

At Play With J

The complete *Vector* articles

by

Eugene E. McDonnell

Revised and annotated by
the worldwide J community



Copyright © Eugene E. McDonnell, 1993-2009.

Vector Books

11 Auburn Road, Redland, Bristol, BS6 6LS, England.

in conjunction with the British APL Association.

<http://www.vector.org.uk>

Second Edition

All Rights Reserved.

The right of Eugene E. McDonnell to be identified as the author of this work has been asserted by him in accordance with the UK Copyright, Design & Patents Act, 1988.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written consent of the publisher or a licence permitting copying in the UK issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1P 9HE.

ISBN 978-1-898728-18-4

Typeset in Book Antiqua and APL 385 Unicode

by Undead Tree Publications

<http://www.undeadtree.com>

To my wife Jeanne Farr McDonnell.

Contents

Preface to the First Edition by Roger Hui	9
Preface to the Second Edition by Chris Burke	10
Introduction	11
Acknowledgements	13
Typographical note:	14
1 MIMD Machines	15
2 Factoring a Number	17
3 The 10,000,000,000th Prime Number.....	25
4 Control Structures	31
5 Jacobi’s Method	35
6 Cribbage 15s.....	45
7 Representing a Permutation.....	51
8 The Bauer-Mengelberg Problem.....	55
9 Heron’s Rule and Integer-Area Triangles.....	65
10 Year’s Digits for 1996	75
11 Riding a Unicycle	79
12 Volutes	85
13 Extended Integers	97
14 Stumping the Rocket Scientist	107
15 Oh, No, Not Eigenvalues Again!	115
16 A Newer Random Link Generator	121
17 To Summarise.....	131

18	Maximum Infix Sums	137
19	Crosswords and Life	143
20	New Model Computer	151
21	New Big Deal.....	155
22	We'll Cross That Bridge When We Come To It	163
23	An Open and Shut Case	169
24	Blists in OEIS	179
25	Someone Just Moved! Who Was It? Or, Apter's Puzzle	191
26	Four Cubes Redux	205
27	Erdős Numbers and Pierce and Engel Expansions.....	213
28	Boggle	227
29	The Counterfeit Coin Problem	241
30	Second Order Josephus.....	253
31	J be Nimble, J be Quick: Nim Addition.....	259
32	Beware Scholes	273
33	Pick A Card, Any Card	281
34	Greed.....	287
35	The Magical Matrix	293
36	Giddyap.....	299
37	Jacob's Ladder	307
38	The Google Test	323
39	Metlov's Triumph	331
40	Belgian Numbers.....	337

41	Token Counting: APL versus J	343
	Answers to Problems.....	351
	References	355

Preface to the First Edition by Roger Hui

In my youth, when I was just starting in APL, on receiving an issue of the *APL Quote-Quad* I would inevitably and eagerly first turn to Eugene McDonnell's "Recreational APL" column. Through these columns I learned that it was possible for technical writing to be erudite, educational, and entertaining, and through them I learned a lot of APL.

Thus it was with Eugene's *At Play with J* articles in *Vector*. In topics ranging from primes to permutations to pyramids to π , with a cast of characters that included Apter, Black, and Crelle, Jacob and Josephus, Blanda and Montana and Taylor, and Scholes, the articles offered up the "smoother pebbles" and "prettier shells" found while playing on the seashore bordering the great ocean of knowledge. And we are all beneficiaries of this play.

I am pleased that *Vector* is publishing the collection of *At Play with J* as a book. I look forward to being educated and entertained once more.

Roger Hui
January 2009,
Vancouver, Canada.

Preface to the Second Edition by Chris Burke

The first edition of *At Play With J* reprinted the articles essentially as they first appeared in *Vector*, the journal of the British APL Association (BAA). As such, it remains a fascinating historical document with one grave drawback: evolution of the language has made some of the code obsolete. Small changes to core grammar and syntax have invalidated entire programs. And for some problems, extensions to the language now allow much better solutions to be written.

The book was of such potential value to the J beginner that it merited a second edition to bring the code examples up-to-date. The BAA Projects Officer Ian Clark invited the active participation of the J community by placing each article on the J wiki, thereby encouraging several experienced J developers to review the articles, update the code and add supplementary information. The result is a completely revised edition with turnkey code for the latest release of J (6.02).

On behalf of the J team, I would like to thank Ian for an outstanding effort in preparing this new edition, and the members of the J community whose contributions made this possible.

Chris Burke
November 2009,
Hong Kong.

Introduction

In 1993 Eugene McDonnell published an article in *Vector*, the journal of the British APL Association, disarmingly entitled *At Play With J*.^{*} Little did anyone suspect that this was the beginning of a brilliant series which would continue until August 2006, at which time Eugene let it be known that he could no longer commit to writing any more contributions.

Here are Eugene's forty-one fine articles reprinted in one volume. They form a series of straightforward but profound mathematical investigations which not only entertain but exercise the intellect.

Do you need to know J to read this book?

When buying any book on a serious topic, obviously you ought to be ready to open your mind to new ideas. This includes taking the trouble to grasp any special notation the author thinks fit to introduce. From this point of view the answer is *no* — you don't need any prior knowledge of J. The author instructs you with sufficient examples in all the notation he needs as he goes along, though it's best to start with the first four chapters or so.

Eugene's "special notation" is ultra-terse — with a superb computer-aid for playing with it: the *J Interpreter*. This comes with the *J Introduction and Dictionary*, to which frequent reference is made. You can download both **free of charge** from <http://www.jsoftware.com> to run on a Macintosh™ or a Windows™ computer.

With these two items you'll have no trouble following Eugene's line of reasoning, to learn a host of amazing facts about number-crunching and puzzle-solving. And, as a bonus, maybe you'll find you've armed yourself with a useful new tool of thought, even if you haven't set out with the intention of "learning yet another programming language".

So... let's play!

^{*} *Vector*, 10, 2, (October 1993), 128-129. ISSN 0955-1433.
Reprinted in this book as Chapter 1: MIMD Machines.

Acknowledgements

This, the Second Edition of *At Play With J*, wouldn't have been possible without the enthusiastic cooperation of the J community, who took it upon themselves to review and update the articles as Eugene McDonnell originally wrote them, *Vector* published them and the First Edition reprinted them, in the light of subsequent changes to the J language, as exemplified in the release of *J Version 6.02* (2009).

Members of the J community generously gave of their time to execute every example of J code in the J602 session window, to check that it ran true, and to furnish up-to-date replacement code where it didn't. The result has been to turn what was at first intended as a souvenir collection of reprints into a first-class educational tool in the J language, which both J novices, and mathematical researchers with no intention, at the outset, of learning J, can confidently follow without fear of being shunted into a disused siding.

At the head of each chapter in this book is the name of the volunteer who took charge of its code content to vouch for its runtime integrity. These names are listed below. Grateful thanks are due to:

Bill Lam
Boyko Bantchev
Brian Schott
Chris Burke
David Mitchell
Don Guinn
Fraser Jackson
Gilles Kirouac
Ian Clark
Joey Tuttle
Kip Murray
Oleg Kobchenko
Ric Sherlock
Roger Hui
Tracy Harms.

Typographical note:

The J interactive session permits very long lines of code to be inserted. Sometimes a line of code intended by the author is too long for the printed page. In this book, long lines are spilled and a special symbol is used to show where this has happened. This symbol: ➤ is a typographical device which is valid only in the context of the printed page: it is *not* a member of the J character set.

For example, for a single line of code to be spilled like this:

```
CEA =. 3 : 'if. (2|#y) do. ub"2 CEA bz y  
else. PJ y ,:IM y end.'
```

would be ambiguous, suggesting the possibility that two separate lines were intended, whereas:

```
CEA =. 3 : 'if. (2|#y) do. ub"2 CEA bz y ➤  
else. PJ y ,:IM y end.'
```

unambiguously represents a single line of code with no line-break.

Long lines of output in the J session are shown in a similar manner:

```
i. 40  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ➤  
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 ➤  
35 36 37 38 39
```

When copy/pasting sample code from the PDF version of this book into a J session ensure that each pair of lines linked by ➤ is entered as one line, replacing ➤ with a single space.

1 MIMD Machines

First published in Vector, 10, 2, (October 1993), 128-129.

Revised by Ian Clark, (April 2009).

I had a request recently from someone who wanted to apply a verb a different number of times to a list of arguments. What was wanted was a simpler way of writing:

$(f\ a), (f\ f\ b), (f\ f\ f\ c)$

My initial response was to say that J did not as yet have a way of describing Multiple Instruction-Multiple Data machine architectures (MIMD), although such a mechanism had been described [1]. I pointed out that a collapsing transpose could solve the problem, but my questioner would have none of that, as it implied a great deal of useless computation. There the matter rested for a while. After several months I had another request from the same person who wanted to know if I had made any progress on the problem. Actually, I hadn't thought about it at all in the interim, but since my questioner seemed to be a determined type, I gave it a few minutes more thought, and found what I think is a neat use of one of J's more interesting differences from APL, the way scan is defined: that is, the verb applied is monadic, not dyadic.

For example, whereas in APL one writes $+\backslash 1\ 2\ 3$ to obtain the continued sum of the values in the argument, in J one would have to write $+\backslash 1\ 2\ 3$ to obtain the same result.

```
+ \ 1 2 3
1 0 0
1 2 0
1 2 3
```

Here the monadic verb *conjugate*, denoted by $+$, is being applied, first to 1, next to 1 2, and last to 1 2 3; since these are real numbers, their conjugates are the same as the arguments, and since J reshapes results so that they conform, and then appends them, we get the zero fills at the right of the top two rows.

Compare this with

```
+ / \ 1 2 3
1 3 6
```

which is the analog to APL's $+\backslash 1\ 2\ 3$.

Finally, here is the solution to the MIMD problem.

First I define three variables, **a**, **b**, and **c**:

```
'a b c'=: 3 4 5
```

Next, I define a verb **f** to be the natural logarithm (\wedge).

```
f=: ^.
```

and apply it once, twice, and thrice, to **a**, **b**, and **c**, respectively:

```
f a
1.09861
f f b
0.326634
f f f c
_0.742579
```

This is the desired result, but done the hard way. Now for the easy way:

Define a verb **g** in which the verb **f** is applied (**@**) to the tail (**{:**) of its argument a number of times (\wedge ;) equal to the length (**#**) of its argument:

```
g=: f@{:^:#
```

For example, **g 3 1 4 1 5 9** applies **f** six times to **9**:

```
f f f f f f 9
0.854804j1.01575
g 3 1 4 1 5 9
0.854804j1.01575
```

Perhaps you already see how this will end. We apply the prefix scan (****) adverb to **g**, and apply this derived verb to **a**, **b**, **c**:

```
(g\)a,b,c ➡
NB. apply g to successively longer prefixes
1.09861 0.326634 _0.742579
NB. q.e.f.
```

Showing once again that where there's a will there's a way. Note that because of the way prefix scan is defined, it is easy to visualize how, in a multiprocessor environment, the applications of **g** to all three arguments can be carried out simultaneously.

Reference

- [1] Bernecky, R., Hui, R. K. W., Gerunds and Representations. *APL Quote Quad* 21, 4, Stanford, Calif., (1991-08), 39-45.

2 Factoring a Number

First published in Vector, 10, 3, (January 1994), 100-105.

Revised by Chris Burke and Ian Clark, (May 2009).

You may be among the readers of *Vector* whose stomachs start churning or whose eyes glaze over when they read the words “tacit definition” in articles about J. This column is meant for you. It is going to take the approach that you know how to write an APL function, and might not be averse to learning how to write a J function in a similar fashion.¹

The example I shall use is fairly short—just twelve lines—and may possibly even be of use to some of you. It was shown to me by Joey Tuttle many years ago. It’s called **factors** and it factors a positive integer *n*. The result of **factors n** is an ordered list of primes such that *n* is the product over the list, that is,

```
n ↔ */ factors n
```

It was able to find the factors of the 18-digit integer $2^{59}-1$ (576460752303423487), which are 179951 and 3203431780337, in 3 min 37.1 s on a Macintosh Quadra 700.²

Various methods of finding the prime factors of a number are given by Knuth [1]. The method used here is the simplest one, Algorithm A, which divides the number by increasingly larger primes until all factors have been found, but the method has been vectorized so that more than one factor may be found at once.

The argument must be a positive integer not greater than **max**, where **max** is a number derived from the floating-point characteristics of your computer. This discussion assumes that your computer uses IEEE floating point arithmetic, which is the case for PCs, Macintoshes, and most Unix machines. On PCs and Unix machines

¹ Even if you don’t know APL, Eugene’s treatment is still readable as an introduction to developing a substantial iterative algorithm in J, notwithstanding the fact that since this article was written computers have become faster and J has acquired a prime factorization primitive (**q:**).

² Nowadays **factors 576460752303423487x** takes 5 seconds to replicate this result on my Mac Mini, bought in 2005. (*Ed.*)

es the maximum is the 16-digit integer $2^{53}-1$ (9007199254740991). For Macintoshes it's the 19-digit $2^{63}-1$ (9223372036854775807), or 1024 times as large. These are the largest integers which can be represented to full accuracy on the hardware of those machines.

First I'll show the function definition, then give explanations of its lines, paying special attention to those things that differ from APL. Let's assume for the moment that you created this function:

```

factors=: 3 : 0
f=. i. 0
t=. 2 3 5 7 11 13 17 19 23 29 31 37
o=. +/\ 432 $ 4 2 4 2 4 6 2 6
whilst. y >: *: {. t do.
    whilst. #m do.
        m=. (-. * t |!.0 y) # t
        f=. f , m
        y=. y % */ m
    end.
    t=. o + {: t
end.
/:~ f , y -. 1
)

```

NB. Line 0
NB. Line 1
NB. Line 2
NB. Line 3
NB. Line 4
NB. Line 5
NB. Line 6
NB. Line 7
NB. Line 8
NB. Line 9
NB. Line 10
NB. Line 11

The line numbers are shown in comments (starting with NB.). These are ignored when the function is executed.

The first thing to notice is that the header line does not indicate whether there is an explicit result or not, nor what its name is if there is one, nor what its valence is, nor what the argument is named, nor which variables are local. Here is how each of these questions is resolved:

Explicit result. Every J function has an explicit result, and it is the value of the last expression executed.

Function name. In J a function is named the same way that a variable is named, by assignment, denoted by ($=:$). We use the “definition” operator, symbolized by the colon ($:$), whose result is a function, not a variable. The left argument of the definition operator is a number indicating the result type—3 means a function. The rest of the text up to the line with a single “)” is the function definition. This creates the function named **factors**.

Valence. For an ambivalent function, the monadic case is given by text up to a line containing $:$ and the dyadic case is given by the remain-

ing text. Either block of text may be empty. There is no such thing as a niladic function in J.

Argument name. By convention, the argument to a monadic function is (y). In a dyadic function the left argument is named (x) and the right argument is named (y).

Local variables. J makes a distinction between local assignment (=.) and global assignment (=:). This removes the need for a list of local names. The rule is that on the first assignment of a name using (=.) the name is made local.

Here beginneth the detailed description of the function's lines:

Line 0 sets the initial factor list to empty. J's "iota" is denoted by (i.). By the way, if the argument is 1, this will be the result as well, since 1 is not a prime and has no prime factors. However, it will still be true that $n \leftrightarrow */ \text{ factors } n$, since the product over an empty vector is 1.

Line 1 creates as the initial list of trial divisors the first 12 primes. This enables the function to use just one iteration for all arguments less than 1370 (37^2 is 1369).

Line 2 forms the list of offsets to be used in creating a new list of trial divisors, after there are no more values left in the current list that divide the current value of the number being factored.

In J, the scan operator is defined differently from the APL scan operator. In APL the scan operator implies the reduction operator. In J the function argument to scan is monadic, not dyadic, so that one has to use "sum" (+/) not "add" (+) if we want the sumscan.

There are 432 items in this vector: 54 repeats of the eight items 4 2 4 2 4 6 2 6. The purpose of this vector is to remove multiples of 2 3 5 from consideration as trial divisors, since these can't be primes. For example,

```
37 + +/\ 4 2 4 2 4 6 2 6 ...
41 43 47 49 53 59 61 67 ...
```

Some of the items in these new trial divisors will not be primes. In the list above 49 is composite. Eliminating multiples of 2 3 5 reduces the number of trial divisors by over 73%.

Line 3 starts with the `whilst.` control word. The control block is the lines up to the matching `end.` control word. Following the `whilst.` word is a test that is executed at the end of the control block, which is described with **Line 10** below.

Line 4 starts with another `whilst.` control word and test, which is described with **Line 8** below.

Line 5 begins the iteration, and forms the vector `m` of newly-found factors of the argument. The factors are found by using the vector of trial divisors of the argument as the left argument to the residue function. However, the residue function in J is fuzzed, just as it is in some APLs, in order to permit proper-looking results when used with near-integers, and also to permit the use of decimal rational numbers as arguments to residue. We are dealing with exact integer arguments, so in order to extend the domain of the residue function we require that it not be fuzzed. To accomplish this in APL systems, the comparison tolerance system variable, `⎕ct`, is set to 0. In J this is done by explicitly modifying the residue function with the “fit” or “customize” operator (`!.`) using 0 as right argument. Thus, instead of writing

```
t | y
```

we write

```
t | !. 0 y
```

and this allows us to work with a residue that has zero fuzz.

Curiously, the *Dictionary of J*, which says that the “fit” operator “modifies certain verbs in ways prescribed in their definitions”, doesn’t describe this use of it with respect to residue. Tsk tsk. The advantage of having the modification of the verb occur in direct connection with its use is obvious: one doesn’t have to remember whether or not `⎕ct` has been modified, nor run into the hazard of not restoring it when it should be. The use is direct and immediate, and applies only to the function in question. Furthermore, anyone reading the function knows immediately that it is unfuzzed.

After finding the residues, their signum (`*`) is taken, yielding a vector of 0s and 1s, and these are negated (`-.`) complementing them to 1s and 0s. This boolean vector is used to select (`#`) the corresponding items from `t`, giving in `m` the new factors to be

appended to the result. In APL there has always been a confusion about slash (/): is it an operator or a function? If we write `+ / 1 2 3` it acts like an operator, but when we write `1 0 1 / 1 2 3` it acts like a function. In J the “#” function is used for the functional case, as in `1 0 1 # 1 2 3`.

Line 6 appends the new factors to the result vector.

Line 7 factors the argument, by dividing (%) it by the product (* /) of the new factors, thus diminishing it.

Line 8 ends the innermost control block. The test `#m` is false if 0, and true otherwise. Thus the control block is repeated until `m` is empty.

Line 9 forms a new vector of trial divisors, by adding the offset vector to the last (`{:`) item of the current vector.

Line 10 ends the outermost control block. Execution is continued if the reduced argument is greater than or equal to (`>:`) the square (`*`) of the first item (`{.`) of the new vector of trial divisors, since in this case there may be more factors. If it is less, this means that there can't be any new factors in the reduced argument, and thus it is either 1 or a prime.

Line 11 removes 1 from the argument (the result of `a -. b` is `a` with items equal to `b` removed; `17 -. 1` is 17; `1 -. 1` is empty). This leaves it unaltered if it is a prime, or makes it empty otherwise. It then appends the argument to the list of factors (essentially doing nothing if it had been 1), sorts (`/:~`) the list into ascending order, and terminates.

