - A and G# - B. This little bit of ingenuity leads to the scheme

```
   ]just=.92 112;90 92;112;92 112;92 90;112 92;112
┌──────┬─────┬───┬──────┬─────┬──────┬───┐
│92 112│90 92│112│92 112│92 90│112 92│112│
└──────┴─────┴───┴──────┴─────┴──────┴───┘
       D     E   F      G     A      B   C
```

It is now possible to use J to observe the effects of this particular just tuning on all the principal intervals based on different starting notes :

```
   2+/\13$;just      NB. tones
204 202 182 204 204 204 204 182 202 204 204 204
C   C#  D   Eb  E   F   F#  G   G#  A   Bb   B

   7+/\18$;just      NB. fifths
702 702 680 702 702 702 702 702 700 702 702 702

   4+/\15$;just      NB. major thirds
386 406 386 408 408 386 406 386 406 408 408 406

   9+/\20$;just      NB. sixths
884 904 884 906 906 906 904 884 904 906 906 906
```

Other theorists had different ways of getting around the problem of D, and because of the ad hoc nature of such systems just systems are also referred to as irregular temperaments. It is doubtful whether they were much applied in practice to harmonised music.

Two other systems have a place in the history of temperament, both of which In his time only the diatonic scale in which there are just three thirds (C – E , F – A and G – B). The F – A ratio in the Pythagorean scheme is (3/5) / (3/4) = 4/5 which is pure, and purity for G – B would suggest a value of (2/3)x(4/5) = 8/15 (0.533) for B which also establishes the purity of the fifth E – B. It has already been observed that there is flexibility when D is introduced. The major third involves the $1/\sqrt{2}$ note F#, and to make the minor third D - F pure means choosing a value of 9/10 rather than 8/9 should be chosen for D. Preserving the G - D fifth then requires adjusting G to 27/40 rather than 2/3 and to preserve the purity of the major third G - B, B should be set to 27/40 = 0.54 compared with the value reached at the fifth step in the fifths progression which is $(2/3)^5 = 128/243 = 0.527$ Thus in this system the purity of the thirds is therefore preserved at the expense of impurity in the fifths.

Mean-tone Temperaments

Mean-tone systems were specifically designed for keyboard instruments. In Pythagorean terms the note E required for the interval of a third is encountered after four steps (C – G – D – A – E), which, assuming that the progression is consistently upwards in pitch, represents an interval of four fifths, or equivalently two octaves and a major third. If the latter interval is to be tuned perfectly the component fifths must be tuned as the fourth power of 5 in the same way as semitones were obtained as powers of 12 in the well-tempered system. As already noted the pure third measures 386.3 cents, so that the two tones which comprise it have a mean value of 193.15 cents (cf. 204 in the just system). This mean is the size of all the whole tones in this system, hence the name "mean-tone temperament". Carrying out the sort of accountancy in the previous paragraph means that the values of the semi-tones intervals must be half of 1200 - (5x193.15) = 117 cents. This in turn means that those semi-tones which are not part of the diatonic scale must have the value 193 - 177 = 76 (cf. 92 under the just tuning described above) leading to a cent scale

```
76 117 117 76 117 76 117 76 117 117 76 117
      D      E   F      G      A      B   C
```

(An astute observer will spot that the above list of numbers totals 1199 highlighting a small rounding effect).

Mean-tone systems can be thought of as ‘splitting the comma’, and the scheme above is not the only way of doing so.

Comparison of systems described

The following table summarises in cents the values of the notes of the chromatic scale of C in the four systems of tuning considered in detail :

	C	C#	D	Eb	E	F	F#	G	G#	A	Bb	B	C
eq-temp:	0	100	200	300	400	500	600	700	800	900	1000	1100	1200
Pyth :	0	114	204	318	408	522	612	702	816	906	1020	1110	1200
just :	0	92	204	294	386	498	590	702	794	884	996	1088	1200
mean :	0	76	193	310	386	503	579	696	772	889	1006	1082	1200

Frequencies

In terms of frequencies, life is even simpler since for Pythagorean tuning the relative frequencies of notes in the scale are now compressed into

```
      ____________________________
      2⁰                          2¹
```

```
fx=.<.@(2&^.)     NB. exponent
fv=.%2&^@x        NB. value
```

Using the verb `fv` of course requires some correction due to the comma effect and the practical requirement that successive octaves should have values 1 and 2

```
    /:~fv 1.5^i.13
1 1.014 1.068 1.125 1.201 1.266
C   C'    C#    D      Eb    E
octave      dim 2nd     2nd     dim 3rd   3rd

       1.352 1.424 1.5 1.602 1.688 1.802 1.898
         F      F#     G    G#     A      Bb     B
        4th      dim 5th    5th  dim 6th    6th      dim 7th    7th
```

Under equal temperament the corresponding frequencies are

```
  1  1.059 1.122 1.189 1.26
octave      dim 2nd     2nd     dim 3rd   3rd

       1.335 1.414 1.498 1.587 1.682 1.782 1.888
       4th      dim 5th    5th  dim 6th    6th      dim 7th    7th
```

The above frequencies are relative; in terms of absolute frequencies, concert pitch is generally taken to be 440hz for the A above middle C, so using values from the above table as divisors the frequency of middle C is somewhere in the range 260-262hz depending on which tuning system is used, and similarly for other notes.

There is nothing in the above which cannot be found in, say, Encyclopedia Britannica, or Grove's Dictionary of Music and Musicians. However the accounts there are not particularly easy to understand, and exposition in J would have helped me greatly. Incidentally I consider that a few figures in Grove's tables under "Mean-tine Temperament" are in error – had the description been in J it would be immediately clear who was right! Perhaps like Ken I am just temperamentally inclined towards J!

Code Summary

```
x=.<.@(10&^.)            NB. exponent
v=.%10&^@x               NB. value
x2=.<.@>:@(2&^.)         NB. 2-exponent
v2=.%2&^@x2              NB. value
rnd=.<.@(0.5&+)          NB. round
cent=.1200&*@(2&^.)@%    NB. convert stop positions to cents
rnd=.<.@(0.5&+)          NB. round to nearest integer
fx=.<.@(2&^.)            NB. exponent
fv=.%2&^@x               NB. value
```

48. Heavens above!

Principal Topics : o. (*circle functions*) `\.` (*outfix*) spherical trigonometry, similitude, rotations, enlargements, determinant, minors, cofactors, cross product, identity matrix, direction cosines, altitude, azimuth, declination, right ascension, celestial sphere, hour angle, celestial meridian, transit

A general objective of J-ottings has been to draw attention to the considerable number of mathematical or mathematical type routines which are built into J primitives thereby leading to significant reductions of programming effort. One such feature is the versatility of `j` which although primarily a complex number constructor is adaptable to other circumstances in which objects are defined by pairs of numbers, for example betting odds (see E #14).

A further example concerns transformations of a 2-dimensional plane by means of matrices of the form $M = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}$ where a and are b real numbers. A transformation such as $M\begin{pmatrix} x \\ y \end{pmatrix}$ is called a **similitude**, that is a transformation which results in the combination of an **anti-clockwise rotation about the origin** and an **enlargement** of the objects described by the coordinates. (For a clockwise rotation exchange b and $-b$). Given `det=.-/ .*` standing for determinant, `det` M is $a^2 + b^2$ and its square root is the enlargement E. The rotation component is represented by M divided by E, resulting in a matrix of the form $\begin{pmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{pmatrix}$ where t is the anti-clockwise angle of rotation.

Now although a similtude could be applied to a triangle whose points are, say, (0,0), (2,1) and (0,1) by

```
   ]M=.2 2$2 _3 3 2          NB. similitude matrix
2 _3
3  2
   ]tri=.2 3$0 2 0 0 1 1
0 2 0
0 1 1
   M +/ .*tri
0 1 _3
0 8  2
```

to give the transformed triangle (0,0), (1,8), (-3,2), clearly *M* is defined by the number pair (*a*,*b*) and so can be represented compactly as a `j` pair, in which case enlargement E and rotation t are given by :

```
   10 o. 2j3
3.60555               NB. enlargement=sqrt of 2^2 +
3^2
   (%10&o.)2j3
0.5547j0.83205        NB. (cos x)j(sin x) where tan
x=3%2
```

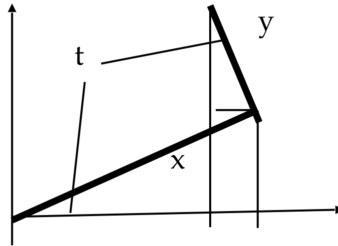
Instead of using a matrix inner product to transform points, simple multiplication is all that is required, so that the previous triangle transformation is given by

```
   2j3*every 0j0 2j1 0j1   NB. triangle transforma-
tion
0 1j8 _3j2
```

Also since multiplication is commutative, a product such as `2j3*2j1` has two geometric interpretations, viz. the similtude `2j3` transforms the point (2,1) to the point (1,8) and the similtude `2j1` transforms the point (2,3) to (1,8).

For rotation without enlargement the transformed coordinates of the point (x, y) in the new frame of reference are
$\begin{pmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos t - y\sin t \\ x\sin t + y\cos t \end{pmatrix}$ which can be confirmed by elementary trigonometry:

new (x,y)



The components of displacement from {x, y} are thus $\begin{pmatrix} x\cos t - y\sin t - x \\ x\sin t + y\cos t - y \end{pmatrix}$

or $\begin{pmatrix} x(\cos t - 1) - y\sin t \\ x\sin t + y(\cos t - 1) \end{pmatrix}$

which can be written $-(1-\cos t)\begin{pmatrix} x \\ y \end{pmatrix} - \sin t\begin{pmatrix} y \\ -x \end{pmatrix}$. The object of this rearrangement will become apparent shortly.

Rotations in three dimensions

Here the geometry is more complicated and `j` no longer helps. Take those rotations in which any line through the origin may be chosen as axis. Define such a rotation by any point on it other than the origin, and normalise this so that the defining point lies on the unit sphere. The results of this normalisation are the **direction cosines** of the axis of rotation, that is the cosines of the angles which this axis makes separately with each of the coordinate axes :

```
   dircos=.% %:@(+/@:*:)         NB. direction
cosines
   dircos 3 4 5
0.424264 0.565685 0.707107
```

A necessary preliminary is to obtain the **cross-product** of the rotation vector and a point to be rotated. To my knowledge there is no primitive which delivers cross-products directly, however it is a reasonably straightforward to write a verb **xp**. First stitch `3 4 5` (defining the axis) to `1 2 3`, a point to rotated, and use the 'all but one' technique described in J-ottings 52 to obtain the submatrices obtained by progressively eliminating one row at a time. The determinant of each of these 2x2 matrices is required with a suitable adjustment for alternating signs leading to :

```
   xp=.4 : '1 _1 1*det every<"(2) 1+\.(dircos
x),.y'
```

One other requirement is an identity matrix of appropriate length :

```
   id=.=@i.@#                      NB. identity matrix
```

Now suppose the anti-clockwise angle of rotation looking outwards from the origin is t. By a pleasing analogy with the two dimensional case the displacement components are

- (1 - cos *t*) times <a vector> - sin *t* times <a cross-product>

where 'a vector' is the result of the matrix multiplication

$$\begin{pmatrix} 1-\lambda_1^2 & -\lambda_1\lambda_2 & -\lambda_1\lambda_3 \\ -\lambda_1\lambda_2 & 1-\lambda_2^2 & -\lambda_2\lambda_3 \\ -\lambda_1\lambda_3 & -\lambda_2\lambda_3 & 1-\lambda_3^2 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

in which the λs are the direction cosines of the axis of rotation. The parameters defining a rotation are thus an axis (three coordinates) joined to the angle t, and it seems natural to take this 4-element vector as the left argument of a rotation verb. Also many people are more comfortable with degrees rather than radians, so define :

```
   dtor=.180%~o.              NB. degrees to radians
```

`rm` defines the rotation matrix above and `rmdata` multiplies it with the coordinates of the data point being rotated :

```
   rm =.(id - */~)@dircos          NB. rotation matrix
   rm 3 4 5
 0.82 _0.24 _0.3
_0.24  0.68 _0.4
 _0.3  _0.4  0.5
   rmdata=.rm@dircos@(}:@[) +/ .* ]
   (3 4 5,dtor 60) rmdata 1 2 3
_0.56 _0.08 0.4
```

(As an aside, taking the z axis as axis of rotation (0 0 1) so that $\lambda_3 = 1$ and $\lambda_1 = \lambda_2 = 0$ gives

```
   rm 0 0 1
1 0 0
0 1 0
0 0 0
```

which multiplies $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ to give $\begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$, while the cross-product of $\begin{pmatrix} 0 & x \\ 0 & y \\ 1 & z \end{pmatrix}$ is $\begin{pmatrix} -y \\ x \\ 0 \end{pmatrix}$ so that this reduces to the formula given earlier for the two-dimensional case.)

Next define `m1` and `m2`, bearing in mind that t has to be extracted as the 4th element of the rotation vector :

```
   m1=.-.@(2&o.@({:@[))     NB. (1-cos t)
   m2=.1&o.@({:@[)    NB. sin t
```

Finally reflect the formula for the dispacement components in

```
   rotate=.] - (m1 * rmdata) + m2 * }:@[ xp ]
   (1 0 0,dtor 90)rotate 1 2 3
1 3 _2
```

A couple of further checks helps confirm understanding :

(a) Rotate the point (1,2,3) through a <u>clockwise</u> angle of 90° about the x-axis:

```
   (1 0 0,dtor _90)rotc 1 2 3
1 _3 2
```

(b) Undo a clockwise rotation of (1,2,3) with an anti-clockwise one :

```
   axis=.?3$10   NB. choose an axis at random
   (axis,dtor 60)rota (axis,dtor _60)rotc 1 2 3
1 2 3
```

Multiple data points are dealt with by, for example

```
   (<3 4 5,dtor 60)rota every 1 2 3;2 3 1;3 1 2
1.03505  2.5299 2.55505
3.03722 1.56268 1.52753
1.82258 0.31773 3.25227
```

Plotting star movements

Every minute of every day we all perform rotations on a merry-go-round called Earth which itself rotates continuously within an even larger solar system which itself gyrates around another even larger sytem and so on. This a special case of 3D rotation in which all data points in the heavens are identified by two rather than than three parameters. Astronomers measure star positions as observed from Earth in angular rather than Cartesian measure. Specifically the two angles used are **altitude A** which corresponds to celestial latitude, and **azimuth Z** which correponds to longitude in terrestrial measurement. The stars themselves lie on the surface of a sphere called the **celestial sphere** which is continuously rotating about the extended Earth axis and on which every star has a latitude and a longitude which are called respectively **declination d** and **right ascension ra**. Analogous to the Greenwich meredian on Earth the celestial sphere requires an arbitrary zero line or **celestial meridian** from which right ascension is measured. This is conventionally taken to be the first point in Aries, which is observable as the rightmost star in the constellation Cassiopea. Azimuth is often measured in sidereal hours from 0 to 24 rather than degrees; the significance of 'sidereal' is that a sideral year is one day longer than a solar year, that is the fixed stars appear to rotate at a slightly slower speed than the sun, the difference being about 4 minutes per day. Stars rise in the east and set in the west, and so to an Earth-bound observer looking outwards to the Pole Star, the celestial sphere appears to rotate in an anticlockwise direction.