In J the semantics of dyadic upgrade and downgrade have been changed. They no longer have the significance of using the left argument as a collating sequence to produce a grade with reference to it. Instead, they are used to sort the left argument into an order specified by the right argument. The definition of dyadic upgrade (`a /: b`) is

`(/: b) { a`

that is, the permutation that puts **b** in ascending order is used to permute the items of **a**.

The most frequent use of these sort verbs is with the left and right arguments identical, in which case the result is the sorted argument, either ascending or descending. The reflexive operator (**~**) applies to a monadic verb to produce a dyadic verb with left argument the same as the right argument. For example, **+~1.2** is **2.4**, and

```
/:~ 2 7 1 8 2 8
1 2 2 7 8 8
```

Below are some examples of the use of the **factors** function with some ten-digit numbers to give you some idea of how it behaves. Notice that the time to factor a number is longest when the argument is a prime, and fairly long also when there are two factors roughly equal to the square root of the argument.*

```
time=: 6!:2 ➡
NB. yields seconds to execute its string argument
fmt=: ":!.20 ➡
NB. formats (":) numbers to 20 places (!.20)

time 'k=: factors 6307059899'
1.11667
fmt k
7 19 47421503

time 'k=: factors 6307059901'
0.85
fmt k
379 16641319

time 'k=: factors 6307059903'
0.983333
fmt k
3 127 3461 4783
```

* The reader is invited to compare these results with those obtained on an up-to-date computer. For the final example, 6307059911 (a prime number), my Mac Mini (2005) timed the result at 0.011236 s (and 0.055676 s with **q:** in place of **factors**). (Ed.)

```
time 'k=: factors 6307059907'  
0.65  
fmt k  
1201 5251507
```

```
time 'k=: factors 6307059909'  
3.51667  
fmt k  
3 24749 84947
```

```
time 'k=: factors 6307059911'  
10.0167  
fmt k  
6307059911
```

Reference

- [1] Knuth, D., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*. Third Edition, Reading, Massachusetts: Addison-Wesley, (1997), Section 4.5.4, “Factoring into primes”. ISBN 0-201-89684-2.

3 The 10,000,000,000th Prime Number

First published in Vector, 10, 4, (April 1994), 110-113.

Revised by Roger Hui, (April 2009).

What is the 10,000,000,000th prime number?

This column tells a story, and it has a moral. It does not concern itself directly with J, the ostensible reason for these columns, but it can be justified because one of the direct antecedents of J is the language A, developed by Arthur Whitney while he worked at the investment banking firm of Morgan Stanley. It was one page of C code for an A-like interpreter, written one afternoon by Whitney at Ken Iverson's Kiln Farm in Ontario [1], that gave Roger Hui the direction he needed to start work on what was to become J. Roger exhibits this code in Appendix A of *"An Implementation of J"*, published by Iverson Software Inc.

Whitney no longer works at Morgan Stanley: he has set out as a freelance and is developing a language called K, which has some affinity with A and J. Hui now works at Morgan Stanley, and it is his adventure hunting down a large prime that the story is about, and Hui used A as the weapon with which he targeted the large prime.

The story begins when Hui's boss at Morgan Stanley challenged him by saying, "You think you're smart, but you don't even know what the 10-to-the-10-th prime is." Hui's immediate response was, "Do you start counting from 0 or 1?" The boss was so taken aback that for a minute or so he didn't understand Roger's question.

The boss's challenge seems to have been meant as an example of a theoretically attainable but practically impossible computational task. This article tells how Roger went about achieving the impossible. I look on it as a triumph of the client-server technology. This column is not so much an article about programming as it is about computer logistics. The programming aspects, while important, are secondary to the story of how Roger went about organizing the solution.

The germ of Hui's solution was to envision a Boolean vector p of length k such that the i th element of p is 1 if i is prime, and 0 otherwise. Just sum-scan this very long vector and look for the

index of $1e10$ in it. How long should such a vector be? The Prime Number Theorem says that the number of primes less than k is roughly $k\%(\log k)$. Solving for k in the equation $1e10 = k\%(\log k)$ gives a value for k about $2.63e11$. Roger, out of prudence, used the value $2.7e11$. A vector of $2.7e11$ elements is unrealizable in the present state of computer memories, especially since A doesn't have a Boolean type: Boolean vectors require the same space as integer vectors. A vector of $4 * 2.7e11$, or $1e12$ bytes long is simply not on the cards. Even a Boolean vector taking just 1 bit per element would have to be more than $3e10$ bytes long, so it was clear that the problem had to be partitioned.

Hui's central program computes the primes between m and n , using the sieve method, eliminating multiples of 2, 3, 5, 7, 11, 13, 17, etc. This can be done independently in parallel on many small intervals that make up the larger interval of interest, and, if portioned out to computers that can communicate with a common central file, will permit the problem to be solved in a shorter time than if only one computer were to tackle it.

The method of partitioning was suggested by the presence of more than 150 workstations on the floor in Hui's part of Morgan Stanley. They are all interconnected Unix machines, and any machine can be used from any other machine. With relatively small effort a multi-processor solution could be set up, using these machines in parallel. The machines are not heavily used at weekends, and it was on a weekend that the experiment took place.

The strategy was to let the machines tackle the problem a billion integers at a time. Hui's colleague at Morgan Stanley, Seth Breitbart, suggested creating 271 files, named 0, 1, 2, ..., 270, each denoting the named interval of $1e9$ consecutive numbers, and each one empty.

What a machine would do once it was set going was to look at the list of files, pick one at random (named m), erase it, work on the interval $(1e9 * m) + i . 1e9$, a million numbers at a time, and after finishing, write a file containing a record giving the number of primes in each of the million-number intervals within that $1e9$ (there are a thousand of them). After finishing, it repeats that process, stopping only when the list of files/intervals is empty.

The machines were set up to process a million numbers at a time since the smallest machine available had enough memory to handle that many numbers at once. Roger notes that there's no great harm if two machines accidentally happen to pick the same file/interval. In the flexible Unix universe additional machines could be brought on stream at any time.

If one is a Unix system superuser it is possible to take all sorts of liberties with these machines, like finding the ids of all other machines, but Hui prefers (wisely, I think) not to be tainted by such capabilities, so to get the machines' names he went about the floor reading the names of the machines from strips of paper affixed to each one, then sat down at his machine and made inquiries about the state of each machine on his list. If it was idle, he set it going on the problem.

As he was doing this, there was a nice dilemma to resolve. Should his time be spent improving the algorithm before launching more machines, or should he spend time looking for additional machines? He favoured the latter approach for the novelty of it, and ended up using about 15 IBM RS/6000s and 60 Sun Sparc 2s and Sparc 10s.

After 20 hours, he had 271 files, each with 1,000 records. From these he made a 271,000-element vector of the number of primes in the intervals $1e6 \cdot i \dots 271000$. By sum-scanning this he knew the interval containing the 10^{10} -th prime. His function `psieve` returns a Boolean list selecting the primes between m and n . Applying this to the magic interval gets the actual 10^{10} -th prime.

Some details about his program "sieve": If a number q is not divisible by any number less than or equal to $\text{sqrt}(q)$, then q is prime. Therefore, to test a number less than $2.7e11$ for primality one need only use trial divisors less than $\text{sqrt}(2.7e11)$ or roughly $6e5$. In practice, Hui precomputed a list of all the 78,498 primes less than $1e6$ recursively, bootstrapping up from 2 3 5 7. (This only takes a few seconds.) It was then a routine matter to determine for any number less than $1e12$ whether or not it was prime: just see whether its

residues with respect to each of the primes less than a million was nonzero; if so, it was a prime.

For the curious, here is a condensed list of the first 10,000,000,002 primes, with their 0-origin ordinal numbers.

0	2
1	3
2	5
3	7
4	11

...

1e10-1	252,097,800,623
1e10	252,097,800,629
1e10+1	252,097,800,637

After doing this, Hui found that there is a table in William Judson LeVeque's *"Fundamentals of Number Theory"*, section 1.1, giving the number of primes less than 10^{3+i} . Hui's table agrees with LeVeque's for 10^{3+i} . For 10^{10} , however, LeVeque says 455,052,512 and Hui says 455,052,511. It turns out that LeVeque is wrong, Hui having checked his results with some help from Lee Dickey at Waterloo University. Dickey tells Hui that his colleagues speculate that LeVeque may have gotten his numbers from lists that D.N. Lehmer compiled, which included 1 as a prime, and LeVeque may have slipped in not subtracting 1 from that particular count. (1 isn't a prime since it doesn't satisfy the definition of a prime: a positive integer n with exactly two distinct factors, 1 and n .)

Now for the moral of the story: Hui tells me he has also since found some work that would have made it much easier to discover the n th prime, for any n .

E.D.F. Meissel, a German astronomer, found in the 1870s a method for computing individual values of $\pi(x)$, the counting function for the number of primes $\leq x$. His method was based on recurrences for partial sieving functions, and he used it to compute $\pi(10^9)$, where π is a function that computes the number of primes less than or equal to its argument.

D.H. Lehmer simplified and extended Meissel's method. Recently, further refinements of the Meissel-Lehmer method which

incorporate some new sieving techniques have been reported by Lagaria *et al* [2]. In this article the authors give an asymptotic running time analysis of the resulting algorithm, showing that for every $\epsilon > 0$ it computes $\pi(x)$ using at most $O(x^{(2/3)+\epsilon})$ arithmetic operations and using at most $O(x^{(1/3)+\epsilon})$ storage locations on a computer using words of length $1 + \epsilon \cdot 2^x$ bits.

The algorithm can be further speeded up using parallel processors. They show that there is an algorithm which, when given M parallel processors, computes $\pi(x)$ in time at most $O((M/x)^{2/3+\epsilon})$ using at most $O(x^{(1/3)+\epsilon})$ storage locations on each parallel processor, provided $M < x^{1/3}$. A variant of the algorithm was implemented and used to compute $\pi(4 \times 10^{16})$.

They report that $\pi(4 \times 10^{16})$ took them 1730 minutes on an IBM 3081K; $\pi(2 \times 10^{12})$ took 3 minutes; $\pi(3 \times 10^{12})$ took 4 minutes. Had he known about this method, Hui could have used a binary search technique to find the 1×10^{10} -th prime, using an $O(2 \log n)$ technique, after narrowing the interval to be searched in by a reasonably generous use of the Prime Number Theorem. The value Hui computed was $\pi(2.52 \times 10^{11})$ and it took him 20 hours on 60-70 workstations. Brains win again over brawn: a well-designed, mathematically knowledgeable algorithm beats brute force!

References

- [1] <http://www.jsoftware.com/jwiki/Essays/Incunabulum>
- [2] Lagaria, Miller, Odlyzko, Computing $\pi(x)$: Meissel-Lehmer Method. *Mathematics of Computation* 44, 170, (1985-04), 537-560.

4 Control Structures

First published in Vector, 11, 1, (July 1994), 136-138.

Revised by Roger Hui, (April 2009).

There have been many proposals for control structures in APL systems before, and there are now current several APL systems which have them. This article will describe the control structures which are made available in the new release of J (version 2). This article assumes that you, like me, are not overly familiar with the general notion of control structures. If in fact you are a long-time user of Fortran or Algol or C, please forgive these callow comments.

The control words introduced are:

```
break.  
catch.  
continue.  
do.  
else.  
elseif.  
end.  
goto_<name>.  
if.  
label_<name>.  
return.  
try.  
while.  
whilst.
```

The four control words `if.`, `try.`, `while.`, and `whilst.` mark the beginnings of control structures that are each terminated by a matching `end.` control word.

The control words `while.` and `whilst.` differ in that the test block in a `whilst.` statement is skipped the first time (the “`s`” in `whilst.` can be thought of as meaning “skip test”) whereas in a `while.` statement it is always executed. As a consequence, the execution block in a `whilst.` statement is always executed at least once, but in a `while.` statement, it may execute zero times.

The words `do.` and `else.` and `elseif.` occur within control structures, separating them into blocks. The control word forms `goto_<name>.` and `label_<name>.` represent an infinite family of

possible control words, for each of which is a different text. For example, one may write:

```
goto_ahead.  
...(statements)...  
label_ahead.
```

or:

```
label_back.  
...(statements)...  
goto_back.
```

A block consists of zero or more control words and sentences that are grouped together by control words occurring within a control structure. The role of blocks is summarized as follows:

```
if. T do. B end.  
if. T do. B else. B1 end.  
if. T do. B elseif. T1 do. B1 ... →  
elseif. Tn do. Bn end.  
try. B catch. B1. end.  
while. T do. B end.  
whilst. T do. B end.
```

Words with **B** or **T** denote blocks. If the first (or only) atom of the result of the last sentence executed in a **T** block is zero, the **B** block following is not executed, otherwise it is executed.

In a series of **elseif. Ti do. Bi**, if the **Ti** are not exhaustive, it is good practice to put a final **elseif. 1 do. Bz**, where **Bz** is a block covering the default procedure when all else has failed, so that **Bz** is executed when no other test has succeeded.

Perhaps an example will make some of these details more concrete. The program `p23` represents a crude but effective process for determining x as the two-thirds power of y for y any positive cube. For reference purposes, line numbers are shown as comments (starting with **NB.**).

```
p23=: 3 : 0  
v=.0  
w=.1  
while. y~:z=.v*x=.v*v do.  
if. z>y do. v=.v-w=. -:w  
else. v=.v+w=. +:w  
end.  
end.  
x  
)
```

NB. Line 1

NB. Line 2

NB. Line 3

NB. Line 4

NB. Line 5

NB. Line 6

NB. Line 7

NB. Line 8

$p_{23} 7^3$ NB. should give 7^2
 49

This program was converted from Program 1.4 on page 4 of Kenneth Iverson's *A Programming Language* [1], written in what was then called "Iverson notation".*

Lines 1 and 2 give initial values to the local variables *v* and *w*.

Lines 3 through 7, inclusive, are a *while*. statement.

Lines 4 through 6 inclusive are an *if*. statement.

The *T* block in the *while*. statement compares the argument (*y*) for inequality with *z*, which is the cube of *v*. If they are unequal, the result of the *T* block will be 1 (nonzero) and the *if*. statement will be executed.

The *T* block in the *if*. statement determines whether *z* is greater than *y*, and if it is the block following *do*. will be executed. Otherwise, the block following *else*. will be executed.

The block following *do*. halves *w* and subtracts this from *v*; the block following *else*. doubles *w* and adds this to *v*. Continuing this process will eventually create a *z* which is equal to *y*, making the result of the test zero, and when this occurs the *if*. statement will no longer be executed. Line 8 will then be executed, giving as the program's result the value of *x*, since the result of a program is the result of the last sentence executed that was not in a *T* block.

The purpose of the *try*. and *catch*. blocks is to permit recovery from a failure in execution. In a statement such as

try. *B* *catch*. *B1* *end*.

if block *B* executes successfully, then *B1* is not executed. If the execution of block *B* fails, then block *B1* is executed.

* *Iverson notation* was the (non-executable) formal language introduced in [1]. Once implemented as A Programming Language, it became APL. (*Ed.*)

The behaviour of the remaining control words can be summarized as follows:

<code>break.</code>	Go to the end of a <code>while.</code> or <code>whilst.</code> control block
<code>continue.</code>	Go the top of a <code>while.</code> or <code>whilst.</code> control block
<code>goto_<name>.</code>	Go to the statement beginning with <code>label_<name>.</code>
<code>label_<name>.</code>	Target of <code>goto_<name>.</code>
<code>return.</code>	Exit the program.

Reference

- [1] Iverson, K. E., *A Programming Language*. Wiley, (December 1962). ISBN 0471430145.

5 Jacobi's Method

First published in Vector, 11, 3, (January 1995), 111-118.

Revised by Ric Sherlock, (April 2009).

Parallel Jacobi

Warning: this column contains material which may either put you to sleep or turn you against applied mathematics altogether. To take some of the sting away I have added a problem which may give you some pleasure in trying to solve. If you completely distrust your ability to read descriptions of programs, no matter how well-written, I advise you to go at once to the section headed "Problem" and avoid the preliminary exposition, or the material following, valuable as it is.

Background

Recently I had need of a program to perform eigenanalyses of square symmetric matrices, and went to *Vector* 9, 3, January 1993, which had Donald McIntyre's article "Jacobi's Method for Eigenvalues: an Illustration of J". I refer you to that article for McIntyre's lucid explanation of what the method is. In the course of transcribing his 11-line Jacobi program, along with its sixteen subprograms and its seven utility verbs, I thought I saw the possibility of speeding it up significantly by taking advantage of some of the parallelism inherent in the problem. I have communicated with McIntyre concerning this, and he tells me that he has used this method for many years, beginning with a Fortran program which he obtained from someone many years ago, transcribing it into APL and recently, as his article shows, into J.

If you look at his program, you will see that at the heart of it are the lines

```
r=. ((cos,-sin),sin,cos) (ia R)} I
Q=. q ip |:r [ R=. r ip R ip |:r
```

The first line amends an identity matrix conforming to the argument matrix by replacing two of its diagonal elements and the two corresponding off-diagonal elements with a 2-by-2 rotation matrix. The elements amended are chosen by finding the off-diagonal element of maximum magnitude, say at row-column

indices p,q , and inserting the 2-by-2 matrix items at locations (p,p) , (p,q) , (q,p) and (q,q) . This amended identity matrix r is then used with two matrix products involving R , the original argument, and Q , originally an identity matrix. Those involving R have the effect of zeroing out elements (p,q) and (q,p) of R , while leaving the eigenvalues of R unaltered. When this operation has been performed a sufficient number of times, one finds that all of the off-diagonal elements are essentially zero, and that the diagonal elements are the eigenvalues of the argument matrix . Those involving Q produce the eigenvectors of the argument matrix.

The valuable book *Matrix Computations* by Golub and Van Loan [1] describes this method (section 8.5), but because the search for (p,q) is $O(n^2)$, goes on to suggest that it might be more efficient to select p and q in a more rigid way. For the case of a 4-by-4 argument, they suggest that p and q be selected in the following order:

p	q
0	1
0	2
0	3
1	2
1	3
2	3

and go back to the beginning, repeating until a sufficiently good solution appears. Golub and Van Loan go on to point out that the rows of the (p,q) table can be arranged in a disjoint, or non-conflicting fashion:

a		b		c	
0	1	0	2	0	3
2	3	1	3	1	2

and that, in a parallel machine, separate processors can be assigned to perform the individual matrix product operations. For example, in the 4-by-4 case, two processors are needed, so that in step A one processor could do the (0,1) case and the other processor could do the (2,3) case; in step B one processor could do the (0,2) case and the other processor could do the (1,3) case; and similarly for step C. They point out that this method works only for even-order matrices, but that the odd case can be handled by bordering the argument matrix on the right and at the bottom with zeros, and then dropping these excess columns at the end. Thus the rotation matrices needed would look like this:

	step A					step B					step C			
	c01	s01	0	0		c02	0	s02	0		c03	0	0	s03
	-s01	c01	0	0		0	0	0	0		0	0	0	0
proc1	0	0	0	0		-s02	0	c02	0		0	0	0	0
	0	0	0	0		0	0	0	0		-s03	0	0	c03
	0	0	0	0		0	0	0	0		0	0	0	0
	0	0	0	0		0	c13	0	s13		0	c12	s12	0
proc2	0	0	c23	s23		0	0	0	0		0	-s12	c12	0
	0	0	-s23	c23		0	-s13	0	c13		0	0	0	0

My contribution enters here. I realized that one doesn't need a parallel machine to obtain the benefits of this parallel Jacobi method. One can combine the rotation matrices, since they are disjunct, as follows:

	step A					step B					step C			
	c01	s01	0	0		c02	0	s02	0		c03	0	0	s03
	-s01	c01	0	0		0	c13	0	s13		0	c12	s12	0
	0	0	c23	s23		-s02	0	c02	0		0	-s12	c12	0
	0	0	-s23	c23		0	-s13	0	c13		-s03	0	0	c03

This technique reduces the number of matrix products required for a matrix of size n by a factor of $n\%2$. Thus the larger the matrix, the greater the savings. A 10-by-10 problem can be reduced by a factor of 5; a 100-by-100 problem by a factor of 50, and so forth.