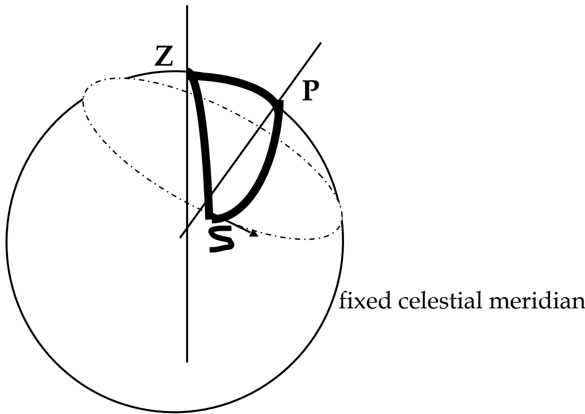
To convert star positions defined by A and Z into (x,y,z) coordinates assume the x-axis runs west to east, the y-axis north to south, and the z-axis upward. The plane x=0 is then a meridian on a fixed celestrial sphere from which Z is measured clockwise. (x,y,z) coordinates are then given by

$$x = \cos A \sin Z; \quad y = \cos A \cos Z; \quad z = \sin A$$

Inverting these formulae to convert from (x,y,z) coordinates to (A,Z) coordinates :

$$A = \sin^{-1} z; \quad Z = \cos^{-1}\frac{y}{\sqrt{1-z^2}} \quad \left(\text{or } \sin^{-1}\frac{x}{\sqrt{1-z^2}}\right)$$

The diagram below shows a star S in at a point in its daily circuit around the fixed star sphere (or, as it is sometimes referred to celestial sphere) :

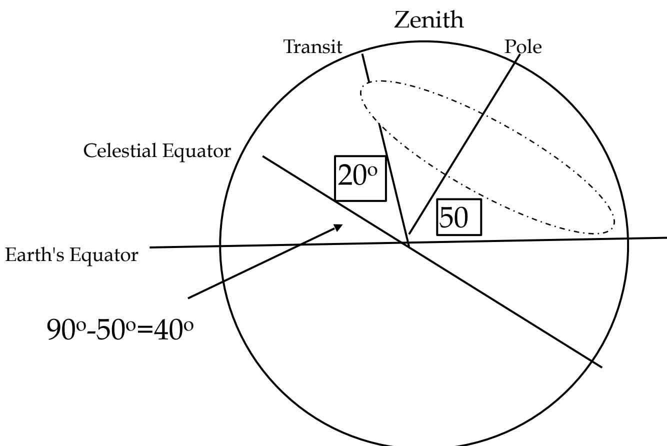


ZSP is a spherical triangle whose sides are all great circles of the sphere and in which P is the pole and Z is the Zenith. The side SP of triangle ZSP remains constant as S proceeds along the dotted line and is equal to 90° minus the declination d. Side ZP is also contant and is equal to 90° minus the latitude l. Quantities which change as S proceeds along its course are thus :

Side ZS = the co-latitude, that is 90° minus the altitude A
angle P = the celestial azimuth, known as the **hour angle**
angle Z = the terrestrial azimuth

Transits

A star is said to **transit** or **culminate** when it is at its highest point in the sky when seen by an observer on Earth. The diagram below shows a cross-section of the celestial sphere containing a star with declination 20° observed at point of transit from a latitude of 50 ° North which transits at (0, -cos 60°, sin 60°).



This diagram can be generalised to show that the altitude at transit is $(90^o - l) + d$ provided that $d<l$ as in the case of the star illustrated. This star transits south, that is to the left of the zenith and dips below the terrestrial horizon for at least part of its circuit. If $d>l$ a star is circumpolar and transits north. Here the altitude at transit is $(90^o + l) - d$, or combining the two cases, the altitude of every star at transit is $90^o - \text{abs}(l - d)$.

Calculating star positions

The position of a star depends on time as a parameter, which can be either local time – where was a star 6 hours ago? – or time by year – where was it 3 months ago? The star sphere appears to us to revolve from east to west completing a revolution in a sidereal day which is shorter by 1/365th of a day (that is approximately 4 minutes) than the solar day. Thus the position of a star 6 hours ago (¼ of a day) is the same as its position 3 months ago (¼ of a year).

For example, consider the star illustrated above with declination 20°, and ask what its position is 3 and 6 hours earlier and subsequently, that is when the hour angles is -90°

```
   (0,(cos 50),(sin 50),dtor -90)rota 0,(-cos 60),sin 60
0.939693 0.219846 0.262003
```

This result can be confirmed by spherical trigonometry applied to the triangle ZPS. The cos formula for a spherical triangle ABC states that if a, b and c are sides measured in angles, and A, B and C are the angles between the sides with A opposite a, etc. then

$$\cos a = \cos b \cos c - \sin a \sin b \cos A$$

so applying this twice to the diagram

$$\sin d = \sin l \sin A - \cos l \cos A \cos Z$$

and $$\sin A = \sin l \sin d - \cos l \cos d \cos H$$

Thus for the star with declination 20°, 6 hours earlier the hour angle is -90° so $\cos H = 0$ and therefore

$$\sin a = \sin 50^0 \sin 20^0$$

The values of *a* and its cosine are thus given by

```
    ]s=.*/sin 50 20              NB. sin a = z
0.262003
    ]cosa=.%:-.*:*/s             NB. cosa by Pythagoras
0.965067
```

and the azimuth value is :

```
   ]Z=.asin(cos 20)%cosa         NB. azimuth
76.8322
```

which enables the x and y coordinates to be found formulae given earlier :

```
   cosa*(sin,cos)Z               NB. x,y coords
0.939693 0.219846
```

The next step is to isolate the hour angle as a parameter (clockwise 90º is the same as anti-clockwise -90º):

```
    v=.monad :'(0,(cos 50),(sin 50),dtor y)rotc 0,(-cos 60),sin
60'
```

and plot values as this moves towards transit at 10º intervals :

```
   v every 90 80 70 60 50 40 30 20 10 0
 0.94    0.22 0.262
0.925  0.0948 0.367
0.883 _0.0264 0.469
0.814   _0.14 0.564
 0.72  _0.243  0.65
0.604  _0.332 0.725
 0.47  _0.404 0.785
0.321  _0.457  0.83
0.163  _0.489 0.857
    0    _0.5 0.866
```

More generally, it is useful to convert time to angular measurement with 24 hours being equivalent to a complete rotation, which suggests a few more utility verbs :

```
   ttor=.o.&(%&43200@(60&#.))     NB. time (hms) to radians
   ttor 12 0 0                    NB. check 12 hrs=pi rads
3.14159
   dtot=.60 60 60&#:@(*&240)      NB. deg to time (hms)
   dtot 180
12 0 0
   atod=.%&60@(60&#.)             NB. angle(deg,min)to deg
```

```
   atod 49 15
49.25
```

The cooordinates of the above star 15 and a half minutes after transit, are given by

```
   (0,(cos 50),(sin 50),ttor 0 15 30) rota 0,(-cos 60),sin 60
_0.0635044 _0.498354 0. 864645
```

that is a little bit to the west, a shade less south and a bit lower, all of as expected.

The next illustration concerns the sun which, unlike other stars whose declination is constant, has declination varying in the course of a year from -23.5° to +23.5 ° and back again. The sine formula in spherical trigonometry states that for a general triangle ABC :

$$\frac{\sin a}{\sin A} = \frac{\sin b}{\sin B} = \frac{\sin c}{\sin C}.$$

At sunrise and sunset the sun's altitude is zero, and so using the first of the two cosine formulae $\sin d = -\cos l \cos Z$ at these times, from which $\cos Z = -\dfrac{\sin d}{\cos l}$. Then using the sin formula, $\sin H = \dfrac{\sin Z}{\cos d}$.
Considering London (latitude of 51° 30') on the 21st December when the declination of the sun is -23° 30', and the altitude at noon, that is transit, is (90° - 51° 30') - 23 ° 30' = 15°00' ,

```
   ]Z=.acos(sin 23.5)% cos atod 51 30 NB. azimuth
50.17
   ]H=.dtot asin(sin Z)%cos 23.5        NB. time to noon
3 47 27.26
```

Now define `cs=.(cos,sin)@atod` and use the general rotation verb `rota` to rotate from transit for 3 hrs 47 minutes and 27.26 seconds :

```
   lat=.51 30
   dec=.-23 30
   tim=.3 47 27.26
   alt=.90-atod|lat-dec

   (0,(cs lat),ttor tim)rota 0,(-cos alt),sin alt
_0.7679 _0.6405 1.278e_7
```

and as expected the sun is south and west at altitude zero.

Code Summary

```
   xp=.cofs@:(norm@[ ,. ])         NB. cross product
   id=.=@i.@#                      NB. identity matrix
    norm=.% %:@(+/@:*:)            NB. normalise list
     cofs=.(* signs)@:det@submats NB. cofactors
       signs=.1 _1&($~ #)          NB. successive 1 and _1
       submats=.1&(+\.)            NB.successive 'all but one'
rows

    rm=.(id - >@(*every<))@norm   NB. rotation matrix…
       id=.=@i.@#
   rmd=.rm@norm@(}:@[) +/ .* ]    NB. …times data point

   m1=.-.@(2&o.@({:@[))            NB.multiplier (1-cos a)for rmd
   m2=.1&o.@({:@[)                 NB.multiplier sin a for xp
   rotc=.] - (m1 * rmd) - m2 * }:@[ xp ] NB.x.=axis,angle
   rota=.] - (m1 * rmd) + m2 * }:@[ xp ] NB.y.=data point

   dtor=.*&(o.%180)                NB. degrees to radians
   v=.monad :'(0,(cos 50),(sin 50),dtor y)rotc 0,(-cos 60),sin
60'
   ttor=.o.&(%&43200@(60&#.))     NB. time (hms) to radians
   rtod=.*&(180%(o.1))             NB. radians to degrees
   atod=.%&60@(60&#.)              NB. angle(deg,min)to deg
   sin=.1&o.@dtor                  NB. sine of an angle
   cos=.2&o.@dtor                  NB. cosine of an angle
   asin=.rtod@(_1&o.)              NB. arcsine of an angle
   acos=.rtod@(_2&o.)              NB. arccosine of an angle
  cs=.(cos,sin)@atod
```

49. Financial Maths and J – part 1, IRR and APR

Principal Topics : IRR (Internal rate of return), income stream. NPV (net present value), APR (annualized percentage rate)

A common way of assessing the profitability or otherwise of an investment is through the rate of return. There are several ways in which 'rate of return' can be calculated, one of the most common being Internal Rate of Return (IRR). Computing IRRs for different income streams is a way of comparing different investment strategies, e.g. when a company makes choices about which of a variety of potential products to develop. Moreover IRR carries on increasing with length of input stream provided inflows are positive, and so it may make sense to abandon further product development when the IRR begins to tail off.

Growth

The phenomenon of 5% compound interest growth can be described equivalently by any one of four numbers namely 5, 0.05, 1.05 and $(1.05)^{-1}$. These differences are expressible in J terms as :

```
p=.percentage rate                           (p>_100)
r=.fractional rate  so r=.0.01*p             (r>_1)
k=.multiplication factor so  k=.>:r          (k>0)
d=.discount factor so d=.%k or %>:0.01*p  (d>0)
```

All of these forms are equivalent ways of describing the same underlying phenomenon, so it is immaterial which value is quoted provided that the intend form is clear. Conversions between the various quantities are given by

```
ptod=.%@>:@(0.01&*)
dtop=.100&*@<:@%
ktop=.*&100@<:
ptok=.>:@*&0.01
```

Negative growth, that is decay, is indicated by `p<0, r<0, k<1` and `d>1.`

IRR

The significance of 'internal' is that the final worth of an income stream after discounting IRR is equal to zero. In other words ultimate worth is neither increased nor decreased so that IRR can be viewed as the continuous money decline which would need to occur to make the overall project do no more than break even in real terms.

If the actual rate of money decline exceeds the IRR, then it would have been better not to have undertaken the project. IRRs should in general be used in a relative sense to compare alternatives, rather than as absolute measures - the higher the IRR of a project alternative, the more flexibility there is for its success compared with other proposals. For valid comparisons, project proposals should have approximately the same initial outlays and the same income stream lengths. If these conditions are not met further considerations concerning investment of surplus and realised funds need to be taken into account.

Computing IRRs

Obtaining values for IRRs means solving polynomial equations for which income streams provide the coefficients.

```
   is=._100 20 50 70 80          NB. an income stream
   p.is
┌──┬───────────────────────────────────────────────────┐
│80│_1.21963 _0.20558j1.14621 _0.20558j_1.14621 0.755789│
└──┴───────────────────────────────────────────────────┘
```

A d value of 0.75579 is extracted and converted to p form by

```
   dtop 0.75579
32.312
```

to obtain an IRR of 32.3%. In words IRR is calculated as 'convert to p form the positive real root of the income stream regarded as polynomial coefficients'.

The final worth of the income stream can be obtained by using the d value as left input to the polynomial evaluator #. Notice that the coefficients in the argument for p. are in ascending power order, while those for are in descending order, so that the value of 0.75579 is confirmed by

```
   0.75579 #.|.is
0.00046177                    NB. effectively zero
```

It is always nice to have a simple case confirmed, so using the first example above

```
   p. _100 0 0 200
┌───┬──────────────────────────────────────────────┐
│200│0.793701 _0.39685j0.687365 _0.39685j_0.687365│
└───┴──────────────────────────────────────────────┘
```

from which 0.7937 leads to a compound interest growth of 26%

```
   dtop 0.7937
25.99
   1.26^3
2.00038
```

d values are obtained by opening the second box resulting from p .

```
   roots=.>@{:@p.          NB. extract list of roots
```

and obtaining the single non-negative, non-complex value, the latter property being tested for by comparison with *conjugates* as provided by + applied to complex numbers. From *tally* it is one small step to #~ as shorthand for 'select', which leads to two further building blocks

```
   real=.#~ (= +)          NB. select real values
   pos=.#~ >&0             NB. select positive values
```