The Problem

Now we come to the playful part. As you can see, the row-column pairs to be included at each step must somehow be derived. In the case of a 4-by-4 matrix, we see that step A uses the pairs (0 1) and (2 3); step B uses (0 2) and (1 3); and step C uses (0 3) and (1 2). The problem is to determine a permutation z that produces the desired result. For example, for $n=4$ any of the following permutations will do:

```

0 2 3 1
0 3 1 2
1 2 0 3
1 3 2 0
2 0 1 3
2 1 3 0
3 0 2 1
3 1 0 2

```

If we set $z = 0\ 3\ 1\ 2$, we can experiment as follows:

```

] a=: (z&{)^(i.<:#z) i. #z ➡
NB. all of the possible permutations
0 1 2 3
0 3 1 2
0 2 3 1

] b=: ((2!#z),2)$,a ➡
NB. exhibit all the pairs of items
0 1
2 3
0 3
1 2
0 2
3 1

] c=: (>"1)b NB. mask shows where lead item ➡
is greater than trail
0 0 0 0 0 1

] d=: c |."_1 b ➡
NB. pairs with leading smaller item
0 1
2 3
0 3
1 2
0 2
1 3

] e=: /:~d NB. pairs in ascending order
0 1
0 2
0 3
1 2
1 3
2 3

```

Problem 1: Define a verb which takes as argument a positive even integer n and yields a permutation which, repeatedly applied to a conforming identity permutation, produces, in successive pairs of items, all possible choices of 2 items from n , with no duplications.

Problem 2: How many of the $n!$ permutations of even order n are solutions to problem 1?

Answers on page 351.

Utility verbs

Our treatment will need these verbs to be already defined. The utility verb **CP** takes a list as argument and returns the Cartesian product of the items of the list.

```
CP=: {@;"1~
```

The utility verb **IM** takes as argument a matrix and yields an identity matrix having the same number of rows.

```
IM=: [: = [: i. #
```

The utility verb **NF** takes a matrix argument and yields its Frobenius norm as result.

```
NF=: [: %: [: +/ [: , *:
```

The utility verb **clean** takes a numeric array as left argument and a positive atom as right argument. It yields a conforming array as result, wherein each element of the left argument with magnitude less than the right argument is replaced by zero.

```
clean=: [ * ] < [: | [
```

The utility verb **bz** takes a matrix argument and yields a similar matrix bordered on the right and below by a new column and row of zeros.

```
bz=: >:@$ {. ]
```

The utility verb **ub** takes a matrix argument and yields a similar matrix with the rightmost column and bottom row removed.

```
ub=: _1 _1&}
```

Principal verbs

The verbs described below were written for J8. If you are using an earlier version of J you may wish to get your system upgraded. Here are the verbs making up my solution to the parallel Jacobi problem. The two verbs **CEA** and **CEAI** produce identical results, but **CEA** is written using the rhetorical control structures which have been added to J recently (see my last article) and **CEAI** uses the algebraic control structures which have been in J from the beginning.

Each main verb `CEA` and `CEAI` (Complete EigenAnalysis) takes as argument a square symmetric matrix `A` and returns two conforming matrices, the first with the eigenvalues along the diagonal, and zeros elsewhere, and the second whose columns are the eigenvectors for the corresponding eigenvalues. They each test the parity of the number of rows of `A`. If this is even they laminate to `A` a conforming identity matrix, using the utility verb `IM`, and then apply the subverb `PJ` to this initial argument. If it is odd, the action is to border `A` on the right and the bottom with a column and row of zeros, using the utility verb `bz`, and then to apply `CEA` (or `CEAI`) to this, and at the end removing the bottom row and rightmost column of each matrix of the result with the utility verb `ub`.

```
CEA=: 3 : 'if. (2|#y) do. ub"2 CEA bz y ➡
else. PJ y,:IM y end.'
```

```
CEAI=: (PJ@(:,IM))`(ub"2@(CEAI@bz))@.(2:|#)
```

The subverb `PJ` (parallel Jacobi) takes as argument an array of two square matrices. It prepares four global variables for use by `hsjr`: a quantity `eps` as the product of a globally defined tolerance `tol` and the Frobenius norm of the first matrix, yielded by the utility verb `NF`; a quantity `s`, the number of rows in the first square matrix; a list `k`, the integers from 0 to `s-1`; and a list `p`, a permutation which will be used to alter the arrangement of the atoms of `k`, using the utility verb `mxp`. It then employs the verb `hsjr` (half of `s` Jacobi rotations) to the limit. At the limit, it yields the desired complete eigenanalysis of the original argument.

```
PJ=: 3 : 0
eps=: tol*NF {. y
s=: # {. y
k=: i. s
p=: mxp s
hsjr ^:_ y
)
```

The subverb `hsjr` (half of `s` Jacobi rotations) takes as argument an array of two square matrices. It begins by making a rotation matrix `rm`, using the verb `RM`. This rotation matrix is used with the first matrix of the argument to develop `PJ0`, the next stage of the eigenvalue matrix, one which has a smaller off-diagonal norm than the previous one, and setting to zero any of its elements which are less than or equal to the quantity `eps`, using the utility verb `clean`. Next, it uses the same rotation matrix `rm` with the last

matrix of the argument, to develop **PJ1**, the next stage in the eigenvector matrix. The two matrices are laminated to give the result array.

```

      hsjr=: 3 : 0
      rm=. (k=:p{k) RM {.y
      PJ0=. ((|:rm)+/ .*({.y)+/ .*rm) clean eps
      PJ1=. ({:y)+/ .*rm
      PJ0,:PJ1
    )

```

The subverb **RM** (rotation matrix) builds a parallel Jacobi rotation matrix.

It takes as left argument a particular permutation of the integers from 0 through **sP1**. It fashions this into a two-column table **t**, then reverses those rows of **t** in which the first atom is greater than the second atom. An array **cs** of 2-by-2 cosine-sine matrices, one for each row of **t**, is formed, using the verb **csm**. These will be used to amend a matrix of zeros in locations specified by a conforming array of 2-by-2 boxes **ix**, whose atoms are each a 2-atom list derived from the corresponding row of **t**, formed using the utility verb **CP** (Cartesian product).

For example, if a row of **t** is 2 3, the 2-by-2 boxes corresponding to it will be:

```

+---+---+
|2 2|2 3|
+---+---+
|3 2|3 3|
+---+---+

```

Finally, a matrix of zeros is formed, conforming to the right argument **y**, and the positions in this corresponding to positions given by the matrices of **ix** will be amended with the corresponding matrices of **cs**, yielding the desired parallel Jacobi rotation matrix.

```

      RM=: 4 : 0
      t=. ((-:s),2)$x
      t=. (>/"1 t)|."0 1 t
      cs=. y csm"2 1 t
      ix=. CP t
      cs ix}0:"0 y
    )

```

The subverb `esm` (cosine-sine matrix) takes as left argument a square matrix and as right argument a 2-element list of indices for that matrix, the first element giving a row number and the second element giving a column number, with the row number less than the column number. If the entry in the matrix at that row-column position is zero, the result will be a 2-by-2 identity matrix. If it is nonzero the result will be a 2-by-2 Jacobi rotation matrix, using the verb `makecs`.

```
esm=: makecs` (=@(i.@2:))@.(0:=<@){[ ]
```

The subverb `makecs` (make cosine-sine table) takes as left argument a square matrix and as right argument a 2-element list of indices for that matrix, the 1st element giving a row number and the 2nd element giving a column number, with the row number less than the column number. It yields a 2-by-2 Jacobi rotation matrix.

```
makecs=: 4 : 0
tau=. (((<2#}. y){x)-(<2#{. y){x}%+:(<y){x
t=. (*tau)%(|tau)+4 o. tau
c=. %4 o. t
s=. t*c
(c,s),:(-s),c
)
```

The subverb `mip` (make index permutation) takes a positive even integer as argument and yields a list which is a permutation of the integers from 0 through one less than the argument. The permutation is such that when applied repeatedly to a conforming list, none of the successive pairs in the lists are equal.

```
mip=: [: C. 1 0 ; <:@(,~ >:@|. )@>:@+:@i.@-:
```

Test Information

Alter the following value as desired to control accuracy and speed:

```
tol=: 1e_6 ➡
```

NB. value should be in the range 1e_2 to 1e_17

NB. Test matrices

```
] A=: 1 1 1 1,1 2 3 4,1 3 6 10,:1 4 10 20
1 1 1 1
1 2 3 4
1 3 6 10
1 4 10 20
```

```

] m=: 1.5 _1 _0.5, _1 2 _1, : _0.5 _1 1.5
1.5 _1 _0.5
_1 2 _1
_0.5 _1 1.5

```

```

] r=: 1 1 0.5, 1 1 0.25, : 0.5 0.25 2
1 1 0.5
1 1 0.25
0.5 0.25 2

```

NB. test results, using tol as specified above

```

CEA A
0.453835      0      0      0
      0 0.038016      0      0
      0      0 2.20345      0
      0      0      0 26.3047

```

```

0.787275 _0.308686 0.530366 0.0601868
_0.163234 0.723091 0.640331 0.201173
_0.532107 _0.59455 0.391833 0.458082
0.265358 0.168411 _0.393897 0.863752

```

```

CEA m
      2      0      0
      0      3      0
      0      0      0

```

```

      0.707107 _0.408248 0.57735
_9.88291e_10 0.816497 0.57735
_0.707107 _0.408248 0.57735

```

```

CEA r
_0.0166473      0      0
      0 1.48012      0
      0      0 2.53653

```

```

      0.721208 0.44428 0.531483
_0.686348 0.56211 0.461473
_0.093729 _0.697601 0.710329

```

Reference

- [1] Golub, G. H., Van Loan, C. F., *Matrix Computations*. Johns Hopkins Studies in Mathematical Sciences, 3rd Edition, (October 1996), Johns Hopkins University Press. ISBN 978-0-80185-414-9.

6 Cribbage 15s

First published in Vector, 11, 4, (April 1995), 135-138.

Revised by Ric Sherlock, (April 2009).

Sir John Suckling (1609-1642) lived to just the age of 33, as you can see. He is supposed to have taken poison to avoid a life of penury during the Civil War in which he was a cavalier, not a roundhead. He was also a poet with an entrancing lyrical gift. He wrote:

*Her lips were red, and one was thin
Compared to what was next her chin
(Some bee had stung it newly);*

More to the point, John Aubrey, in his *Brief Lives*, tells us that Suckling also invented the game of cribbage, developing it from an earlier game, called noddly. I know nothing of noddly, but I was taught cribbage while I was in the army (1944-6) by a fellow soldier named Goman, who came from Duluth, Minnesota (named for a French explorer named Du Luth). According to Goman, Duluth was the cribbage capital of the world. The reason for this was that Duluth is at the very western end of the westernmost of the Great Lakes, Lake Superior, and in the winter, when lake traffic has stopped, there is nothing for people to do but go to their neighbourhood pub and play cribbage while drinking beer. They also had the world's championship cribbage match, to which all of the best cribbage players in the world came. Most of them didn't have to come far, since many were from Duluth. This may have changed over the last 50 years.

Just recently, when our nightly double solitaire game palled, my wife and I decided to play cribbage instead. I got out the cribbage board I had made, close to fifty years ago, and, with the aid of some wooden matches to use as pegs, we began to play. My wife hadn't played the game as much as I had, and it was clear that she was missing some opportunities to score because she didn't see some of the card combinations that added to 15. Such scores are a key part of the game.

To help her out, I decided to tabulate all the combinations of cards of length two through five that added to 15. This was easy enough for length two and three, but it became a little tedious and uncert-

ain for greater lengths. I decided to take a few minutes longer and get it right by doing it with J.

Because I had been spending most of my time recently on studies involving Jane Austen, I hadn't been doing much programming, and I was a bit rusty. My first attempt used a technique which generated all of the representations of 2, 3, 4, and 5-digit decimal numbers, removed those which had more than four of any digit, sorted the rows that were left (so that 4 1 became 1 4, for example), removed duplicates, and removed rows that didn't add to 15. This strategy foundered because I ran out of space.

I took a few more minutes to think of a more space-efficient strategy, and decided to use a program called `part` that had been communicated to me several years ago by Roger Hui, which allowed one to generate all the length k partitions of a positive integer n . For example, *

```

2 part 7
1 6
2 5
3 4
3 part 6
1 1 4
1 2 3
2 2 2

```

Having this made it almost too easy.

First I had to generate the partitions of 15 having length 2, 3, 4, and 5.

Next, since in cribbage the card values are from 1 (for ace) to 10 (for 10, jack, queen, or king), I had to remove rows having elements greater than 10.

Next, for the 5-partitions, I had to remove the last row, since this would consist of five threes (how did I know this?), and thus not be valid (there are only four threes in a deck of cards).

Last, I wanted to box the results so they could be joined together.

To get rid of rows containing values beyond 10 I wrote:

```
i=: *./"1@(10&>:) # ]
```

* This example needs the verb: `part`. See page 49.

This takes as argument a table of numbers and produces a Boolean list by taking the and ($*$.) over ($/$) the rows ("1) of the conforming table having a 1 for values for which 10 was greater than or equal ($>:$), and using this to copy ($\#$) only those rows from the argument ($[]$).

To get rid of the last row if the rows had length 5, I wrote:

```
m=: }: ^: (5 = #@{.)
```

This curtails ($\}$;) the table if ($\wedge:$) 5 is equal to ($=$) the length ($\#$) of its first ($\{.$) row.

Boxing is primitive, so the entire result could then be obtained by writing:

```
] p=: 2 3 4 5 <@m@i@part"(0) 15
+-----+-----+-----+
| 5 10|1 4 10|1 1 3 10|1 1 1 2 10|
| 6 9 |1 5 9 |1 1 4 9 |1 1 1 3 9 |
| 7 8 |1 6 8 |1 1 5 8 |1 1 1 4 8 |
|      |1 7 7 |1 1 6 7 |1 1 1 5 7 |
|      |2 3 10|1 2 2 10|1 1 1 6 6 |
|      |2 4 9 |1 2 3 9 |1 1 2 2 9 |
|      |2 5 8 |1 2 4 8 |1 1 2 3 8 |
|      |2 6 7 |1 2 5 7 |1 1 2 4 7 |
|      |3 3 9 |1 2 6 6 |1 1 2 5 6 |
|      |3 4 8 |1 3 3 8 |1 1 3 3 7 |
|      |3 5 7 |1 3 4 7 |1 1 3 4 6 |
|      |3 6 6 |1 3 5 6 |1 1 3 5 5 |
|      |4 4 7 |1 4 4 6 |1 1 4 4 5 |
|      |4 5 6 |1 4 5 5 |1 2 2 2 8 |
|      |5 5 5 |2 2 2 9 |1 2 2 3 7 |
|      |      |2 2 3 8 |1 2 2 4 6 |
|      |      |2 2 4 7 |1 2 2 5 5 |
|      |      |2 2 5 6 |1 2 3 3 6 |
|      |      |2 3 3 7 |1 2 3 4 5 |
|      |      |2 3 4 6 |1 2 4 4 4 |
|      |      |2 3 5 5 |1 3 3 3 5 |
|      |      |2 4 4 5 |1 3 3 4 4 |
|      |      |3 3 3 6 |2 2 2 2 7 |
|      |      |3 3 4 5 |2 2 2 3 6 |
|      |      |3 4 4 4 |2 2 2 4 5 |
|      |      |      |2 2 3 3 5 |
|      |      |      |2 2 3 4 4 |
|      |      |      |2 3 3 3 4 |
+-----+-----+-----+
```

To make this a bit more useful, I copied it to my text editor (MacWrite Pro) and changed 10 to T and 1 to A, adjusted widths a bit, and added a few header and footer lines, giving:

Ways of counting fifteen with 2 3 4 5 cards in cribbage

+-----+-----+-----+-----+-----+														
5	T	A	4	T	A	A	3	T	A	A	A	2	T	
6	9	A	5	9	A	A	4	9	A	A	A	3	9	
7	8	A	6	8	A	A	5	8	A	A	A	4	8	
		A	7	7	A	A	6	7	A	A	A	5	7	
		2	3	T	A	2	2	T	A	A	A	6	6	
		2	4	9	A	2	3	9	A	A	2	2	9	
		2	5	8	A	2	4	8	A	A	2	3	8	
		2	6	7	A	2	5	7	A	A	2	4	7	
		3	3	9	A	2	6	6	A	A	2	5	6	
		3	4	8	A	3	3	8	A	A	3	3	7	
		3	5	7	A	3	4	7	A	A	3	4	6	
		3	6	6	A	3	5	6	A	A	3	5	5	
		4	4	7	A	4	4	6	A	A	4	4	5	
		4	5	6	A	4	5	5	A	2	2	2	8	
		5	5	5	2	2	2	9	A	2	2	3	7	
					2	2	3	8	A	2	2	4	6	
					2	2	4	7	A	2	2	5	5	
					2	2	5	6	A	2	3	3	6	
					2	3	3	7	A	2	3	4	5	
					2	3	4	6	A	2	4	4	4	
					2	3	5	5	A	3	3	3	5	
					2	4	4	5	A	3	3	4	4	
					3	3	3	6	2	2	2	2	7	
					3	3	4	5	2	2	2	3	6	
					3	4	4	4	2	2	2	4	5	
									2	2	3	3	5	
									2	2	3	4	4	
									2	3	3	3	4	
+-----+-----+-----+-----+-----+														

T = 10, J, Q, or K

The key to this is Roger Hui's partition function, `part`:

```
part=: 4 : 0
  if. (1<x)*:x<y do. ➡
    (x,~(*y)*x<:y)$1>.y*1=x return. end.
  p=. x part&<: y
  s=. >:i.<.y%x
  t=. ({."1 p)<.-.-/ @_2&{.)"1 p
  b=. s<:/t
  i=. (+/"1 b)#s
  j=. (,b)#(*/$b)$i.{:$b
  m=. i,.j{p
  (}:{."1 ,. 1: + {:"1 - {."1) m
)

partcheck=: 4 : 0
  t=. x part y
  assert. x={:$t
  assert. (i.#t) -: /: t
  assert. (i.x) -: "1 /: "1 t
  assert. (0=x)+.y=+/"1 t
)
```

All in all, I spent a happy half-hour at play with J, and my wife now beats me pretty regularly at cribbage.