The process of obtaining the internal rate of return then consists of four steps in sequence :

```
   irr=.dtop@pos@real@roots      NB. internal rate of return
for an is
   irr is
32.312
```

IRR calculations mirror those for fair loan repayment rates. The formula $(rk^n)/(k^n\text{-}1)$ gives the fraction of an amount loaned which must be repaid in each of n periods when interest p% is paid on the declining balance. The words 'declining balance' reflect the fact that IRR calculations make an implicit assumption that the inflows are reinvested at the calculated compound interest rate, which can sometimes lead to a rosy-eyed picture of absolute IRR values. Using the input stream `is` as an illustration, if k = the IRR, the progress of each flow the residual capital value is
$-100k^4 + 20k^3 + 50k^2 + 70k + 80$.

which is evaluated as

```
   (|.is) p.ptok irr is
_5.684e_14
```

More generally the residual value of the stream following a given discount rate is defined by

```
   resid=.dyad :'(|.y)p. ptok x'
   0 resid is              NB. equivalent to +/is
120
   10 resid is             NB. discounted ay 10%
```

```
97.71
```

Define

```
   Repay=.dyad : '(x*(1+x)^y)%_1+(1+x)^y' NB. Repayment Amounts
```

so that the consequent repayment amount for a loan of 10000 are given by

```
   0.005 Repay 12         NB. factor for monthly rate of 0.5%
0.086066
   10000*0.086066         NB. monthly repayments
860.66
```

A lender sees the transaction as an income stream of _10000 followed by 12 monthly payments of 860.66 whose IRR is therefore

```
   irr _10000, 12#860.66
0.49992                   NB. i.e. monthly rate of 0.5%
```

illustrating that IRR matches the fair rate of loan interest paid by equal installments on a declining balance. 'Thus `irr` and `Repay` are related by the equivalence :

```
          r -:  irr _1,n#(r Repay n)
```

IRR is reduced if payments are deferred, whereas if a loan is paid off early using the same proportionate repayments, the IRR increases. These statements are confirmed by

```
   irr _10000, (11#860.664),0,860.664   NB. last inflow delayed
0.49378
   irr _10000,6#2*860.664      NB. inflow rate doubled
0.92988
```

The related term ized (Annual Percentage Rate) takes into account that the equivalent annual rate is not 12*0.005, corresponding to 6%, but rather this rate compounded over the 12 periods :

```
   1.005^12
1.06168              NB. APR is 6.17%
```

To describe APR, use the conversions to and from `p` to `k` :

```
   apr=.ktop@(ptok@irr ^ <:@#)
   apr _10000, 12#860.66
6.168
```

Note on computational practicalities

`irr` may fail if the income stream is too long, or if the data generates a pathological situation for `p.`'s underlying root solver. In such circumstances, alternative numerical methods must be employed such as the Newton Raphson method (see E #23 “Numerical Problems analysed in J”) , which can be expressed in the adverb

```
   Newton=.1 :']-x % x D.1'(^:_)("0)
```

Suppose that a loan of 1000 is repaid in 24 instalments of 45.27. The APR is worked out by

```
   ms=.(24$45.27),_1000           NB. money stream reversed
   fn=.#.&ms                      NB. define a polynomial
   dtop fn Newton 0.9             NB. guess d=0.9 and apply N-R
 0.67447                          NB. monthly rate of return =
0.6745%
   1.0067447^12                   NB. convert to annual rate
1.084                             NB. APR = 8.4%
```

Net Present Value and Sequential IRRs

After two periods the value of the IRR on the series `is` is

```
   irr _100 20 50
_18.5857
```

The interpretation of this negative IRR is that the inflows must be inflated by (100-18.56)% = 0.814 to achieve the final overall zero outcome, that is

$$\frac{50}{(0.814)^2} + \frac{20}{(0.814)} = 75.4 + 24.6 = 100$$

Sequential IRRs can be obtained by a verb

```
   irrs=.;@:(irr every)@}.@(<\)   NB. successive irr's
   irrs is
_18.5857 15.6152 32.3121
```

When the IRR becomes positive in the third period the calculation becomes

```
   +/_100 20 50 70%1.1561^i.4
0.010231
```

The concept of Net Present Value is that future values are discounted progressively by a given percentage. This is expressed in the verb

```
   npv=.]*(ptod@[)^(i.@#@])                        NB. net present
value
   +/15.61 npv _100 20 50 70
0.010231
```

Assuming that the calculation is about money, and the periods are years, the combination of factors such as price inflation, deprciation, obsolsence, foregone returns on alternative investments, etc. are all bracketed under the heading 'inflation'. A decision to proceed with investment should be based on whether inflation over the entire period of investment is expcted to be less than 32.3%.
If future returns are promised as, say, interest payments at a rate of interest, say 10%, npv may be more appropriate than including this factor in IRR. The effect is naturally to reduce IRR :

```
   irr 10 npv is
20.28
   irr 25 npv is
5.85
```

Next consider an income stream such as

```
   is1=._100 20 45 60 _10 55 70 20 10 10
```

where some items are negative, reflecting, for example, costs incurred in developing a second version of a software product. Since polynomials of even order must have an even number of positive roots, using `p.` means that spurious values of IRR will necessarily arise, as for example in

```
   irr _100 20 45 60 _10
_85.04 6.67385
```

where the massive compound 85% decay is inadmissible. To deal with such circumstances it makes sense to insert a filter at the `d` level to exclude discount rates of greater than, say 3, or, at the other end of the scale, than 0.5, hence

```
   filter=.#~ (<&3)*.(>&0.5)
   irr=.dtop@filter@:pos@real@roots
   irrs is1
0 _22.18 10.22 6.674 19.96 27.96 29.33 29.82 30.19
```

The initial value of `_80` following the first inflow has been filtered to 0. Following the third inflow of `60` the IRR becomes positive at 10.2%, following which there is a dip to 6.67 due to the negative inflow of 10. Towards the end of the series subsequent increases in IRR tail off as a result of smaller inflows.

Now define another income stream `is2` with the same values as `is1` (total outflows = 110, total inflows =290), only in a different order with the higher values deferred :

```
   is2=._100 20 45 60 _10 10 10 20 70 55
   irrs is2
0 _22.18 10.22 6.674 9.919 12.37 15.81 22.4 25.22
```

which shows that the slower progress of the IRR.

Code Summary

```
   ptod=.%@>:@(0.01&*)
   dtop=.100&*@<:@%
   ktop=.*&100@<:
   ptok=.>:@*&0.01

   irr=.dtop@filter@pos@real@roots             NB. internal
rate of return
      dtop=.100&*@<:@%                      NB. discount factor to
%age
      filter=.#~ (<&3)*.(>&0.5)
      pos=.#~ >&0                           NB. select positive
values
      real=.#~ (= +)                               NB. select real
values
      roots=.>@{:@p.                               NB. extract list
of roots
   irrs=.;@:(irr every)@}.@(<\)             NB. successive irr's
   resid=.dyad :'(|.y)p. ptok x'            NB. residual value

   Repay=.dyad : '(x*(1+x)^y)%_1+(1+x)^y' NB. Repayment Amounts
   npv=.]*(ptod@[)^(i.@#@])                        NB. net present
value

   apr=.ktop@(ptok@irr ^ <:@#)

   Newton=.1 :']-x % x D.1'(^:_)("0)
```

50. Financial Maths and J – part 2; Growth Rates

Principal Topics : IRR (Internal rate of return), income stream. NPV (net present value), NFV (net future value), cost benefit ration, average growth rate, average compound growth, annuities

Average Growth Rates

'Growth' is a concept which is superficially easy but more subtle when it comes to ways of. measuring it. The income stream `is` has an overalf absolute growth of 20+50+70+80 = 220 over 4 periods which represents is an overall growth per period of the 4th root of 2.2 = 1.2178, that is 21.78%. This is identical to th IRR if all the inflows in `is` were in the last period

```
   4%:220%100
1.21788
   is
_100 20 50 70 80
   irr _100 0 0 0 220
21.7883
```

as opposed to an IRR of 32.1% (see E #49 "Financial Mathematics part 1"). IRR is not the only criterion for comparing different project proposals, and it is natural to ask questions about growth, suuch as what is the average growth. The successive period growths in `is` are 20 50 70 80 divided by 100 12 170 240 and so their average arithmetic average is

```
   gwth=.monad :'(}.y)%(}:+/\|y)'
   gwth is
0.2 0.416667 0.411765 0.333333
   mean=. +/ % #
   mean 1+gwth is
1.34044
```

quite close to the IRR of 32.3%. However this takes no account of discounting future costs Another possibility is to adjust all inflows and outflows to net present values by applying a discount rate `p`, and then obtain the ratio of ((inflows-outflows)/outflows), that is (net benefits)/costs. First the 5% and 10% discounted values of `is` are compared in

```
   npv=.] *(ptod@[)^(i.@#@])      NB. net present value
```

```
   5 npv is                        NB. is discntd at 5%
_100 19.0476 45.3515 60.4686 65.8162
   10 npv is                       NB. is discntd at 10%
_100 18.1818 41.3223 52.592 54.6411

   mean gwth  5 npv is
0.307983                           NB. is discntd at 5%
    mean gwth 32.1 npv is
0.190569                           NB. is discntd at IRR
```

19% growth may sound impressive but has to be set against the discounted value of 220 which is `+/}.32.31 npv` is = 100.00, in other words the initial investment has been returned in real terms but no more. This is what IRR predicts. A more realistic way of computing average growth rate is to compute the sum of the outflows :

```
   costs=.+/@:(0&>.@-)            NB. outflows (i.e. -ve values)
```

and divide all items in the stream by costs to obtain (net benefits)/ costs :

```
   bcr=.(+/%costs)@:npv           NB. net benefit cost ratio
```

Now compare the streams `is1` and `is2` each discounted by 10% :

```
   0 bcr is
1.2
   10 bcr is
0.667372
   32.1 bcr is
0.00429935
```

This gives yet another insight into IRR as that discounted value for which the original investment is returned but with no ultimate increase in wealth, although of couse in the business context the inflows are available to spend, reinvest, pay salaries or salaries, etc. Here are another two slightly longer income streams, both with total outflows of 110 and inflows of 290 and differing only in the order of the inputs

```
   is1=._100 20 45 60 _10 55 70 20 10 10
   is2=._100 20 45 60 _10 10 10 20 70 55

   10 bcr every is1;is2
0.80926 0.67133
```

Converting `bcr's` to an average compound growth rate (`agr`) means taking the 9th root using `%` :

```
   9%:1+0.80926 0.67131
1.0681 1.0587
```

```
   agr=.ktop@((<:@#@])%:>:@bcr)
   10 agr every is1;is2
6.80985 5.8729
```

that is the `agr` s are roughly 7% and 6% given a discount rate of 10%. For `is`

```
   0 agr is
21.7883
```

a value which how already been met as a form of the 4th root of 2.2. For a discount rate equal to the IRR

```
   32.1 agr is
0.107311
```

that is, the original investment is returned in real terms, and an average intermediate growth rate of around 11% for redistribution has been experienced.

If discounting is applied at around the **`irr`** value of 30.2 to `is1` then the net benefit would be more or less zero :

```
   30 agr is1       NB. net benefits are …
0.0527249           NB. … virtually zero
```

The difference between average growth rate calculations and IRRs is that the former set a discount rate and return a rate of income irrespective of the ultimate effect on capital, wheras the latter calculates a discount rate account on the assumption that capital does no more than break even. Thus AGRs should not be directly compared with IRRs, but should be seen as an alternative criterion. For income streams which do not differ significantly in either length or initial value, it usually results in the same relative ordering.

The relationship between `irr` and `agr` can be observed by moving all inflows, 220 in total in the case of `is`, to the final period :

```
   irr _100 0 0 0 220
21.7883
   0 agr every _100 0 0 0 220;is
21.788 21.788
```

that is for a nil discount rate the average compound growth rate is unaffected by when the inflows occur, whereas the earlier they occur the greater is the `irr` value.

In the case of personal investment, IRR can give a rough guide to the true value of investment returns. Consider a share purchase for £33.50 which attracts nine annual dividends of £2, £2, £2, £2.50, £3, £4, £4.45, £5.25 and £4.50, immediately after the last of which it is sold for £65.80.

```
   is3=._33.5 2 2 2 2.5 3 4 4.45 5.25 4.55 65.88
   irr is3
14.5668
```

This gives the investor a figure with which to compare the inflation rate in the same period. For lower rates the sum of the discount rates and average growth rate are roughly similar as would be expected :

```
   0 2.5 5 10 15 agr every <is3
11.1 8.71 6.51 2.5 _1.04
```

Annuities

These are a special case where the inflows are regular, and so the relevant calculations have closed forms. The factor $\{1\text{-}(1+p)^{-n}/p\}$ is a multiplier which converts payments per period to present value, so an annuity paid for 10 periods at 6% has a present worth per pound given by

```
   AnnFac=.-.@((ptok@[)^-@])%0.01&*@[   NB. Annuity factor
   6 AnnFac 10
7.36
```

as confirmed by

```
   irr _7.36,10#1
6
```

Future Value Analysis and Average Compound Growth Rate

An alternative method for evaluating investments is Future Value Analysis (FVA) which is a form of mirror image of NPV, that is for a given discount rate it is the value necessary in tomorrow's money which is equal to that in today's whereas NPV gives the value of today's money tomorrow.

```
   npv=.] *(ptod@[)^(i.@#@])      NB. net present value
   nfv=.] * ptok@[ ^ i.@-@#@]     NB. net future value
```

The examples below illustrate how `npv` 'anchors' values in the first period whereas `nfv` anchors them in the last period.

```
   is
_100 20 50 70 80
   10 npv is
_100 18.1818 41.3223 52.592 54.6411
   10 nfv is
_146.41 26.62 60.5 77 80
   (10 nfv is)*100%146.41
_100 18.1818 41.3223 52.592 54.6411
```

npv and nfv are inverse in the sense that if the nfv series is scaled down to the initial value of the npv series, the two are identical. Also sfince the between item ratios are unchanged there is no change to the IRRs :

```
   ;irr every (10 nfv is1);10 npv is1
18.3545 18.3545
```

With no discounting each original unit in is has grown to 2.2. With 'reverse discounting' at 10% as in nfv each original 100 units have been shown above to have grown to +/26.62+ 60.5+77 80 = 244.12 units. This is expressed in

```
   ret=.((+/@:}.@nfv) % -@{.@])   NB. return per unit
```

Here is the result of doing this for the series is3 :

```
   4 ret is3
3.02
```

meaning tthat each original unit has grown to £3.02. This is the 10th root of 3.02 = 1.1169 which is therefore the **average compound growth rate** of is3 over the period of the stream. Define

```
   acg=.ktop@(<:@#@] %: ret)      NB. avge compound growth

   4 acg is3                      NB. equivalent % comp grwth
11.7
   4 agr is3                      NB. average growth rate
7.37
```

The interpretation of ACG is that when the final value of an income stream is fixed, for a given inflation rate the ACG is the average rate which must be applied to previous elements in the stream in order to achieve the final value. This might be an appropriate analysis in e.g. pension planning, or in determining the initial investment and return for the reinvestment of inflows necessary to achieve a final target. In the example above 11.7% reinvestment is required to combat an inflation rate of 4%. This can be compared with an IRR of 13.5%.