7 Representing a Permutation

First published in Vector, 12, 1, (July 1995), 125-128.

Revised by Roger Hui, (April 2009).

This column explores some ways of changing among different ways of representing a permutation.

Representations of a permutation:

standard	reduced	atomic
0 1 2	0 0 0	0
0 2 1	0 1 0	1
1 0 2	1 0 0	2
1 2 0	1 1 0	3
2 0 1	2 0 0	4
2 1 0	2 1 0	5

The tables above give three different forms of length-3 permutations. It is useful to be able to go between the standard and the atomic forms, and this conversion is facilitated by the reduced form. We develop the following verbs:

ra reduced from atomic
ar atomic from reduced
sr standard from reduced
rs reduced from standard.

With these we can convert from each of the forms to any other. A *factorial digits* number base is used to convert between the atomic and reduced forms. The verb **fdb** gives the factorial digits base for permutations of the order of its argument.

fdb=: >:@i.@-

For example,

fdb 3
3 2 1

With this base we can convert an atomic to a reduced form and vice-versa:

(fdb 3)#: 4
2 0 0
(fdb 3)#. 2 0 0
4

So the two verbs `ra` and `ar` are easily defined:

```
ra=: ([: fdb [) #: ]
ar=: ([: fdb #) #. ]
```

For example:

```
3 ra 4
2 0 0
ar 2 0 0
4
```

To convert from a reduced to a standard form is somewhat more difficult. The trick is to begin at the right and add 1 to each atom which is equal to or greater than the atom at the left. This ensures that all atoms are kept distinct, and that at each step we have a permutation. For example, suppose we take the length 9 reduced form of the atomic form 288918:

```
] r=: (fdb 9) #: 288918
7 1 2 1 3 1 0 0 0
```

and then work from the right to develop the standard form:

```
0
0 1
0 1 2
1 0 2 3
3 1 0 2 4
1 4 2 0 3 5
2 1 5 3 0 4 6
1 3 2 6 4 0 5 7
7 1 3 2 6 4 0 5 8
```

and the last result is the desired standard form. We could define a function that follows this procedure, but we can do better than this, employing the identity I discovered in 1968. For integer k and permutation p ,

```
k , p + p >: k ↔ /: /: k , p
```

and arrive finally at the desired verb:

```
sr=: /: @ /: @ , /
] s=: sr r
7 1 3 2 6 4 0 5 8
```

The last verb we need, to translate from standard to reduced form, is arrived at by noting that, if r is the reduced form of a standard form s , then $i\{r$ is obtained from $i\{s$ by taking a count of how many atoms to the right of $i\{s$ are less than $i\{s$.

For example:

```

7 1 3 2 6 4 0 5 8
7 > x x x x x x x x = 7
1 >           x       = 1
3 > x         x       = 2
2 >           x       = 1
6 > x x x       = 3
4 > x           = 1
0 >           = 0
5 >           = 0
8 >           = 0

```

J provides two related adverbs, *prefix* and *suffix*. The first, *prefix*, is applied to longer and longer *prefixes*, whereas *suffix* is applied to shorter and shorter *suffixes*. Thus the *rs* verb we need can be defined as:

```

rs=: ([: +/ }. < {.)\.
rs s
7 1 2 1 3 1 0 0 0

```

With these four verbs, *ar*, *ra*, *rs*, *sr*, it is possible to obtain any of the three forms from any other. We don't need a verb to go directly from standard to atomic or vice-versa. However, J provides this as a primitive verb, denoted by *A.* and called 'Atomic Index' for its monad and 'Atomic Permute' for its dyad. For example,

```

A. s
288918
288918 A. i. 9
7 1 3 2 6 4 0 5 8

```

So with *A.* a primitive, the four verbs we laboured over above are more interesting for pedagogical than for practical reasons. Atomic permute doesn't care what its right argument is; it will permute any object of sufficient length:

```

288918 A. 'netrilacy'
certainly

```

A useful verb to generate a table of all permutations of a given length is easy to write:

```

apn=: i.@! A. i.

```

```
      apn 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

You would need a computer with a rather large amount of main store to generate `apn 12` ...about 46e9 bytes (8 bytes per element, 12 elements per row, 479,001,600 rows). Of course, the computer would also have to be able to address that large a store, too. Judging from the current state of affairs, it may well be almost the year 2000* before we routinely have these capabilities on our desktops.

* Eugene's prediction made in 1995 (>46 GB) was somewhat optimistic. As of October 2009 the Apple iMac™ comes with the option of 4 GB of installed memory. (*Ed.*)

8 The Bauer-Mengelberg Problem

First published in Vector, 12, 2, (October 1995), 115-122.

Revised by Fraser Jackson, (April 2009).

This paper discusses a combinatorial problem arising in the field of music, and shows the importance of the A. primitive discussed in my last column.

The problem was told to me many years ago by Ken Iverson, who had heard it from Adin Falkoff, who in turn had heard it from Stephen Bauer-Mengelberg, a conductor / programmer who was a colleague of Ken and Adin's at IBM's Systems Research Institute at UN Plaza in New York City in the early 1960s.

Picturesque but irrelevant detail: Adin tells of asking Bauer-Mengelberg how one of the pieces he conducted at a concert the night before had gone. The answer was "The first movement went only so-so, but with the second movement I floated off the podium."

The problem deals with the twelve-tone music associated with the composer Arnold Schoenberg. I am not a musician, so I shall only briefly describe it musically, and then convert it into a problem in combinatorial mathematics.

The problem is to describe all the ways in which the twelve semi-tones of the octave can be written so that each is used exactly once, and so that each interval possible within the octave occurs exactly once. Illing [1] gives an example of such a piece:



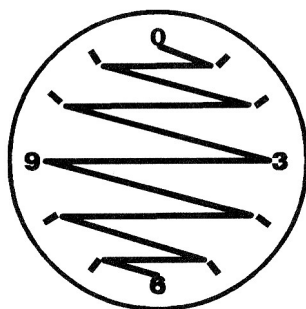
The notes begin with A natural, and then alternately rise and fall, in the sequence B flat, G sharp, B natural, G natural, C natural, F sharp, C sharp, F natural, D natural, E natural, and D sharp. I find it convenient to number these notes according to their signed distances from A natural, which I number as 0. The twelve notes are then seen as

0 1 _1 2 _2 3 _3 4 _4 5 _5 6

And it simplifies things if we take these mod 12, giving

0 1 11 2 10 3 9 4 8 5 7 6 [A]

I have found it helpful visually to write these numbers as the hours on a clock face (using 0 in place of 12), and to connect the hours by lines in the order given, that is, draw a line connecting 0 to 1, 1 to 11, 11 to 2, and so on, ending with a line drawn from 7 to 6.



This clock figure makes more apparent various symmetries that reduce the number of permutations that need to be considered.

If we take the first difference of [A], we get the following:

1 10 _9 8 _7 6 _5 4 _3 2 _1

and if we take this mod 12, we get

1 10 3 8 5 6 7 4 9 2 11 [B]

and it is easy to see that the list [A] is a 0-origin permutation having a first difference, mod 12 [B] which is a 1-origin permutation. Thus we have transformed the musical problem, having to do with twelve-tone rows, into the combinatorial problem of determining all the permutations of $i \cdot 12$ having a first difference which is a permutation of $>: i \cdot 11$. That is, we want to know how many such permutations there are, and what they are. To make it easier to discuss "a permutation having a first difference mod permutation length also a permutation". I'll call such an object a 'dil' (from *Distinct Interval List*).

There are 479,001,600 permutations of $i \cdot 12$, so it is a large problem to sift through these permutations looking for dils. For example, to load the table of all permutations of order 12 would take

x: $4 \times 12 \times 12$, or 22,992,076,800 bytes. I believe that this would be impossible to load in real memory on the largest contemporary machine. This paper explores ways to cut it down to a more manageable size.

I heard the problem in the early 1960s when Iverson notation was available only on the printed page, and worked at it by hand for several months without making much progress. Recently I decided to tackle it once more, beginning by studying the permutations of smaller order. I found that dils occur only among even length permutations. The order two permutations are easy: both are dils: 0 1 and 1 0, having an interval of 1. These can be done mentally, but it quickly becomes necessary to develop programming tools to aid in the exploration:

```

pt=: i.@! A. i.      NB. permutation table
mfd=: # | }. - }:    NB. modular first difference
mn=: -: ~.          NB. distinct items?
dil=: mn@mfd"1       NB. a dil?
dils=: dil # ]       NB. all dils
pt 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
mfd 0 1 5 2 4 3
1 4 3 2 5
mn mfd 0 1 5 2 4 3
1
dil 0 1 5 2 4 3
1

```

Studying the dils of order 4 give us some insight into the problem:

```

dils pt 4      NB. dils of length 4
0 1 3 2
0 3 1 2
1 0 2 3
1 2 0 3
2 1 3 0
2 3 1 0
3 0 2 1
3 2 0 1

```

Some symmetries are present that will let us cut the problem down in size. Only permutations beginning with 0 need be considered, since the others can be obtained by clock face rotations:

```
ro=: #@] | + NB. rotate y by x
1 ro 0 1 3 2
1 2 0 3
2 ro 0 1 3 2
2 3 1 0
```

and similarly for the others. I call the dils beginning with zeros 'basic dils', since all the others can be obtained from them by rotation, or, in musical terms, by transposing. By looking for dils only among permutations beginning with 0, our order x: !12 problem has been reduced reduced to an order x: !11 problem, or 39,916,800. Here are the basic dils of orders 4, 6, and 8:

```
a4=: dils(i.!3)A. i.4
a6=: dils(i.!5)A. i.6
a8=: dils(i.!7)A. i.8

a4          a6          a8
0 1 3 2      0 1 5 2 4 3    0 1 3 6 2 7 5 4
0 3 1 2      0 2 1 4 5 3    0 1 6 5 3 7 2 4
              0 4 5 2 1 3    0 1 7 2 6 3 5 4
              0 5 1 4 2 3    0 1 7 3 6 5 2 4
                      0 2 1 5 3 6 7 4
                      0 2 3 6 5 1 7 4
                      0 2 5 1 7 6 3 4
                      0 2 7 6 1 5 3 4
                      0 3 1 2 6 5 7 4
                      0 3 2 7 1 5 6 4
                      0 3 5 1 2 7 6 4
                      0 3 5 6 2 1 7 4
                      0 5 3 2 6 7 1 4
                      0 5 3 7 6 1 2 4
                      0 5 6 1 7 3 2 4
                      0 5 7 6 2 3 1 4
                      0 6 1 2 7 3 5 4
                      0 6 3 7 1 2 5 4
                      0 6 5 2 3 7 1 4
                      0 6 7 3 5 2 1 4
                      0 7 1 5 2 3 6 4
                      0 7 1 6 2 5 3 4
                      0 7 2 3 5 1 6 4
                      0 7 5 2 6 1 3 4
```

Further efficiencies are possible. Notice that all of these dils not only begin with the constant 0, but end with a constant that is half of the order: 2, 3, and 4 for orders 4, 6, and 8, respectively. This means that in searching for dils we only have to look at those permutations beginning with 0 and ending with a constant, with some permutation between them.

The desired inner permutation is given by:

```

si=: i. -. 0: , -: ➡
NB. integers thro n-1, less 0 and -:n
si 2

    si 4
1 3

    si 6
1 2 4 5

    si 8
1 2 3 5 6 7

    si 10
1 2 3 4 6 7 8 9

    si 12
1 2 3 4 5 7 8 9 10 11

```

By having to consider only inner permutations of order $n-2$, we have now reduced our problem to one of order $!10$, or 3,628,800. Furthermore, looking carefully again at the tables **a4**, **a6**, and **a8** above, we see that only the first half of the basic dils need to be tested, since the rest can be found by clock face reflections in the y -axis. That is, any one of the rows in the lower half of any of these tables is obtainable from one of the rows in the upper half. The verb **ry** reflects a dil about the y -axis:

```

ry=: # | # - ]
ry 0 1 3 2
0 3 1 2

```

This means that to find the dils of order 12, we have to test only $x: -: !10$, or 1,814,400 permutations. This is a reduction from $x: !12$ by a factor of 264.

Since we can always retrieve a dil if we know its atomic number and its length, we don't need to exhibit the complete row. It suffices to obtain only its atomic number. For example, the dils of order 4 can be obtained using only 8 integers, rather than the 32

required by the display of the four atoms of each permutation form of the dil.

We can define a verb `dan` to give us the dils in atomic number form:

```
dan=: (dil # A.) NB. dil atomic number
dan pt 4         NB. atomic numbers of dils →
of order 4
1 4 6 8 15 17 19 22
```

There are two additional clock face reflective symmetries in these dils. In addition to the y-axis symmetry mentioned above, there are reflections possible in the x-axis, and in both the x and y axes. For example, the dil:

```
r=: 0 1 3 2 7 10 8 4 11 5 9 6
```

can be reflected in the x-axis by:

```
rx=: [: |. # | -:@# - ]
rx r
0 9 1 7 2 10 8 11 4 3 5 6
```

and in the x-y axes by:

```
rxxy=: [: |. # | -:@# + ]
rxxy r
0 3 11 5 10 2 4 1 8 9 7 6
```

I haven't found a way to use these further symmetries to reduce the work necessary to solve the dil problem. The program I use to find the primitive dils of order n is:

```
pdon=: 3 : 0
NB. argument is 4-item list, e.g. pdon 12 5040 0 1814400
'n i b m'=. y
NB. n is length of permutation
NB. i is size of batch (depends on memory size and n)
NB. b is base index (usually 0 initially)
NB. m is maximum item number (usually -:!n-2)
NB. z is result, list of indices of primitive dils of order n
z=. ''
s=. si n          NB. for example, si 8 is 1 2 3 5 6 7
h=. -:n          NB. for n=8, h is 4
while. b<m do.
  t=. 0.,((b+i.i)A. s),.h NB. provide another batch
  z=. z,dan t          NB. append primitive dil atomic #s to z
  b=. b+i              NB. step base by batch size
  (": b,6!:1 ' ') 1!:2 (2)
end.
z
)
```

The line assigning `t` shows the utility of being able to specify the right argument to the `A.` primitive. On my computer, it took about 10 minutes to compute the dils of order 10. I don't know how long it took to do those of order 12. I started it going just before I went to bed, and it was ready in the morning.

For the record, the number of dils of orders 2 through 12 are:

order	primitive dils	basic dils	all dils
2	1	1	2
4	1	2	8
6	2	4	24
8	12	24	192
10	144	288	2880
12	1928	3856	46272

Here are a few nicely symmetrical dils of order 12:

```

pty12s=: 646517 3154657 4275293 5762095 7289175 9306655
pty12s=. pty12s, 11633649 12187013 13754599 14826363 ➡
16823821
pty12s A. i. 12
0 1 3 10 2 5 11 8 4 9 7 6
0 1 10 8 3 11 5 9 2 4 7 6
0 2 3 10 1 5 11 7 4 9 8 6
0 2 7 10 11 3 9 5 4 1 8 6
0 3 1 2 10 5 11 4 8 7 9 6
0 3 7 8 10 5 11 4 2 1 9 6
0 4 3 1 8 5 11 2 7 9 10 6
0 4 5 8 3 1 7 9 2 11 10 6
0 4 9 11 2 1 7 8 5 3 10 6
0 5 1 10 8 9 3 2 4 7 11 6
0 5 8 4 3 1 7 9 10 2 11 6

```

If you're a musician you might try playing these. They also make interesting clock face patterns. If you have a current version of J on your computer you can see them drawn using the graphics facilities available. You can load the required functions and create two useful functions by entering the following:

```

require 'graph'
gwin=: 3 : 0 NB. right argument is window name
0 0 0 gwin y NB. default pen color is black
:
gdopen y
gdpen 2 NB. default pen size, reset with gdpen
gdpen color x NB. left arg sets initial pen color
gshow''
)

```

```

glines=: 3 : 0
0 0 0 gdlines y
:
x gdlines ,y NB. left argument resets pen color
gdshow ''
)

```

Additional information about using the J graphics facilities are described in *Fractals Visualization and J* by Clifford Reiter, [2].

Here is the beginning of a sample session of visualizing dils on a clock face to help you get started.*

The default graphics window has range ($_{-1}, 1$) in both x and y

```

] r12=: 12 %: _1 ➡
NB. 12th root of negative 1.
0.965926j0.258819
] all=: r12^2*i.12 ➡
NB. first 12 powers of this root
1 0.866025j0.5 0.5j0.866025 6.12574e_17j1 ➡
_0.5j0.866025 _0.866025j0.5
_1j1.22515e_16 _0.866025j_0.5 _0.5j_0.866025 ➡
_1.83772e_16j_1 0.5j_0.866025
0.866025j_0.5
] coords=: +.all ➡
NB. split numbers into real & imaginary parts
      1      0
0.866025      0.5
      0.5      0.866025
6.12574e_17      1
_0.5      0.866025
_0.866025      0.5
_1 1.22515e_16
_0.866025      _0.5
_0.5      _0.866025
_1.83772e_16      _1
      0.5      _0.866025
0.866025      _0.5

```

* The graphics tools in J have significantly changed since the article was written. The above J code for the two graphics functions provides a simple way of achieving a similar result to Eugene's original code. (Ed.)

With these defined you can create a graphics window with

```
gwin 'clock'
0 glines coords,{.coords NB. completes polygon
```

and display the lines for a given permutation on the clock face with

```
perm=: 12 | 3+ry 0 1 11 2 10 3 9 4 8 5 7 6
p=: perm{coords
RED glines p
```

References

- [1] Illing, Robert, *A Dictionary of Music*. Penguin Books, (1950), 297 (fig. f).
- [2] Reiter, Clifford, *Fractals, Visualization and J*. Lulu.com, Third Edition, (2007), ISBN 978-1-4303-1980-1.
<http://www.lulu.com/content/635966>
<http://www.lulu.com/content/635938> (coil binding.)

9 Heron's Rule and Integer-Area Triangles

First published in Vector, 12, 3, (January 1996), 133-142.

Revised by Kip Murray, (April 2009).

I. Preliminaries

This note makes use of several less-well-known parts of J: the fix (f.) and Taylor series coefficient (t.) adverbs and the polynomial rootfinder verb (p.).

To make the following accessible to all readers, the following verbs are defined:

C=: @ NB. compose	f C g x <-> f g x
D=: % NB. divide	18 D 3 <-> 6
H=: -: NB. halve	H 12 <-> 6
I=:] NB. identity	I 6 <-> 6
P=: */ NB. product	P 1 2 3 <-> 6
R=: %: NB. square root	R 36 <-> 6
S=: +/ NB. sum	S 1 2 3 <-> 6

The following convention applies to verbs f, g, and h:

(f g h) y <-> (f y) g (h y) NB. (%: , *:) 16 <-> 4 256

II. Heron's Rule

Heron's rule for the area A of a triangle with sides a , b , and c is usually written in two steps. First the semi-perimeter s is computed:

s=: (a + b + c) D 2

For example:

```
(13 + 14 + 15) D 2
21
      42      D 2
21
```

And then the following expression for the area is computed:

A=: R (s * (s - a) * (s - b) * (s - c))

Continuing the example:

```
R (21 * (21 - 13) * (21 - 14) * (21 - 15))
84
R (21 *      8      *      7      *      6      )
84
R 7056
84
```

Heron's is a scalar-oriented formula, with the lengths of the three sides and the semi-perimeter playing separate roles in the formulation. We make a first approach to an array formulation by considering the triangle to be defined by a 3-item list of side lengths. We then determine the semi-perimeter by a verb `SP`:

```
SP=: H C S
```

The next step is to replace the three explicit subtractions by appending a zero to the list and subtracting the resulting four values from the semi-perimeter, then taking the product over this result, and finally the square root of the product.