Here is another comparable dividend stream :

```
   is4=._27.9 1.5 1.5 1.6 1.7 1.8 1.8 1.95 2 57.4
```

which is lower on all these measures :

```
   (4 ret is3),(irr is3),(4 acg is3),(4 agr is3)
3.015 13.46 11.67 7.373
   (4 ret is4),(irr is4),(4 acg is4),(4 agr is4)
2.647 12.61 11.42 7.135
```

whereas the following stream is higher in all values :

```
   is5=._29.5 1.5 2 2 2.5 3 4 4.5 75.3
   (4 ret is5),(irr is5),(4 acg is5),(4 agr is5)
3.307 17.97 16.12 11.66
```

thus demonstrating that the various techniques lead to different values but in general lead to the same ordering.

Code Summary

```
   irr=.dtop@pos@real@roots              NB. internal rate of
return
     dtop=.100&*@<:@%              NB. discount factor to %age
     pos=.#~ >&0                   NB. select positive values
     real=.#~ (= +)                NB. select real values
     roots=.>@{:@p.                NB. extract list of roots
   npv=.] *(ptod@[)^(i.@#@])       NB. net present value
      ptod=.%@>:@(0.01&*)

   bcr=.(+/%costs)@:npv            NB. net benefit cost ratio
      costs=.+/@:(0&>.@-)          NB. outflows (i.e. -ve values)

   nfv=.] * ptok@[ ^ i.@-@#@]      NB. net future value
      ptok=.>:@*&0.01

   acg=.ktop@(<:@#@] %: ret)             NB. avge compound
growth
   agr=.ktop@((<:@#@])%:>:@bcr)          NB. average growth rate
per cent

   AnnFac=.-.@((ptok@[)^-@])%0.01&*@[    NB. Annuity factor
  ret=.((+/@:}.@nfv) % -@{.@])           NB. return per unit
```

Data Streams

```
   is=._100 20 50 70 80
   is1=._100 20 45 60 _10 55 70 20 10 10
   is2=._100 20 45 60 _10 10 10 20 70 55
   is3=._33.5 2 2 2 2.5 3 4 4.45 5.25 4.55 65.88
   is4=._27.9 1.5 1.5 1.6 1.7 1.8 1.8 1.95 2 57.4
   is5=._29.5 1.5 2 2 2.5 3 4 4.5 75.3
```

TOPIC INDEX

A
accept-reject technique, 35
alternative verbs, 12
alternating group, 30
altitude, 46
annualized percentage rate (APR), 49
annuities, 10, 50
ANOVA, 29
anti-cyclic groups, 26
apb notation, 46
autostereograms, 3
average compound growth (ACG), 50
average growth rate (AGR), 50
azimuth, 48

B
balanced rounding, 24
benefit cost ration, 50
betting methods, 14
betting systems, 14
between groups sums of squares, 29
binding, 6
Binet formula, 42
binomial coefficients, 42
binomial therem, 37
bits to integer conversion, 4
bookmaker's odds, 14
Boolean verbs, 16
boundary values, 28
Box-Muller formula, 28
bridge hook, 2, 36, 42

C
Cancellation, 44
Cartesian coordinates, 13
Cap, 6
Catalan numbers, 44
celestial harmonies, 47
celestial meridian, 48
celestial sphere, 48
cell, 20
cents (in music), 47
Chinese remainder problem, 41
chromatic scale. 47
circle of fifths, 47
ciphers, 39
cleaning small numbers, 11
clock arithmetic, 32, 41, 47
clock multiplication, 39

cofactors, 27, 48
collating sequence, 7
combinations, 33, 34, 44
commutativity, 6
complex conjugates, 13
complex logarithms, 13
complex powers, 13
compound interest, 10, 12
cones, 43
congruences, 41
conjugations, 5
conjunctions, 2
connectivity, 38
continued fractions, 42
contradiction, 16
convergence, 9
conversion rules, 15
cost benefit ratio, 50
critical path, 38
cross product, 48
currying, 10, 12
cyclic groups, 26

D
d'Alembert's system, 14
de Moivre's theorem, 13
declination, 48
decryption, 40
derangements, 30
determinant, 13, 27, 48
diagonals of arrays, 8
diatonic scale, 47
dihedral groups, 26, 30
direction cosines, 48
distance tables, 37
distance, 2
dominant, 47

E
eigenanalysis, 9
encryption, 40
enlargements, 48
equal temperament, 47
error control, 12
Euler's phi, 32
exponential ciphers, 39

F
factorial digits, 33
fantasy betting, 14
Farey series, 25
feasibility, 38

Fermat's little theorem, 32
Fibonacci numbers, 37, 42, 43
fill characters, 1, 4
finite arithmetic, 32
fork, 2
frequencies (music), 47
frequency distributions, 28

G
GCD, 41
generating functions, 37
geometric mean, 2
gerund, 4,11, 18, 24, 26, 32, 33, 37, 38
golden angle, 43
golden ratio, 43
greedy algorithm, 37
grouping, 28
groups, 13, 30

H
harmonic mean, 2
heterogeneous arrays, 1
hour angle, 46
hypercomplex numbers, 13

I
identity matrix, 9, 13, 17, 46, 48
identity element, 26
income streams, 49, 50
infinity, 9, 12
inheritance, 2, 5, 6
inner product, 1, 9, 13, 23, 38, 46
interactions, 29
internal rate of return (IRR), 49, 50
intervals (music), 47
inverse permutations, 8
inverse, 12, 26, 39
inverses in finite arithmetic, 41
isomorphism, 26

J, K
Johnson ordering, 33
just intonation, 47
keywords (in encryption), 40

L
Latin square, 29
left binding, 9
Lehmer's algorithm, 33
lexical ordering, 33
list constructor, 18
logarithms, 13

loops, 38
lower triangular matrix, 46
Lucas numbers, 42

M
Manhattan diagram, 44
Martingales, 14
mappings, 8
matrix inverse, 20
matrix multiplication, 9, 13
mean tone temperament, 47
mean, 2 ,7
median, 7
merged axes, 8
merging lists, 22
minors (of a matrix), 19
minors, 27, 47
moments, 2
mood, 6
multiple choice tests, 11
multiplicative inverse, 39

N
nearest neighbour, 38
negative exponential, 14
net future value, 50
net present value, 10, 49, 50
networks, 38
normalisation, 2, 9, 14, 48
noun rank, 20
numerical data types, 15

O
occurrence numbers, 17
octave, 47
odds, 14
orthogonality, 29
overround, 14

P
parity, 30, 33
partitioning, 37
Pascal triangle, 34, 42
permutation list, 33
permutations, 30, 33
polar coordinates, 13
polynomial factors, 45
polynomial multiplication, 45
polynomial quotients, 45
population forecasts, 9
powers, 13
predicate, 16

primeness, 32
private keys, 39
public keys, 39
Pythagorean comma, 45
Pythagorean tuning, 45

Q
quadratic congruences, 41
quadratic residues, 41
quaternions, 13

R
race cards, 14
ragged arrays, 1
random angles, 35
random exponential, 28
random networks, 38
random Normal, 28
random numbers, 35
random races, 14
random sentences, 36
random words, 36
rank inheritance, 2, 6
rank, 2
ranking, 7, 17
rational approximations, 15, 25
reachability, 38
rectangularity, 4
recursion, 23, 27, 33, 37
reflections, 7, 13
residual sums of squares, 29
right ascension, 48
ripple shuffle, 31, 32
roots of equations, 45
rotations, 7, 13, 48
rounding 28
row and column headings, 22
row and column proportions, 22
RSA ciphers, 39

S
Sapir-Whorf Hypothesis, 21
savings schemes, 10
scalarisation, 4
scans, 12
schoolmasters rank, 7
SELECT (SQL), 14
shortest path, 38
sigma, 32
similitude, 46
simulation, 14, 35, 36
simultaneous linear congruences, 41

slack, 38
sorting, 7
spherical trigonometry, 46
SQL, 8
square roots in finite arithmetic, 41
statement separator, 6, 18
Stern-Brocot trees, 25
subgroups, 26
subtotallng, 27
syllogism, 16
symmetric networks, 38
symmetry test, 8

T
tables, 12, 20, 44
tau, 32
tautology, 16
tied ranks, 7
ties, 7
time, 34
Tompkins-Page ordering, 33
totient, 32
trace, 34
trains, 6
transformations, 2, 8
transit, 46
transition matrices, 9
transitivity, 6
transpose, 33
treatment sums of squares, 29

U, V, W, Z
upper triangular matrices, 25, 44
valence, 2
verb rank, 5, 20
Vigenère table, 40
weighted graphs, 38
weighted random numbers, 14
within groups sums of squares, 29
Wolf fifth, 47
zigzag matrix, 18

VOCABULARY INDEX

name	sym-bol	principal references	part of speech
adverse	`::`	12	conjunction
agenda	`@.`	4, 33, 37, 38	conjunction
alphabet	`a.`	7, 30, 40	noun
amend	`}`	11	adverb
anagram	`A.`	30, 34, 40	verb
anagram index	`A.`	30, 33, 34	verb
angle	`r.`	13	verb
anti-base	`#:`	16	verb
append	`,`	3, 22, 40	verb
appose	`&:`	2	conjunction
at	`@:`	2	conjunction
atop	`@`	1, 2	conjunction
base	`#.`	23, 28	verb
basic characteristics	`b.`	2, 12	adverb
behead	`}.`	23	verb
bond	`&`	2	conjunction
Boolean	`b.`	16	adverb
box	`<`	20	verb
cap	`[:`	6	verb
circle function	`o.`	35, 48	verb
complex	`j.`	13, 14, 46	verb

compose	&	1	conjunction
copy	#	22, 28, 29	verb
curtail	}:	33, 37	verb
cycle direct	C.	30, 32	verb
deal	?	7, 11, 28, 33, 36	verb
decrement	<:	12, 34	verb
derivative	D.	23	conjunction
drop	}.	33	verb
evoke gerund	`:	46	conjunction
extended precision	x.	15	verb
factorial	!	30	verb
foreign	!:	34	conjunction
format	"	22	verb
from	{	22, 23	verb
GCD	+.	13, 32, 40	verb
gerund	`	18, 26, 32, 33, 37, 38	conjunction
grade down	\:	6, 7, 14, 26, 35, 47	verb
grade up	/:	6, 7, 17, 18, 22, 26, 31, 32, 40	verb
halve	-:	23, 41	verb
head	{.	23, 41	verb
imaginary	j.	46	verb
increment	>:	12, 23, 34	verb
index of	i.	30, 40	verb
infinity	_	9, 23	noun
infix	\	18, 25, 34, 42, 47	adverb
interval index	I.	14, 28	verb

item amend	`{.`	11	adverb
key	`/.`	12, 17, 29	adverb
laminate	`,:`	3, 22	verb
LCM	`*.`	13, 32	verb
left	`[`	2, 34, 39	verb
less	`-.`	2, 3, 45	verb
link	`;`	6, 22	verb
log	`^.`	2	verb
magnitude	`|`	23	verb
match	`-:`	2, 38	verb
matrix divide	`%.`	2	verb
matrix inverse	`%.`	2, 45	verb
member in	`e.`	32, 38	verb
member of interval	`E.`	14, 36	verb
not	`*.`	32	verb
nub	`~.`	17, 32	verb
oblique	`/.`	19, 42, 45	adverb
obverse	`:.`	12	conjunction
outfix	`\.`	34, 48	adverb
passive	`~`	2, 27, 42, 47	adverb
permute	`C.`	30	verb
pi times	`o.`	13	verb
polar	`r.`	13	verb
polynomial	`p.`	37, 45	verb
power (conjunction)	`^:`	9, 10, 12, 23, 25, 31, 32, 40, 42	conjunction

<table>
<tr><td>prefix</td><td>\</td><td>34, 42</td><td>adverb</td></tr>
<tr><td>prime factors</td><td>q:</td><td>32</td><td>verb</td></tr>
<tr><td>rank</td><td>"</td><td>1, 4, 6, 7, 8, 11, 20, 22, 26, 27, 42, 46</td><td>conjunction</td></tr>
<tr><td>ravel</td><td>,</td><td>3</td><td>verb</td></tr>
<tr><td>ravel item</td><td>,.</td><td>3</td><td>verb</td></tr>
<tr><td>raze</td><td>;</td><td>4, 17, 29, 34</td><td>verb</td></tr>
<tr><td>real/imaginary</td><td>+.</td><td>46</td><td>verb</td></tr>
<tr><td>reflex</td><td>~</td><td>17, 26, 34, 35, 39, 42, 45</td><td>adverb</td></tr>
<tr><td>residue</td><td>|</td><td>23</td><td>verb</td></tr>
<tr><td>reverse</td><td>|.</td><td>23</td><td>verb</td></tr>
<tr><td>right</td><td>]</td><td>2, 18, 34, 39</td><td>verb</td></tr>
<tr><td>roll</td><td>?</td><td>14, 35, 36</td><td>verb</td></tr>
<tr><td>rotate</td><td>|.</td><td>23</td><td>verb</td></tr>
<tr><td>self-classify</td><td>=</td><td>9, 17, 21, 46</td><td>verb</td></tr>
<tr><td>shift</td><td>|.</td><td>6, 10, 26, 40</td><td>verb</td></tr>
<tr><td>stitch</td><td>,.</td><td>3, 22, 40, 46</td><td>verb</td></tr>
<tr><td>suffix</td><td>\.</td><td>18, 19, 27, 34</td><td>adverb</td></tr>
<tr><td>tail</td><td>{:</td><td>23, 37</td><td>verb</td></tr>
<tr><td>take</td><td>{.</td><td>22, 23, 25, 28, 33</td><td>verb</td></tr>
<tr><td>tally</td><td>#</td><td>8, 22, 36</td><td>verb</td></tr>
<tr><td>Taylor coefficients</td><td>t.</td><td>37</td><td>verb</td></tr>
<tr><td>tie</td><td>`</td><td>4, 46</td><td>conjunction</td></tr>
<tr><td>transpose</td><td>|:</td><td>7, 8, 16, 19, 26, 27, 33</td><td>verb</td></tr>
<tr><td>under</td><td>&.</td><td>1</td><td>conjunction</td></tr>
</table>