```
Heron=: R C (P C (SP - (0 , I)))
```

This is a slightly more efficient form than APL expression 318 in the FinnAPL Idiom Library [1].¹

```
Heron 13 14 15
84
```

Fixing the definition of `Heron`, and giving this fixed version the name `Hrn`, by using the fix adverb (`f.`) yields a form in which the names of defined items are replaced by their values. Doing this insures that changes in the items defined do not alter the definition of the item in which they are used. As a side effect, a fixed verb is generally faster than an unfixed equivalent.

```
Hrn=: Heron f.
Hrn
%:@(*/@(-:@(+/-) - 0 , ]))2
Hrn 13 14 15
84
```

III. Integer Heron

In preparing examples for Heron's formula, I thought it would make the examples clearer if I could find triangles having integer sides that also had integer areas. I explored consecutive triplets of integers among the first 200 sets of triplets.

¹ which is: $(1 \text{ } 1 \text{ } \< \backslash X^{\wedge} . = \& X) \neq X$

² There are several different ways of displaying the definition of a verb. The one shown is not the default (which is "boxed"). To see verbs displayed in the mode shown ("linear"), first execute the expression: $(9! : 3) \text{ } 5$

The first step was to build the table of triplets:

```
T=: 3 ]\ i. 202
```

The table has 200 rows:

```
$ T
200 3
```

Its first and last four rows are:

```
  4  _4 {."0 _ T
0   1   2
1   2   3
2   3   4
3   4   5
196 197 198
197 198 199
198 199 200
199 200 201
```

Applying `Hrn` to the rows of this table gives a list of areas.

```
Areas=: Hrn"1 T
```

We determine which of the areas are integral (those equal to their own floor):

```
Mask=: (= <.) Areas
```

And use the mask so found to give us the winning rows of `T`:

```
] Winners=: Mask # T
1   2   3
3   4   5
13  14  15
51  52  53
193 194 195
```

The areas corresponding to these are:

```
Hrn"1 Winners
0 6 84 1170 16296
```

The triangle 1 2 3 is degenerate (ugh!).

IV. A Recursive Formula

I looked at `Winners` for some clue as to how the series could be prolonged, but without success. Then I thought of N. J. A. Sloane's book *A Handbook of Integer Sequences* [2]. I looked in it for the series 1 3 13 51 193 without avail. Then, knowing that each series in the

book began with 1, which was sometimes prefixed to a series which began naturally with some other integer, I looked for 1 2 4 14 52 194 and 1 3 5 15 53 195, but again to no avail. Finally, I divided the even column 2 4 14 52 194 by 2, looked for 1 2 7 26 97, and this time struck pay dirt. It was Sloane's sequence 700.

Sloane's entry for series 700 not only gave a number of additional values but, more importantly, it gave a doubly recursive formula for finding the values, in common mathematical notation:

$$A(n) = 4A(n-1) - A(n-2).$$

So now I was able to extend the series as far as I wanted. I wrote a J version of A:

```
A=: ((4*A@<:)-(A@<:@<:))`>:@.(<2)
```

which, as you can see, is doubly recursive in A. It tests whether the argument is less than 2 (<2), giving one plus the argument (>:) as result in these cases, and otherwise yields the difference between A of $n-1$ (<:) and A of $n-2$ (<:<:).

I calculated additional results of A, for arguments 6 through 14, derived triplets from the results, and applied Hrn to the triplets, and in each case found an integer area. But was I satisfied? No.

V. Generating Functions

O them doddhunters and allanights, aabs and baas for agnomes, yeess and zeess for incognits, bate him up jerrybly!

James Joyce, Finnegans Wake, p. 283.

The reason the story carries on is that I was unhappy with the long execution times required by the deeply rooted calling trees of the double recursion. I had to terminate the execution of A 30 after five hours with no result. My mind turned to the subject of generating functions, something I had often heard about and often, with little or no success, had tried to master. I was stimulated to do this because of three books. These were K. E. Iverson's new book *Concrete Math Companion*; the Ronald Graham, Donald E. Knuth, and Oren Patashnik book *Concrete Mathematics*, which I shall refer to as GKP; and most of all, the H. S. Hall and S. R. Knight book *Higher Algebra*, first edition 1887, and usually referred to as Hall & Knight, worthy successor to Todhunter's *Algebra for Schools and Colleges*.

Both of these books are celebrated by James Joyce in the mathematics chapter of his *Finnegans Wake*. Iverson's new book and GKP focus sharply on generating functions. From GKP I learned a four-step process that promised to allow me to have my will with arbitrary generating functions. I plodded through their examples, and tried to duplicate their results on my problem. No luck. I turned to Iverson's book, and found out one important thing that GKP had neglected to tell me, that is, that the key to generating functions was the ability to generate the coefficients of Taylor series, something that J is well suited for, since it contains a primitive (`t.`) to do just that. However, that is about all I was able to learn there. Lastly, I got out my rusty red copy of Hall & Knight, and it came through. The examples they gave were of the same kind as mine, that is, they dealt primarily with doubly recursive functions, where the n th term is some linear combination of the two preceding terms. Their explanations were carefully laid out in great detail.

Here is how they go about it.

Given a sequence with a sufficient number of terms, it is possible to describe how to extend the sequence arbitrarily. The first thing to do is to get rid of the notion that we are dealing with a mere list of numbers. Instead we think of the list as being the coefficients in a polynomial with a never-ending set of terms, that is, an infinite series. Thus the list:

1 2 7 26 97 ...

in fact defines the first several coefficients of the infinite series:

$$(1 \cdot y^0) + (2 \cdot y^1) + (7 \cdot y^2) + (26 \cdot y^3) + (97 \cdot y^4) + \dots$$

This is where Hall and Knight lost me. They say, from out of the blue, that each term after the second is equal to the sum of the two preceding terms multiplied respectively by the constants `_1` and `4`. Thus:

$$7 = (_1 \cdot 1) + (4 \cdot 2)$$

or

$$7 = _1 \ 4 \ +/ \ . \ * \ 1 \ 2$$

This implies that if we take any three consecutive terms r , s , t , they are related by:

$$t = (_1 * r) + (4 * s)$$

which can be rewritten as:

$$0 = (1 * r) + (_4 * s) + (1 * t)$$

In this equation the coefficients

$$1 \quad _4 \quad 1$$

of r , s , and t form the scale of relation of the infinite series. They are the coefficients of a quadratic polynomial written in ascending order:

$$(1*y^0) + (_4*y^1) + (1*y^2)$$

Now this is all stated baldly in Hall & Knight, and I was thoroughly lost. How does one find the scale of relation, and what was the point of it? Luckily, the authors soon give the game away, noting that if a sufficient number of the terms of a series be given, the scale of relation may be found, and proceed to show how to do just that.

Suppose the first four terms of the series are, in order, a , b , c , and d . Assume then that the general term is arrived at by multiplying the two preceding terms by p and q , and adding. We are able to write the following pair of equations:

$$\begin{aligned} c &= (p*a) + (q*b) \\ d &= (p*b) + (q*c) \end{aligned}$$

and then it is a simple matter to solve this linear system for p and q by writing

$$'p \ q' =: (c,d) \% . (a,b), : (b,c)$$

For example, if a , b , c , and d are 1, 2, 7, and 26 we write

$$\begin{array}{l}] \ 'p \ q' =: 7 \ 26 \% . \ 1 \ 2, : 2 \ 7 \\ _1 \ 4 \end{array}$$

and we can form the scale of relation by appending a 1 to the negative of these:

$$\begin{array}{l}] \ s =: 1 \ , \sim - \ 7 \ 26 \% . \ 1 \ 2 \ , : 2 \ 7 \\ _1 \ _4 \ 1 \end{array}$$

A pretty way to write this in J is to form a table t as follows:

$$\begin{array}{l}] \ t =: 2 \] \ y =: 1 \ 2 \ 7 \ 26 \\ _1 \ 2 \\ _2 \ 7 \\ _7 \ 26 \end{array}$$

and then we can write

$$\begin{array}{l} (\{ : \% . \} :) t \\ _1 \ 4 \end{array}$$

So that we can form the scale of relation using the function `scr`:

```
scr=: 1 ,~ [: - [: ({: % . }:) 2 ]\ ]
```

And use it to get our scale of relation:

```
scr y
1 _4 1
```

What can we do with a scale of relation? Suppose we take the vector product of the scale of relation and any three successive terms of the infinite series, say 7 26 97

```
] a=: 1 _4 1 * 7 26 97
7 _104 97
+/_ a
0
```

This sum will always be zero as a consequence of the way the infinite series and the scale of relation are interrelated. Consequently, if we do the polynomial multiplication of the scale of relation with the infinite series beginning with 1 2 7 26, we find that all the terms after the first two are zero:

1	2	7	26	97	362	...
1	_4	1				
1	2	7	26	97	362	...
	_4	_8	_28	_104	_388	...
		1	2	7	26	...
1	_2	0	0	0	0	...

Since all the terms of this product after the first two are zero, and since we can ignore trailing zeros in a list of polynomial coefficients, we find that the infinite product of the scale of relation and the infinite series reduces to the linear polynomial:

$$(1 * y^0) + (_2 * y^1)$$

In practice it is difficult to represent or work with infinite series, so we enable the process by using only the first two terms of the series. We can then do the polynomial multiplication of these two terms with the scale of relation, and take only the first two terms of the resulting product. Ordinary polynomial multiplication is given by:

```
pm=: +//. @ (*/)
```

and our special infinite series multiplication by this scale of relation polynomial is given by:

$$\begin{array}{r} \text{spm} = : 2 \{ . \text{pm} \\ 1 _4 1 \text{spm} 1 2 7 26 \\ 1 _2 \end{array}$$

The next thought to convey to you is the most important one in the whole paper, so **PAY ATTENTION!**

Let me write the situation schematically:

$$\frac{\text{Infinite series} \times \text{Scale of Relation}}{\text{Scale of Relation}} \leftrightarrow \text{Infinite series}$$

That is, if I multiply and divide the infinite series by the scale of relation, I end up with the infinite series. But I know the numerator is simply a linear polynomial. So I can substitute the linear polynomial for the numerator and write:

$$\frac{\text{Linear Polynomial}}{\text{Scale of Relation}} \leftrightarrow \text{Infinite series}$$

This suggests that an infinite series of the kind we are describing can be represented as a rational polynomial whose numerator is the linear polynomial found as the product of the infinite series with its scale of relation, and its denominator is the scale of relation, and that this rational polynomial is fully equivalent to the infinite series. By this chicanery I have managed to encapsulate the whole infinite series in a rational polynomial. In J we represent a polynomial by a list of coefficients *c* bonded to the polynomial primitive *p.*, that is,

$$c \& p.$$

is a polynomial with coefficients *c*.

We can thus represent an infinite series by the rational polynomial function *gf*, using its product with the scale of relation as the numerator, and the scale of relation as the denominator.

$$gf = : 1 _2 \& p. \% 1 _4 1 \& p.$$

There isn't much we can do directly with *gf*, since the only meaningful arguments for it are those which make the infinite series converge, so we are restricted, if that is what we want to do, to arguments less than one in magnitude. But that isn't what we want to do. We are only interested in the coefficients of the terms in the series,

and J provides us with the tool needed to find these, and that is the Taylor coefficient adverb (`t.`). Thus if we apply `t.` to `gf`, and apply this derived function to any non-negative integer argument, the result will be the corresponding coefficient:

```
gf t. i. 12
1 2 7 26 97 362 1351 5042 18817 70226 262087 978122
```

Compared to the doubly recursive verb `A`, the time required by `gf t.` is significantly less and its advantage in speed increases rapidly with the size of the argument.

I estimate `A 30` would have taken 15 hours to complete on my computer, versus the 1.2 seconds taken by `gf t. 30`.

VI. Partial Fractions

Hall & Knight discuss the relevance of partial fractions in handling recurrences, and work through some examples. This leads to the ability to derive an even simpler expression for the general term of the series. The method works as follows: separate the generating function into a sum of partial fractions with constant numerators and linear denominators. That is, find constants a , b , A , and B such that:

$$gf \leftrightarrow \frac{A}{1-ax} + \frac{B}{1-bx} \quad (1)$$

The constants a and b are the roots of the scale of relation quadratic polynomial. These can be obtained using the polynomial rootfinder primitive, which is the monad of the verb `p.`, by

```
] 'a b'=: , > }. p. 1 _4 1
3.73205 0.267949
```

You might recognize these roots as

```
2 + %: 3 and 2 - %: 3
```

In (1) the denominators can be removed by multiplying each term by the scale of relation, giving:

```
1 _2&p. ↔ (A * (1 , -b)&p.) + (B * (1 , -a)&p.)
```

This linear system can be solved for A and B by writing

```
] 'A B'=: 1 _2 %: 1 1 ,: -(b,a)
0.5 0.5
```

and now we can write a function `gt`:

```
gt=: (A * a^]) + (B * b^])
gt i. 10
1 2 7 26 97 362 1351 5042 18817 70226
```

The function `gt` is 5 times faster than `gf t`.

But wait! Since `B` and `b` are each less than one, the right hand expression is always less than one and isn't really needed — we can replace it by a ceiling (`>.`). And since `A` is 0.5, we can replace it by halving (`-:`) giving us an even simpler expression:

```
gtt=: >. @ -: @ (a & ^)
gtt i.10
1 2 7 26 97 362 1351 5042 18817 70226
```

The function `gtt` is twice as fast as `gt`.

References

- [1] *FinnAPL Idiom Library*.
<http://www.pyr.fi/apl/texts/Idiot.htm>
- [2] Sloane, N. J. A., *On-Line Encyclopedia of Integer Sequences* (OEIS):
<http://www.research.att.com/~njas/sequences/>

10 Year's Digits for 1996

First published in Vector, 12, 4, (April 1996), 123-126.

Revised by Roger Hui, (April 2009).

This problem is a variation of an old one that originated as a Fortran puzzle in the MIT alumni magazine, adapted for use with J. Here it is:

Create a character table T, having 101 rows, each row representing a J expression, according to the following rules:

- (a) The result of executing row *i* must be the atom *i*,
and:
- (b) The characters '1', '9', '9', and '6' must appear in that order in each row, and no other digits may be present.
(In the Fortran puzzle, the digits could appear in any order.)

Expressed in J,

- (a) each row: `r =. i { T`
must satisfy the requirement that: `i -: ". r`
for *i* an item of `i.101` (and thus an atom),
and:
- (b) `'1996' -: r -. a. -. '0123456789'`

There are two additional requirements, suggested by Roger Hui:

- (c) Character constants are not permitted.
If they were then all solutions would need no more than two tokens. For example 7 could be represented by `#'1 9 9 6'`.
- (d) J allows **b**-form constants, in which a decimal integer base appears to the left of **b** and the digits to the right of **b** may include not only the digits 0 through 9 but also the letters **a** through **z**, representing digits 10 through 35.
For example, the octal representation of 63 is `8b77` and the hexadecimal representation of 255 is `16bff` and the decimal number 100 can be written as `1buzz`.
The **b**-form of constants is allowed, but the digits **a** through **z** are excluded, as well as `0 2 3 4 5 7 8`.
(If **a** through **z** were not excluded almost all solutions would be one token long.)

Here are some examples of invalid rows. The reason each example is unacceptable is given directly after it.

19+6+9	The digits are not in the prescribed order.
1+96+1	The digits are not 1996.
3*19[96	It contains a '3'.
1{.99 6	It yields a list result, not an atom.
#' 1996 '	It uses a character constant.
1bzp996	It uses the digits z and p.

As a valid example, row 19 might be

+ / 1 9 9 [6

and this satisfies the test

19 -: ". ' + / 1 9 9 [6 '

The objective of the problem is to use the minimum number of tokens in each row, as measured by the J 'Word Formation' primitive (;:). The foregoing list for row 19 has 5 tokens, and it is thus superior to:

1+9+9[6

which uses 7 tokens, but it is inferior to

19<.96

which uses only 3 tokens.

Entries will be judged in the following way: if *L* is the list of the number of tokens in each row of a given entry, and *M* is the list of the minimum number of tokens in all entries submitted, then the entry which minimizes $+ / L - M$ is the winning entry.*

To ease your minds, I should say that yes, a complete set of solutions is always possible, and this has been demonstrated mathematically by Donald Knuth and Roger Hui, among others. Since $*1996$ is 1 then $^{\wedge}.*1996$ is a solution for 0; and since $^{\wedge}.\circ.1$ is between 1 and 2, then applying floor or ceiling gives solutions for 1 and 2. Using more instances of \circ . provides solutions for larger numbers, ad infinitum. Clearly, this shows that a solution is always feasible. Most derived using this method are not, however, very short. Coming up with a short solution for each integer is your problem.

* This makes reference to a competition which is no longer running. For sample solutions to all numbers 0-100 see page 351.

To help you get started, let me suggest that you use a strategy like that employed by Roger Hui. He used a J session in the following way to develop his table: He worked with two windows present on his screen: an executable window, and a script window called `1996.js` which contained one solution per line. Initially, each row is set with the row number, a comma, some spaces, and a 0. For example, row 25 would look like this:

```
25,      0
```

You can write potential solutions in the script window, and have them executed in the execution window to see if they are correct:

```
25,      1+9+9+6
```

Roger provided himself with a suite of utility functions:

```
mat    =: (5&}.@}:);._2 @(1! :1) ➡
@(((<'\junk\1996.js')"_ )
len    =: /:~@((({.,#)/.~)@:(#@;:))
check  =: *./@ (0&= +. (=i.@#))@: ".
pfx    =: ' ' ,.~ [ : " : #@;: ,. i.@#
tab    =: [ : \:~ pfx ,. ]
```

mat reads the script file and constructs a matrix from it. As it stands, it is suitable for use with IBM-compatible PCs. To change it for use on Unix or Macintosh systems, you should replace the text `(5&}.@}:)` with `5&}. .` (*Note: the script must end with a `linend. Ed.`*)

check checks that each row is either zero (unsolved) or has the correct number.

len makes a two-column table with the first column giving a length and the second column giving the number of solutions with that length (unsolved numbers have a length of 0).

tab makes a table of the solutions sorted in decreasing length, and thus is handy for attacking the really bad solutions.

I wrote the following, to check that only the digits '1996' appear, in that order, in the solution:

```
d1996=.*./@([:( '1996' "_ -: ] -. a."_ -. ➡
'0123456789' "_)"1 ])
```

To see what these utilities can do for you, after you've created your `1996.js` file and filled in a few entries, experiment with expressions like:

```

$mat 0
check mat 0
len mat 0
+/*/"1 len mat 0 NB. total number of tokens
tab mat 0

```

And after you've filled in all the entries,

```
d1996 mat 0
```

This problem should help familiarize you with some lesser-known parts of J, like b-form constants, the new p: and q: primitives, and the monadic, or base-2 form of the base primitive (#.). For example, the following five-token expression:

```

# .p:q:|_19b96
91

```

creates the number `_19b96`, which has the decimal value `_165` (in base `_19` the values 9 and 6 evaluate to `_171` and 6, with sum `_165`); takes the magnitude of this number, yielding 165; finds its prime factorization with q:, yielding 3 5 11; uses p: to find the third, fifth and eleventh primes in the 0-origin series 2 3 5 7 11 13 17 ... , yielding 7 13 37; and applies the primitive #. to evaluate this list in base-2, yielding 91 (+/4 2 1*7 13 37). Another five-token expression for the same value is:

```

>:1#.q:996
91

```

There is a solution to 91 which is shorter than this, by the way.

Answers on page 351.

Endnote (Ed.)

The tradition begun by Eugene lives on in the J Wiki as *Puzzle of the year*: <http://www.jsoftware.com/jwiki/Puzzles/POY%202005...2006...2007...2008...2009...etc>.

11 Riding a Unicycle

First published in Vector, 13, 1, (July 1996), 154-158.

Revised by Gilles Kirouac, (April 2009).

This article deals with two topics dealing with permutations having a single cycle, which can be called unicycles. The first arises from a recent Internet inquiry, and the second resuscitates an obscure mathematician from two centuries ago to give him credit for having invented list processing.

We might ask how many unicycles there are for permutations of a given length. This number can be found by the use of Stirling numbers of the first kind, which came about precisely from a need to count the number of ways to arrange n objects into k cycles. In their APL95 paper, *Representations of Recursion*, Roger Hui and Ken Iverson give an efficient way to generate the table of values for these Stirling numbers for cycles:

```
S1v=: 1: `([S1r $:@<:) @. * " 0
S1r=: (0: ,]) + <:@[ * ],0:
S1v 4
0 6 11 6 1
S1v i.10
1      0      0      0      0      0      0      0      0
0      1      0      0      0      0      0      0      0
0      1      1      0      0      0      0      0      0
0      2      3      1      0      0      0      0      0
0      6      11     6      1      0      0      0      0
0     24     50     35     10     1      0      0      0
0    120    274    225     85     15     1      0      0
0    720   1764   1624    735    175    21     1      0
0   5040  13068  13132   6769   1960   322    28     1
0  40320 109584 118124  67284  22449  4536   546    36     1
```

As you can see, the number of ways that n objects can be arranged in a unicycle is $!(n-1)$.

I. The Bernecky Problem

Bob Bernecky, of Snake Island Research, in Toronto, sent a message on the Internet recently asking for help in deriving from the sequence of link fields in a linked list of records the permutation which would put the records into order. That is, suppose the records looked like this:

No.	Name	Link	
0	Bee	6	
1	Zee	0	
2	Que	8	
3	Gee	5	
4	Pea	2	(A)
5	Jay	9	
6	Dee	3	
7	Vee	1	
8	Tea	7	
9	Key	4	

The first record in the list is assumed to be in the leading position, but the locations of the other records is arbitrary. The ‘Link’ field in (A) gives the number of the successor record, and is 0 if it is the last record. In (A), the record following the first is number 6, and this is followed by number 3, which is followed by number 5, and so forth. It should be evident that the list of links is a permutation in (non-standard) cycles form, in other words, a unicycle.

What was desired was to have the records arranged as follows:

No.	Name	Link	
0	Bee	6	
6	Dee	3	
3	Gee	5	
5	Jay	9	
9	Key	4	(B)
4	Pea	2	
2	Que	8	
8	Tea	7	
7	Vee	1	
1	Zee	0	

The desired permutation is given by the ‘No.’ field in display (B), that is, 0 6 3 5 9 4 2 8 7 1. My usual way of exploring such problems is to head in the general direction where I imagine a solution may be found, with no maps or guides or bearers, and just beat my way unaided through the jungle with a machete. To my satisfaction I found that I could obtain the solution by applying raze (;) to the cycles-direct (C.) of the link list, and rotating this result so that it begins with 0:

```
y=: 6 0 8 5 2 9 3 1 7 4
(i.&0 |. ]) ; C. y
0 6 3 5 9 4 2 8 7 1
```


This is somewhat mysterious to me, since `C.` applied to an open list is supposed to convert from the direct form of a permutation to the cycles form, and here it looks as if the reverse is happening. (See the *J Introduction and Dictionary* for a description of the cycles and direct forms of a permutation.) Roger Hui provided me with the following explanation:

```
rot=: i.&0 |. ]

g0=: 3 : '{&y^:(i.#y) 0'
f0=: g0 { ]

fs=: {. "1 @ ({/\) @ (i.@# , <:@# # , :)
g1=: rot@fs
f1=: g1 { ]

g2=: rot@;@C.
f2=: g2 { ]
```

The `g`'s (and therefore the `f`'s) are equivalent on the vector `x`:

```
x=: 6 0 8 5 2 9 3 1 7 4
(g0 -: g1) x
1
(g0 -: g2) x
1
```

But they are not equivalent on arbitrary `x`:

```
g0 12?.12
0 10 3 4 6 1 2 5 7 9 8 0
g1 12?.12
0 10 3 4 6 1 2 5 7 9 8 0
g2 12?.12
0 11 10 3 4 6 1 2 5 7 9 8
```

In fact, the functions are equivalent exactly on those arguments that are a single cycle (`1: = #@C.`), and the explanation you seek lies in why `g2=:rot@;@C.` “works” on a single cycle.

The so-called “link list” representation of `x`:

```
0 1 2 3 4 5 6 7 8 9
x 6 0 8 5 2 9 3 1 7 4
```

specifies that 0 goes to 6, 6 to 3, 3 to 5, 5 to 9, etc., and that is what `C.` does in obtaining the cycle representation from the direct representation. If there is more than one cycle (if there is stuff left

over from this process), C. then does it again on the remaining elements to get the next cycle.

Since the argument is a single cycle, the raze of what results simply removes the boxing, and the rotation converts from the J convention of starting a cycle by its maximal element to the alternative convention of starting a cycle from 0.

II. Crelle’s Device

Histories of computing generally date the beginning of list-processing techniques to 1963 or so, with some possible smatterings of these techniques dating back to the days of Von Neumann, circa 1947. Imagine my surprise, then, to find that the date is off by over a hundred years.

The German engineer and mathematician August Leopold Crelle lived from 1780 to 1855. He made many minor contributions to mathematics, but is generally much better known as the founder and editor of the mathematical periodical *Crelle’s Journal*. His foreshadowing of list processing is described in L. E. Dickson’s monumental *History of the Theory of Numbers* (Vol. I, chap. VII, p. 185). First some discussion of primitive roots is necessary.

If we consider the powers of the positive numbers less than a given prime p , mod p , we note that some of these powers contain distinct elements, while others have repetitions. For example, when $p=7$ we get:

f=:] | [: ^/~ i.&.<:
f 7

1	1	1	1	1	1
2	4	1	2	4	1
3	2	6	4	5	1
4	2	1	4	2	1
5	4	6	2	3	1
6	1	6	1	6	1

(C)

and we see that the numbers 1 2 4 6 give rise to rows with repetitions, but rows 3 and 5 contain distinct elements. This property of 3 and 5 is what characterizes them as primitive roots of 7. Dickson wrote (in 1918):

A. L. Crelle[’s] ... device for finding the residues modulo p of the powers of a will be clear from the example $p = 7, a = 3$. Write under

the natural numbers < 7 the residues of the successive multiples of 3 formed by successive additions of 3; we get

1	2	3	4	5	6
3	6	2	5	1	4

Then the residues 3, 2, 6, of 3, 32, 33,... modulo 7 are found as follows: after 3 comes the number 2 below 3 in the table; after 2 comes the number 6 below 2 in the table; etc.

In other words, Crelle's device uses a linked list to convert from a list of multiples to a list of powers, mod some prime p . From the list

3 6 2 5 1 4

he produces

3 2 6 4 5 1

and this corresponds to the row beginning with 3 of table (C).

Crelle's list is clearly a single cycle, and thus a unicycle. His device works for each primitive root of a given prime. It is a way for converting from addition to multiplication, and is thus analogous to logarithms.

All hail Crelle, father of list processing!

12 Volutes

*First published in Vector, 13, 2, (October 1996), 144-153.
Revised by Joey Tuttle, (April 2009).*

This article describes an amazing algorithm that I learned from Joey Tuttle. It produces an integer volute. I call it amazing on good evidence, because I was amazed when he first showed it to me. I had written a function to produce such volutes many years ago [1], and thought I had done a fairly efficient job, but Tuttle’s analysis was far superior.

An integer volute can be drawn in a variety of ways. All of the cases we’ll consider place the integers in the cells of a rectangular, and usually square, table. Two kinds of volutes can be drawn in this manner: an involute and an evolute. In an involute the smallest integer appears in a corner of the table, and the integers increase as they get closer to the centre. In an evolute the largest number appears in a corner of the table, and the integers decrease as they get closer to the centre of the table.

0	1	2	3	4	24	23	22	21	20
15	16	17	18	5	9	8	7	6	19
14	23	24	19	6	10	1	0	5	18
13	22	21	20	7	11	2	3	4	17
12	11	10	9	8	12	13	14	15	16
involute					evolute				

In the involute above the numbers increase in a clockwise rotation. They could just as easily have increased in a counter-clockwise rotation. The number 0 appears in the top left corner of the table, but it could just as well have appeared in any of the other three corners. There are then eight possible ways of drawing the involute: with 0 appearing in any of four corners, and the numbers increasing in a clockwise or counter-clockwise rotation. Similar remarks apply to the evolute, mutatis mutandis. Furthermore, either form can be derived from the other by applying the verb 24&- to it. The least number is 0 in these volutes. A volute having 1 as the least number can be obtained from one of these by adding 1 to it.

Our verbs will take as argument the length of the side of the square

table. For example, if **e** is our evolve verb,

```

      e 3
    8 7 6
    1 0 5
    2 3 4

```

I'll give six solutions to the problem, each one increasing in speed. The best is a variation of the marvellous technique shown me by Tuttle.

In the book *Concrete Mathematics* by Graham, Knuth, and Patashnik, Exercise 3.40 takes a scalar approach to the problem of constructing an evolve.

It assumes an x,y coordinate system, with 0 at coordinates (0,0). It gives two ways of arriving at a solution, problems a and b. Both solutions produce a bottom right clockwise volute.

In problem a, it squares (*****) its argument, then produces the list of that many consecutive integers, beginning with 0. It finds the x and y coordinates of each number (**GKPax**, **.GKPay**), upgrades the resulting two-column table (**/:**), then reshapes (**\$**) this upgrade list into a square (**,~**) table. For example, for a square of side 3, problem a takes **n=: i.*:3** and yields x and y , upgrades this, and reshapes the upgrade:

```

      ] n=: i.*:3
    0 1 2 3 4 5 6 7 8
      ] xy=: (GKPax,.GKPay) n
    0 0
    0 1
    _1 1
    _1 0
    _1 _1
    0 _1
    1 _1
    1 0
    1 1
      ] w=: /:xy
    4 3 2 5 0 1 6 7 8
      (,~3)$w
    4 3 2
    5 0 1
    6 7 8

```

In problem a the placement of each integer requires the evaluation of two functions, **GKPax** to give the x -coordinate, and **GKPay** to give the

y -coordinate. A function `evGKPa` to give a square evolute of a given order is:

```
GKPae=: 0:=2&| NB. GKPae y=1 if y is even
GKPao=: 1:=2&| NB. GKPao y=1 if y is odd
GKPaq=: <.@+:@%: NB. floor double sqrt
GKPam=: <.@%: NB. m is floor sqrt
GKPal=: >.@-:@GKPam NB. ceiling half m
GKPar=: _1.^GKPam NB. parity (1 or _1)
GKPat=: (*>:@GKPam NB. m*(1+m)
GKPax=: GKPar*(([]-GKPat)*GKPae@GKPaq)+GKPal
GKPay=: GKPar*(([]-GKPat)*GKPao@GKPaq)-GKPal
evGKPa=: ,~ $ /:@(GKPax ,. GKPay)@i.@*:
```

In problem b the table of x - y indices is used to produce the integers, one at a time.

```
GKpb0=: +:@(>./"1)@:|
GKpb1=: >@, @{@(;~)@(>.@-:@- + i.)
GKpb2=: _1: ^ </"1
GKpb3=: *:@[ + GKpb2@] * [ + +/"1@]
GKpb4=: (GKpb0 GKpb3 ] )@GKpb1
evGKpb=: ,~$GKpb4
```

`GKpb1` produces the 2-column table of x, y coordinates. `GKpb0` gives a list of the doubles ($+$) of the maximum over ($>./$) the rows ($"1$) of the magnitudes (l) of the items in the table. `GKpb2` produces a list where the items are $_1$ where in the corresponding row x is less than y , and 1 otherwise. `GKpb3` squares its left argument, and adds this to the product of the $_1$ 1 list with the sum of the left argument and the sum of the rows of the right argument. `GKpb4` supplies the appropriate left and right arguments to `GKpb3`. The verb `evGKpb` reshapes this list, giving the same result as `evGKPa`. For example:

```
] xy=: GKpb1 3
 _1 _1
 _1  0
 _1  1
  0 _1
  0  0
  0  1
  1 _1
  1  0
  1  1
] ns=: (GKpb0 GKpb3 ] ) xy
4 3 2 5 0 1 6 7 8
```

```
(,~3) $ ns
4 3 2
5 0 1
6 7 8
```

In Vector 11 4, Keith Smillie [2] gives a suite of functions to produce a square evolute. (I've replaced his 'rows' function by #, made it 0-origin, and abbreviated his names `QuarterTurn` and `Wind` and `Spiral`). It produces its result by successive windings of new layers onto the beginning empty table.

The function `Wd` is, you will note, recursive:

```
QT=: [ , "2 |. @ | : @]
Wd=: [ `(((#@[{.]) QT [) Wd #@[}.])@.(0:<#@])
Sp=: (i. @ ((-/ @ ]), 0:)) W i.@(*/)@]
evKS=: (i.@(0:,0:)) Wd i.@*:@]
```

Smillie's volute is top right clockwise and has the advantage of not being limited to square results. You can find the details of the algorithm in his Vector article.

In my 1977 article I give a function like Smillie's in achieving its result by winding, but iterative (^:) rather than recursive. The function `evEEM0` below reverses and transposes (|:@|.) its argument (thus giving it a clockwise quarter turn), then appends as a new bottom row a vector of integers (i.) as long as the number of rows in the argument (#), with its first item the number of items in the argument (*/@\$). This is initiated with an empty table (i.0 0), and is repeated double (+:) the argument times. It is bottom right counter-clockwise.

```
evEEM0=: |:@|. , */@$ + i.@#
evEEM1=: [ : i.0 0"_[] NB. constant empty table
evEEM=: evEEM0^:(+:`evEEM1)
```

I sent an early draft of this paper to Roger Hui for comments, and he replied:

Motivated by your comment on `+/ \ ^ : _1`, I arrived at the following solution after studying the results of *

```
f=: +/ \ ^ : _1 @ evJKT
```

It is more direct but less concise than your solution; it takes less space, and is faster for *n* greater than about 200.

* The definition of the verb `evJKT` is given on page 91.

Here is the kind of thing Roger saw:

```

      f 5
    24 23 22 21 20
  _15 _15 _15 _15 _1
    1  _7  _7  _1  _1
    1  1   3  _1  _1
    1 11  11  11  _1

      f 7
    48 47 46 45 44 43 42
  _23 _23 _23 _23 _23 _23 _1
    1 _15 _15 _15 _15 _1 _1
    1  1  _7  _7  _1  _1 _1
    1  1  1   3  _1  _1 _1
    1  1 11  11  11  _1 _1
    1 19 19 19 19 19  _1

      f 9
    80 79 78 77 76 75 74 73 72
  _31 _31 _31 _31 _31 _31 _31 _31 _1
    1 _23 _23 _23 _23 _23 _23 _1 _1
    1  1 _15 _15 _15 _15 _1  _1 _1
    1  1  1  _7  _7  _1  _1  _1 _1
    1  1  1  1   3  _1  _1  _1 _1
    1  1  1 11  11  11  _1  _1 _1
    1  1 19 19 19 19 19  _1  _1
    1 27 27 27 27 27 27 27  _1

```

And here is his version:

```

even  =: 0: = 2&|
odd   =: 1: = 2&|
line0 =: *:@(-even) - odd + i.
c3    =: 1: ,. ] ,. _1:
top   =: odd   }. "1 ((| ,. ,+: ,. |.) # "1 ➡
c3@(_1&+ )@(_8&*) )@: >: @i. @-: @(-~ >: @even)
bot   =: -@even }. "1 ((| ,. ,<: @+: ,. |.) # "1 ➡
c3@(_5&+ )@(_8&*) )@: >: @i. @-: @(-odd)
evHUIb=: [: +/\ line0 , top , bot
evHUIf=: evHUIb f.

```

Hui's volute is top left counter-clockwise. If you look at the timings in the table below you will see that by size 89 Hui's version is as fast as Tuttle's (the unrounded ratio of Hui's to Tuttle's for case 89 was 1.4:1).

Now we come to Joey Tuttle's masterpiece. I asked him recently how he had arrived at it, but he no longer remembered. He had only a dim

recollection that it arose in connection with one of Martin Gardner's *Scientific American* columns. I conjecture that it may have been Gardner's column on Stanislas Ulam's spiral of primes (see Smillie's *Vector* article). So I don't know how it came to Joey, but I do know that until I went at the problem backwards from the conclusion, I had no idea how the function worked, so that's how I'll describe it to you.

I should say that Joey's function produced an involute, that is, he gave the result:

```

0  1  2  3  4
15 16 17 18 5
14 23 24 19 6
13 22 21 20 7
12 11 10 9  8

```

Now, we could get the result we desire by subtracting this from 24, but that would spoil my fun, so bear with me.

Let's begin with an evolute, and see if we can trace it back (we're reverse engineering) so that we can produce the function ourselves. In the course of doing this, we'll find it convenient to use J's ability to provide inverses for many primitive and derived verbs, using the power conjunction to the minus-1 power ($^:_1$):

```

evJKT 5
24 23 22 21 20
9  8  7  6 19
10 1  0  5 18
11 2  3  4 17
12 13 14 15 16

```

It seems to me that Joey must have had a series of insights. I assume that his first insight was to ravel this:

```

] q=: ,evJKT 5
24 23 22 21 20 9 8 7 6 19 10 1 0 5 18 11 2 3 4 ➡
17 12 13 14 15 16

```

I think he then had the tremendous insight that this list came about as the result of an upgrade. We'll get p, the permutation inverse to q, easily:

```

] p=: /: ^:_1 q
12 11 16 17 18 13 8 7 6 5 10 15 20 21 22 23 24 ➡
19 14 9  4 3  2 1 0

```

(Since upgrade is self inverse, we could have got p from q even more easily by writing $p = : / : q$ — but we’re not assuming that all readers will know this fact about upgrade.)

Now the final stupendous insight was to assume that p was produced by a sum scan of some list d , that is

$$p = . + / \backslash d$$

We can determine what the d was that gave us p by applying the inverse of sum scan:

$$\begin{array}{l}] d = . + / \wedge ^ : _1 p \\ 12 _1 5 1 1 _5 _5 _1 _1 _1 5 5 5 1 1 1 1 _5 _5 \rightarrow \\ _5 _5 _1 _1 _1 _1 \end{array}$$

This is beginning to look promising. At this point we can write the overall function $evJKT$:*

$$evJKT = : , \sim \$ / : @ (+ / \backslash) @ evJKT2$$

This $sumscans (+ / \backslash)$ the result of $evJKT2$, upgrades $(/ :)$ it, and reshapes $(\$)$ the upgrade into a square $(, \sim)$ table.

All this is great art — the rest is carpentry. Forget the leading 12 for the moment. If we do, we see that the other items all have the magnitude 1 or 5, and alternate in sign: first $_1$ and 5, then 1 and $_5$, and so on in alternation. There are nine groups of $_1 5 1 _5$, used cyclically. We note also that the groups increase in count: one each of $_1$ and 5, two each of 1 $_5$, three each of $_1$ and 5, then four each of 1 $_5$, and lastly an anomalous four of $_1$. We can generate the list of one each of the nine values easily:

$$evJKT1 = : < : @ + : \$ _1 : ,] , 1 : , -$$

The phrase $_1 : ,] , 1 : , -$ when applied to its argument gives us the right argument to the reshape verb:

$$\begin{array}{l} (_1 : ,] , 1 : , -) 5 \\ _1 5 1 _5 \end{array}$$

and the phrase $< : @ + :$ gives us the left argument:

$$\begin{array}{l} (< : @ + :) 5 \\ 9 \end{array}$$

* needs definition of $evJKT2$ from page 92.

so that the whole phrase gives us the list of values to be replicated:

```
(<:@+: $ _1: , ] , 1: , -) 5
_1 5 1 _5 _1 5 1 _5 _1
```

Now we need to specify how many times each of these is to be replicated:

```
evJKT0=: }:@(2: # >:@i.)
```

This verb begins by giving us a list of integers:

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

adds one to this:

```
>:i.5
1 2 3 4 5
```

gives us two of each:

```
2 # 1 2 3 4 5
1 1 2 2 3 3 4 4 5 5
```

and curtails (}:) this list:

```
}: 2 # 1 2 3 4 5
1 1 2 2 3 3 4 4 5
```

The last item should be a 4, not a 5, but as you shall see, we'll find it useful to produce an extra item.

The verb `evJKT0` encapsulates the whole:

```
evJKT0 5
1 1 2 2 3 3 4 4 5
```

Now we can write `evJKT2`:

```
evJKT2=: _1&|.@(evJKT0 # evJKT1)
```

The expression in parenthesis does the main job:

```
(evJKT0 # evJKT1) 5
_1 5 1 1 _5 _5 _1 _1 _1 5 5 5 1 1 1 1 _5 _5 _5 ➡
_5 _1 _1 _1 _1 _1
```

There is an extra `_1` at the end, but the next expression rotates this list one to the right, moving the extra `_1` to the beginning of the list:

```
_1&|. (evJKT0 # evJKT1) 5
_1 _1 5 1 1 _5 _5 _1 _1 _1 5 5 5 1 1 1 1 _5 _5 ➡
_5 _5 _1 _1 _1 _1
```

Why is the last `_1` rotated to become the leading item? A little thought will convince you that the value of the first item doesn't have to be

12; in fact its value is irrelevant—it could be any number at all! It serves only to act as a base for the ensuing sumscan, which, in turn, serves as the argument to upgrade. The upgrade will give the same result no matter what the value of the first item is; only the relative values of the items are important. And, since we are going to discard the last item of the list and then prefix the list with an arbitrary value, it suffices to combine these operations by performing the `_1` (right) rotate of the list.

The verb `evJKT` provides the finishing touches to the result of `evJKT2`:

```
evJKT=: (,~ $ /: @ (+/\) @ evJKT2)
```

It uses `evJKT2` to produce the list which it then sumscans:

```
+/\ evJKT2 5
_1 _2 3 4 5 0 _5 _6 _7 _8 _3 2 7 8 9 10 11 6 1 ➡
_4 _9 _10 _11 _12 _13
```

Notice that the items of this list are distinct and include all 25 integers from `_13` through `11`.

The upgrade of this list is obtained:

```
(/:@(+/\)@evJKT2) 5
24 23 22 21 20 9 8 7 6 19 10 1 0 5 18 11 2 3 4 ➡
17 12 13 14 15 16
```

and, at last, this list is reshaped (`$`) into the desired square integer evolute:

```
(,~ $ /: @ (+/\) @ evJKT2) 5
24 23 22 21 20
 9  8  7  6 19
10  1  0  5 18
11  2  3  4 17
12 13 14 15 16

evJKT 5
24 23 22 21 20
 9  8  7  6 19
10  1  0  5 18
11  2  3  4 17
12 13 14 15 16
```

The relative timings of the `GKPa`, `GKPb`, `KS`, `EEM`, `HUI`, and `JKT` verbs are of interest. The column headings give the size of table generated and the row stubs indicate the verb used.

The timings are relative to those of JKT set to 1.

verb\size	5	8	13	21	34	55	89
GKPa	12	23	85	110	200	224	244
GK Pb	5	10	37	47	83	94	106
KS	6	8	20	23	33	167	_
EEM	2	4	10	9	12	13	16
HUI	3	3	5	3	3	2	1
JKT	1	1	1	1	1	1	1

The infinite (`_`) entry for row `KS` in the column headed `89` indicates that there wasn't enough memory to complete execution. This was probably due to the many levels of recursion required.

The timings show the superiority of array strategies (`KS`, `EEM`, `HUI`, and `JKT`) over scalar strategies (`GKPa` and `GK Pb`), and the superiority of iteration (`EEM`) over recursion (`KS`) and the superiority of a strategy minimizing data movement (`HUI` and `JKT`) over strategies involving a great deal of data movement (`KS` and `EEM`).

I haven't discussed how the Tuttle algorithm can be modified to yield the other volute types. If you look back at the verb `evJKT1`, you'll see that the expression to the right of the reshape sign (`$`) is

```
_1: , ] , 1: , -
```

so that, given the argument `5`, it yields

```
_1 5 1 _5
```

The key is that this consists of `_1 5` followed by its negative, `1 _5`.

A little experimentation will convince you that the eight possible changes of sign and order of the list `_1 5` will give you all eight types of evolve:

<code>1 5</code>	top right clockwise
<code>1 _5</code>	bottom right counter-clockwise
<code>_1 5</code>	top left counter-clockwise
<code>_1 _5</code>	bottom left clockwise
<code>5 1</code>	bottom left counter-clockwise
<code>5 _1</code>	bottom right clockwise
<code>_5 1</code>	top left clockwise
<code>_5 _1</code>	top right counter-clockwise

So that to obtain a top left clockwise volute, one would use `_5 1 5 _1` instead of `_1 5 1 _5`.

A verb to yield involutes is somewhat simpler than the verb yielding evolutes.* The greater simplicity arises because no attention has to be paid to the first and last items: as they are generated so they are usable:

```

ivJKT0 =: |.@}:@ (2: # >:@i.)
ivJKT1 =: <:@+: $ [: (] , -) 1 , ]
ivJKT2 =: ivJKT0 # ivJKT1
ivJKT  =: ,~ $ /:@(+/\)@ivJKT2

ivJKT0 5 NB. just the reverse of evJKT0 5
5 4 4 3 3 2 2 1 1

ivJKT1 5 NB. essentially unchanged
1 5 _1 _5 1 5 _1 _5 1

ivJKT2 5 NB. different
1 1 1 1 1 5 5 5 5 _1 _1 _1 _1 _5 _5 _5 1 1 1 5 ➡
5 _1 _1 _5 1

+/\ivJKT2 5 NB. same sumscan
1 2 3 4 5 10 15 20 25 24 23 22 21 16 11 6 7 8 ➡
9 14 19 18 17 12 13

/:+/\ivJKT2 5 NB. same upgrade
0 1 2 3 4 15 16 17 18 5 14 23 24 19 6 13 22 21 ➡
20 7 12 11 10 9 8

5 5$/:+/\ivJKT2 5 NB. same reshape
0 1 2 3 4
15 16 17 18 5
14 23 24 19 6
13 22 21 20 7
12 11 10 9 8

ivJKT 5
0 1 2 3 4
15 16 17 18 5
14 23 24 19 6
13 22 21 20 7
12 11 10 9 8

```

* Thanks to Joey Tuttle for providing the definitions for `ivJKT0`, `ivJKT1`, `ivJKT2`, `ivJKT`, which were missing from the original published article. (Ed.)

References

- [1] McDonnell, E. E., Spirals & Time. *APL Quote Quad* 7, 4, (Winter 1977), 20-22.
- [2] Smillie, K., Primes, Spirals and Coffee Tables. *Vector*, 11, 4, (1995), 104-107.

13 Extended Integers

First published in Vector, 13, 3, (January 1997), 127-135.

Revised by Chris Burke, (April 2009).

Extended Integers

J has recently had added to it a new class of number, called *extended integer*. An extended integer can be produced by applying the extend verb `x:` to an ordinary integer.

```
12345678901
1.23457e10
x: 12345678901
12345678901
```

An extended integer may also be written directly by putting an `x` at the end of an ordinary integer:

```
12345678901x
12345678901
```

If one or more of the integers in a list is extended, they are all extended:

```
1 12345678901 2
1 1.23457e10 2
1 12345678901 2x
1 12345678901 2
```

Various primitives produce extended integer results if the argument is extended. For example, very large exact factorials are possible:

```
! 30x
265252859812191058636308480000000
*/ x: >: i. 30
265252859812191058636308480000000
```

Some verbs `f signal domain error` on some extended arguments because the result is not integral; however, `<.@f` and `>.@f` will work on extended arguments. I think you'll get the idea from a few examples: *

* The example showing `domain error` where the result is not integral, now returns an extended rational value, a feature which was added to J since this article was written.

and know y and g , we can find x by taking the base- g logarithm of y :

$$x = g^y$$

In solving (A) for the particular problem of APL\360 and its descendants (including J), x can be as large as 2,147,483,646. For this value of x , g^x has over 9,000,000,000 decimal digits, and would take several hundred large volumes to print out. There are tactics one can employ to cut the size of the problem down, but extended-precision arithmetic will still be required. In J's implementation, the phrase `M&|@^` is recognized and can be computed efficiently both in time and space.

The technique discussed by Shallit to solve the problem is due to Pohlig and Hellman [2]. You'll have to look up the references if you are interested in the mathematical background, since I shall focus on the problem's algorithmic aspects.

The Gory Details

Several global constants are needed. The modulus used in random number generators of the APL\360 kind must be a prime. The largest prime that can be stored as a 4-byte integer is in fact also the largest integer that can be stored, that is, one less than 2^{31} . This prime was discovered by Euler and for over a hundred years was the largest prime known. It is the Mersenne prime M31, too, for those of you interested in the Euclidian perfect numbers.

$$M =: x: <: 2^{31}$$

It is convenient to have the value of the integer one less than M handy:

$$L =: <: M$$

For the random number generator to have maximum period, the generator g must be a primitive root of the modulus. A primitive root of a prime has the property that its powers, mod the prime, are distinct. For example, the prime 7 has 3 and 5 as primitive roots, because their powers, mod 7, are distinct:

$$\begin{array}{r} 7 | 3 \ 5^{\wedge / > : i . 6 \\ 3 \ 2 \ 6 \ 4 \ 5 \ 1 \\ 5 \ 4 \ 6 \ 2 \ 3 \ 1 \end{array}$$

but the other positive integers less than 7 have repeated elements:

```
7|1 2 4 6^/>>:i.6
1 1 1 1 1 1
2 4 1 2 4 1
4 2 1 4 2 1
6 1 6 1 6 1
```

Dr. Bryant Tuckerman, of the IBM Watson Research Laboratory in Yorktown Heights, New York, gave the APL\360 implementors the primitive root 7^5 , or 16807. A decade or so later people began exploring random number generators extensively, and were surprised to find that there was no better generator than this for the modulus M:

```
g=: x: 7^5
```

A prime-power factor is an integer all of whose factors are the same. For example, 32, 9, 125, 49, and 11 are all prime-power factors. The prime-power factors of L play a key role in the algorithm. The primitive q: in J yields the prime factors of a number, but these may be repeated. For example, q:12 is 2 2 3. The algorithm requires that repeated primes be replaced by their product. The verb h, to be defined later*, factors numbers and replaces repeated items by their product:

```
f=: h L NB. f is 2 9 7 11 31 151 331
```

Certain powers of the generator g are needed. Those needed are the quotients of dividing L by f, and multiplying this quotient by the integers less than the items of f. For example, for the factor 7 we get:

```
,.B=: (L%7)*i.7
0
306783378
613566756
920350134
1227133512
1533916890
1840700268
```

and similarly for the other factors.

* To get this and subsequent examples to work, input the set of definitions given in the Appendix on page 105.

The verb `p`, to be defined later, raises the generator `g` to any integer power, mod `M`. We use it to raise the generator to the powers `B`:

```

      ,.C=: p (L%7) * i.7
      1
1600955193
 894255406
1205362885
1752599774
1537170743
1599590586

```

Such a list is made for each prime-power factor. These are boxed and joined together, forming `q`, a list of lists, containing 542 numbers altogether (+/2 9 7 11 31 151 331), and too large to display here.

```
q=: <@p@j"0 f
```

You should be warned that the formation of `q` takes a minute or so to execute, depending on the speed of your computer.* I find it convenient to comment out this line in the script, and insert the value of `q` directly.

We need the quotient of `L` with its factors as a separate global noun:

```

      ,. e=: L % f
1073741823
238609294
306783378
195225786
69273666
14221746
6487866

```

Those are all of the global nouns. Now we have to deploy a number of utility verbs. The verb `w`:

```
w=: ~. ^ #/.~
```

raises each item of its argument's nub to its tally:

```

      w 2 2 2 3 3 5 7
8 9 5 7

```

* 0.25 seconds on a laptop built in 2001 (*Ed.*)

The verb `h`:

```
h=: w @ q:
```

factors its argument and produces the prime power factors from it:

```
h L
2 9 7 11 31 151 331
```

The verb `s`:

```
s=: M&|^
```

raises its left argument `x` to the power of its right argument, mod `M`:

```
3 s 2
9
16807 s 2000
75099568
```

The verb `p`:

```
p=: g&s
```

raises `g` to the power of its argument:

```
p 2000
75099568
```

The verb `j`:

```
j=: L% * i.
```

divides its argument into `L`, and multiplies this quotient by the non-negative integers less than it.

```
,. j 7
0
306783378
613566756
920350134
1227133512
1533916890
1840700268
```

The Main Problem

With all this behind us, we're ready to discuss the main problem. Suppose we find the value of `y`, the random link, is 1209311799:

```
y=: 1209311799
```

We define a verb `t`:

```
t=: f ,. q i.> ] s e" _
```

The phrase

```
] s e" _
```

can be replaced by

```
,.D=: y s e NB. let this be (D)
2147483646
473297587
1537170743
353622995
4096
709324280
667991092
```

The heart of the matter is that each distinct value that y may take yields a different list D . We look for the index of each item of D in q , finding:

```
] E=: q i.> D
1 1 5 4 23 142 268
```

These values are the residues we seek. We form a table F by stitching f and E :

```
] F=: f ,. E
2 1
9 1
7 5
11 4
31 23
151 142
331 268
```

and this is also the result of applying verb t to y :

```
t y
2 1
9 1
7 5
11 4
31 23
151 142
331 268
```

If you refer to Hui's article, you'll see that F is similar to the table shown on page 64, beneath the expression $c\ mr\ q$. That is, F is a table of moduli and residues. For example, the items in list D correspond to the factors in f . In particular, the value $1537170743x$ corresponds to the factor 7. We gave all the possible values that

may be taken on in this position in list *C* back a bit. In *C* we find that the value **1537170743x** is in position 5, and so in *F* we find the value 5 next to the 7 in the first column. It is a remarkable fact that the expression `y s e` produces values which must occur also in the corresponding item of *q*, and that its index in the list in *q* is the residue we want for the next step.

The verb *r*:

```
r=: {: @ (cr1/ @ t)
```

inserts Hui's verb* *cr1* between each of the items of its argument, and yields a two-item list, with the first item necessarily equal to *L*, and the second item the power of *g* yielding *y*, mod *M*.

```
x: z=: cr1/F
2147483646 1234567
L
2147483646
```

We take the tail of *z* as our desired *x*:

```
] x=: {: z
1234567
```

We can verify that *x* is indeed the desired value by applying *p* to it:

```
p x
1209311799
y
1209311799
```

Are we happy? We shouldn't be yet, because this wasn't precisely the problem we wanted to solve, which was, how many times had the random number generator been used to arrive at the given random link. This part is easy, because we know that the initial value of the random link is 16807, which corresponds to exponent 1. All we have to do to get the value we want is to decrease *x* by 1. This gives us at last the verb *ner*:

```
ner=: <:@r NB. number of executions of roll

x: ner y
1234566
```

* See the Appendix on page 105. The verb *cr1* comes from Reference [1].

References

- [1] Hui, R. K. W., The Ball Clock Problem. *Vector* 12, 2, (1995), 55-66.
- [2] Pohlig, S. C., Hellman, M. E., An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Trans. Info. Theory* IT-24 (1978) 106-110.
- [3] Shallit, J. O., Merrily We Roll Along: some aspects of ?. *APL83 Conf. Proc.*, reprinted in: *APL Quote-Quad* 13, 3, (1983-03), 243-248.

Appendix

NB. Script for finding x in $y = m|g^x$, knowing y , M ,
 NB. and g , particularized for use in analyzing the
 NB. behavior of the linear congruential random
 NB. number generator found in APL\360 and its
 NB. descendants.

NB. UTILITY VERBS

```
w=: ~. ^ #/.~
h=: w@q:
```

NB. GLOBAL NOUNS

```
M=: x: <: 2^31
L=: <: M
g=: x: 7^5
f=: h L NB. f is 2 9 7 11 31 151 331
```

```
s=: M&|@^
p=: g&s
j=: L&% * i.
q=: <@p@j"0 f
e=: L % f
```

NB. HUI's CHINESE REMAINDER VERBS FROM VECTOR 12 2,
 NB. P 66, INCLUDING GCD AS A LINEAR COMBINATION

NB. Chinese Remainder

```
ab    =: |.@(gcd/ * [ % +./)@(&{.)
cr1   =: [: |/\ *.&{. , ,&{: +/ .* ab
chkc  =: [: assert ,&{: -: ,&{. | {:@cr1
cr     =: cr1 [ chkc
```

NB. GCD as a Linear Combination

```
g0    =: , ,. =@i.@2:
it    =: {: ,: {. - {: * <.@%&{./
gcd   =: (}).@{. )@(it^:(*@{.@{:})^:_))@g0

assert=: 13!:8@(12"_)^:-.
```

NB. MAIN VERBS

```
t=: f ,. q i.&> ] s e"_
r=: {: @ (cr1/ @ t)
ner=: <:@r NB. number of executions of roll.
```

14 Stumping the Rocket Scientist

First published in Vector, 13, 4, (April 1997), 123-129.

Revised by Roger Hui, (April 2009).

The Abstract Problem

This column concerns a statistical application, having to do with a rating problem involving five integer variables, related as follows:

```
a >: 0
c <: a
d < 100 * c
t <: c
i <: a - c
```

My interest in this application arose because the rating process is usually stated in quite a pedestrian way, yet has the reputation of being arcane and involved in the extreme. I'll give the pedestrian statement first, then an analysis of the statistical boundaries of the problem, next a J program following the statement as closely as possible, and lastly a J verb which is more concise and more efficient. In the second section I'll describe the physical situation giving rise to the statistical application.

To obtain the rating of a given system of these five variables proceed as follows:

Step 1: c divided by a. Subtract 0.3, then divide by 0.2.

Step 2: d divided by a. Subtract 3, then divide by 4.

Step 3: t divided by a, then divide by 0.5.

Step 4: Start with 0.095, and subtract i divided by a. Divide the product by 0.04.

The sum of each step cannot be greater than 2.375 or less than zero. Add the sum of steps 1 through 4, multiply by 100 and divide by 6. This is the rating.

We form the argument to the program as a five-item list:

```
a, c, d, t, i
```

I'll write the program in J. The first line shows the change brought by Release 3.03, January 1997 in the way of doing indirect assign-

ment; one-letter names are now treated in the same way as multiple letter names, that is, with a space separating names.

```
Rating=: verb define
  'a c d t i' =. y
  step1 =. ((c % a) - 0.3) % 0.2
  step2 =. ((d % a) - 3) % 4
  step3 =. (t % a) % 0.05
  step4 =. (0.095 - i % a) % 0.04
  (100*+/2.375<.0>.step1,step2,step3,step4)%6
)
```

The number of tokens in this program is easily found:

```
#;:5!:5<'Rating'
79
```

The time required by `Rating` is 0.024. The four steps are roughly, but not exactly, the same. My impulse is to see whether I can make them exactly similar, for if we can we can take advantage of the array processing abilities of J.

I take Step 1 as the pattern. It has the form:

$$((v \% a) - w) \% z$$

Step 2 follows the pattern exactly. Step 3 lacks the $- w$ part, but that is easily fixed using the identity:

$$\begin{array}{l} x - 0 \\ x \end{array}$$

Using this, we'll rewrite Step 3 as:

```
step3 =. ((t % a - 0) % 0.05
```

Step 4 is only slightly more complicated. It reverses the minuend and subtrahend.

```
step4 =. (0.095 - i % a) % 0.04
```

We can switch the two around by using the identity:

$$\begin{array}{l} (s - t) \% u \\ (t - s) \% -u \end{array}$$

to give us:

```
step4 =. ((i % a) - 0.095) % _0.04
```

What I had in mind by putting them in the same form was to be able to take advantage of J's array processing abilities to get rid of the four local variables by writing something like:

```
x =. (c, d, t, i) % a
```

or:

```
x =. (}. % {.) y    NB. behead divided by head
```

If we now form two lists, one of minuends and another of divisors, we can replace the four Step statements by:

```
m =. 0.3 3 0 0.095
n =. 0.2 4 0.05 _0.04
(x - m) % n
```

Next, reciprocate *n* to replace division by multiplication:

```
] b =. % n
5 0.25 20 _25
b * (x - m)
```

If we now distribute the multiplication within the parentheses we get:

```
(b * x) - (b * m)
```

And, since the right limb is the product of constants, we can replace it by its product:

```
] q =. b * m
1.5 0.75 0 _2.375
(b * x) - q
```

I'm trying to arrive at an expression involving a linear polynomial, and am almost there. I have in mind using J's polynomial primitive (*p.*). For that I'll have to form *a* as the negate of *q* and reverse the order of the terms:

```
] a =. - q
_1.5 _0.75 0 2.375
a + (b * x)
```

Whew! We've got our linear polynomial (actually, four of them). This has been tedious, although eventually interesting. We now can replace all of the steps of *Rat ing* by:

```
(100 * +/ 2.375 <. 0 >. a + (b * x)) % 6
```

or, using the polynomial primitive,

```
(100 * +/ 2.375 <. 0 >. (a , b) p. x) % 6
```

Looking at this, we get irritated by that *100 ** and that *% 6*.

We can use two identities:

```
u * +/ v
+/ u * v

(+/ v) % w
+/ v % w
```

and arrive, after a bit of algebra, at:

```
] e=: 100r6 * a
_25 _12.5 0 39.5833

] f=: 100r6 * b
83.3333 4.16667 333.333 _416.667

] g=: e ,. f
_25 83.3333
_12.5 4.16667
0 333.333
39.5833 _416.667

] h=: 100r6 * 2.375 NB. 39.5833 is 475r12
39.5833
```

Table **g** lists in its leading column the constant coefficients, and in the last column the linear coefficients for each of the four linear polynomials.

```
Rtg=: [: +/ 0 >. h <. g p. }. % {.
```

In this verb, the trailing four items are divided by the leading item, and used as the right argument to the polynomial primitive, with the left argument table **g**. The four evaluations are constrained to lie in the interval from 0 to 475r12, inclusive, and the constrained values are summed to give the rating.

The verb **Rtg** has 16 tokens and takes 0.007 units of time: about a quarter of the size, and less than one-third the time of the program **Rating**. Having the four linear polynomial coefficients allows us to determine the meaningful boundaries of all systems.

Table A

event	min	max
c % a	0.3	0.775
d % a	3	12.5
t % a	0	0.11875
i % a	0.095	0

Here's how to read this table:

If, for example, the result of `c%a` is 0.3 or less, the rating will be 0 for the `c%a` event. If it is 0.775 or greater, the rating will be `475r12`. Similarly for the next two rows. For the last row, a result for `i%a` of 0.095 or greater will give a rating of 0 for that event. A result of 0 (it can't be less) will give a rating of `475r12` for that event.

Here are some numerical examples:

The maximum rating can be obtained by the system of values:

```
mxr =: 800 620 10000 95 0
Rtg mxr
158.333
```

Recall that the ratings depend on the ratio of the trailing values to the leading value. When the leading value is 800, the list `mxr` produces the maximum rating of 158.333, since

```
620 10000 95 0 % 800
0.775 12.5 0.11875 0
```

give the values in the column headed `max` in Table A.

Changing the system to give the maximum values possible given the constraints listed at the beginning of this section does not give a greater result:

```
Rtg 800 800 80000 800 0
158.333
```

Conversely, the minimum rating (zero) is obtained with the system:

```
Rtg 800 240 2400 0 76 NB. result really 0
3.55271e_15
```

And similarly, we can say that changing the system to:

```
Rtg 800 0 0 0 800
0
```

will produce the same zero rating.

The Physical Problem

Now I have to apologize to readers outside of the United States of America for imposing on your good nature for so long, when what I was describing derives from the parochial form of football popular in the the USA but (I believe) not well-known outside

that country. In that game there is a preeminent hero called the quarterback. He stands behind a line of seven myrmidons, the central one of which (called the center), hands the ball between his legs to the quarterback while in a crouching stance and facing away from the quarterback.

The quarterback can hand the ball in turn to one of the three other people behind the line like himself, or can run with the ball, or he can throw it forward, aiming it in the direction of one of his running teammates. This is called a forward pass, and it is his ability to deliver forward passes so that they are caught by a teammate before hitting the ground that is measured by the rating system described so laboriously above.

The five variables so artfully abbreviated above are now made plain to you:

- a is the number of forward passes attempted.
- c is the number of passes caught by an eligible teammate.
- d is the distance traversed from the line to the point of completion of the play, for all pass plays.
- t is the number of completed passes which result in a goal, or *touchdown*.
- i is the number of attempted passes which are ingloriously caught by a member of the opposing team—an *interception*.

As a sample piece of data I'll use the lifetime data of the quarterback George Blanda, who played professional football in the USA for a number of teams from 1949 through 1975. Before showing you this data, I'll interject some personal history.

George Blanda and I were in the graduating class of 1949 at the University of Kentucky. George had been the successful quarterback of the college football team. He became a professional player immediately, and played for many years.

When my job moved my family and me to Palo Alto, California, in the fall of 1974, I became aware that my old classmate George was still playing football for a living, and not only that, but he was a stellar performer. Week after week it was he who saved the day in the last minute for his team, the Oakland Raiders.

(Oakland is a large city across the bay from San Francisco, and about thirty-five miles north of Palo Alto.)

I was 48, but felt a resurgence of youth in seeing what my coeval Blanda was still doing on the football field. He played through the seasons of 1974 and 1975 before finally retiring (actually he was forced out by his management, who wanted to bring in younger players).

George holds the career record for the total number of points scored by a football player, 2,002. The nearest player to him has scored 1,699 points.

Let us see then what George Blanda's lifetime statistics are:

attempts:	4007
completions:	1911
yards:	26920
touchdowns:	236
interceptions:	277

Applying our `Rtg` program gives us his career rating:

```
Rtg gb =: 4007 1911 26920 236 277
60.6475
```

Blanda doesn't have a particularly good rating largely because of the great number of interceptions he threw. Quarterbacks with high ratings usually have many more touchdown passes than interceptions. The quarterback Joe Montana, for example, while playing for the San Francisco football team compiled a record of:

```
Rtg jm =. 4600 2929 35124 244 123
93.5
```

This was the highest career rating for any quarterback to have played the professional game.

Ratings are also compiled during the football season, as well as for entire careers. Has anyone ever achieved the maximum rating? No one has ever done it for a career, or even for a season, but for a single game it has been done. The player John Taylor of the San Francisco team was called on in one game to throw the ball (he had never done this in a game before). It went for twenty yards, was completed, and scored a touchdown.

So Taylor's rating for that game was:

Rtg 1 1 20 1 0
158.333

I got the title for this column from the fact that American sports-writers and broadcasters are confident that the formula is so arcane it baffles even rocket scientists. We know better, of course. It really only baffles sportswriters and broadcasters.

15 Oh, No, Not Eigenvalues Again!

First published in Vector, 14, 1, (July 1997), 135-139.

Revised by Kip Murray, (April 2009).

I can't explain why it is that I keep running into problems associated with eigenvalues. I don't seek them out, and have no interest in them, but there it is — they keep cropping up before me and I have somehow to find a way to drive a stake through their hearts before I can go on to something else.

When the eigenvalue problem last loomed before me, I found that I had to go back to basics in order to come to terms with it. In the course of doing so, I happened upon a technique in J for finding eigenvalues which is eminently satisfying pedagogically, since it can be used for small matrices to show all the theory of eigenvalues, even though it is staggeringly inefficient, becoming dreadfully slow for matrices of size 5 or 6 or larger. It is its pedagogical merit that I commend to you.

The basics are simple. If A is a square numeric matrix, we replace each diagonal atom a_{ii} with the two-atom list a_{ii}_1 . What we are doing is replacing the problem of evaluating the determinant of a numeric matrix with that of evaluating the determinant of a matrix of polynomials. Of course, the non-diagonal terms are the simplest of polynomials, that is, constants, but the diagonal terms are all linear polynomials. Our problem is to evaluate the determinant of this polynomial matrix. In J we evaluate the determinant of a numeric matrix with the monad of the dot conjunction:

`det=: -/ . *` NB. (1)

For example,

```
] m=:2 3,:5 8
2 3
5 8
det m
1
```

The verb `det` is not directly suited to our purpose, but with a few manoeuvres we'll get where we want to go.

First, let's take a suitable matrix for our example (the example and much of the approach I have borrowed from Lanczos [1], chapter II):

```

] A=: 33 16 72 , _24 _10 _57 ,: _8 _4 _17
33 16 72
_24 _10 _57
_8 _4 _17

```

We can obtain the diagonal atoms of A using the monad d [JP 3.B.m4*]:

```

d=: (<0 1)&|:      NB. JP 3.B.m4
] a0=: d A
33 _10 _17

```

We append _1 to each of these using J's stitch (,.):

```

] a1=: a0 ,. _1
33 _1
_10 _1
_17 _1

```

In order to be able to work with a matrix in which a single number is replaced by a list of two numbers, it will be necessary to box the rows of the 2-column matrix above, using monad B1 [JP 1.C.m11]:

```

B1=: <"1          NB. JP 1.C.m11
] a2=: B1 a1
+-----+-----+
|33 _1|_10 _1|_17 _1|
+-----+-----+

```

In order to amend the diagonal atoms of A, we'll box the atoms of matrix A first, using monad B0 [JP 1.C.m10]:

```

B0=: <"0          NB. JP 1.C.m10
] a3=: B0 A
+---+---+---+
|33 |16 |72 |
+---+---+---+
|_24|_10|_57|
+---+---+---+
|_8 |_4 |_17|
+---+---+---+

```

* meaning phrase m4 from section B of chapter 3 of the book: *J Phrases*.

The amend adverb in J needs the indices of the places to be amended. We obtain these using the monad `d` again, modified by the adverb `IR` [JP 3.B.a3]:

```
IR=: @(i.@$@)      NB. JP 3.B.a3
] a4=: (d IR) A
0 4 8
```

We can now use `amend` to obtain the matrix we want:

```
] a5=: a2 d IR } a3
+-----+-----+-----+
|33 _1|16      |72      |
+-----+-----+-----+
|_24  |_10 _1|_57      |
+-----+-----+-----+
|_8    |_4     |_17 _1|
+-----+-----+-----+
```

For use later, we'll collect the steps taken so far to form the verb `db`:

```
db=: B1@(_1 ,.~ d) d IR } B0
```

Now we are in position to evaluate the determinant of this matrix. We can't use the `det` verb from above, since it can only deal with matrices whose atoms are simple numbers. My first thought was to use the definition of the Determinant conjunction `u . v` from the *J Introduction and Dictionary*:

```
u=: -/
v=: *
DET=: v/,@,`({."1 u .v$:@minors)@.(1&<@{:@$)"2
minors=: }."1@(1&([\.))
```

replacing the definitions of `u` and `v` by polynomial difference and polynomial product, dyads `dif` and `ppr` [JP 9.C.d1 and 9.C.d2]. However, I hadn't proceeded far when I suddenly realized that, since `DET` was the definition of the monadic form of the dot conjunction, I ought to be able to use `dot` itself directly, and not mess with its sybilline definition. The first thing I did was to provide myself with modified forms of `dif` and `ppr`, ones which operated on boxed atoms:

```
dif=: -/@,:      NB. JP 9.C.d1
ppr=: +//.@(*/)  NB. JP 9.C.d2
difb=: dif&.>
pprb=: ppr&.>
```

Taking this tack, I could now write, in direct analogy with (1),

```
detp=: difb/ . pprb
```

and then apply this to matrix a5:

```
detp a5
+-----+
|6 _11 6 _1|
+-----+
```

I wanted an open, not a boxed result, so modified the definition of detp:

```
detp=: >@(difb/ . pprb)
] a6=: detp a5
6 _11 6 _1
```

This looked promising, but was it indeed the characteristic polynomial of my matrix? One way to find out was to use the Cayley-Hamilton theorem, which says that a matrix satisfies its own characteristic equation. To see whether this was so, I found the first four powers of A, using the monad I [JP 9.A.m0] to form the 0th power of the matrix, that is, the identity matrix:

```
I=: =@i.@#          NB. JP 9.A.m0
ip=: +/ . *         NB. inner product
] a7=: A&ip^:(i.@>:@#`) I A
1      0      0
0      1      0
0      0      1

33     16     72
_24    _10    _57
_8      _4     _17

129     80    240
_96     _56   _189
_32     _20   _59

417     304    648
_312    _220   _507
_104     _76   _161
```

then multiplied these by the supposed coefficients of the characteristic equation:

```

] a8=: a6 * a7
6    0    0
0    6    0
0    0    6

_363 _176 _792
_264 110  627
 88   44  187

774  480 1440
_576 _336 _1134
_192 _120 _354

_417 _304 _648
_312 220  507
104   76  161

```

and lastly, summed a8, trusting that the result would be the zero matrix:

```

+ / a8
0 0 0
0 0 0
0 0 0

```

From here it was a short step to the eigenvalues: they are the roots of this polynomial, so I could use J's polynomial primitive (p.):

```

p. detp a5
+---+-----+
|_1|3 2 1|
+---+-----+

```

The result I wanted was the open (>) of the tail ({:) of this one, so I made another modification:

```

] a9=: >@{: @p. @detp a5
3 2 1

```

The parts can now be assembled to give the eigenvalue finder cm:

```

cm=: > @ { : @ p. @ detp @ db
cm A
3 2 1

```

If these are the eigenvalues, their product should be equal to the determinant of the matrix.

```

*/ cm A
6

```

det A
6

So far, so good. Furthermore, if these are the eigenvalues, if any one of them is subtracted from the diagonals of A, the determinant of the result should be zero. Is this the case?

det A - 3 * I A
2.4869e_14
det A - 2 * I A
_1.73195e_14
det A - 1 * I A
0

Yes it is.

The verb *cm* is a model of the monad of the *c .* primitive described in the *J Introduction and Dictionary*.^{*} It differs in that the roots are given in order of descending magnitude, which is how the polynomial (*p .*) primitive provides them, rather than the ascending order prescribed in the *Dictionary*. Since *c .* has not yet been implemented, it's anyone's guess how *p .* and *c .* will be reconciled. I've brought this matter to the attention of the proper authorities, so they do at least know that the problem exists.

Reference

[1] Lanczos, Cornelius, *Applied Analysis*, Prentice-Hall, (1956).

^{*} i.e. in 1996 when the article was written. The *c .* primitive no longer exists. (*Ed.*)

16 A Newer Random Link Generator

First published in Vector, 14, 4, (April 1998), 122-127.

Revised by Kip Murray, (April 2009).

The random link generator used in many J\APL systems is an example of the linear congruential kind introduced in 1948 by the mathematician D. H. Lehmer, of the University of California in Berkeley, California. It depends on two numbers, 16,807 and 2,147,483,647. The first is 7^5 , a primitive root of the second, which is Euler's prime, $<: 2^{31}$. The basis of the algorithms for the roll and deal primitives is the expression for computing the next link `r1` in the chain of random links:

```
r1=: 1x
] r1=: 2147483647 | 16807 * r1 NB. repeat
```

The chain is 2,147,483,646 links long, and contains all of the integers between 1 and 2147483646, but not in any easily discernible order. The first link in the chain is 16807, and the next few links are

282475249, 1622650073, 984943658, 1144108930, 470211272...

The values seem to be suitably random. What's more, they satisfy many of the tests mathematicians and computer scientists have devised for judging randomness.

This column discusses a generator of a very different kind, that has its roots in a method devised in 1958 by G. J. Mitchell and D. P. Moore. Donald Knuth's book *Seminumerical Algorithms* [1] had its first edition in 1969 (the year I bought my copy) and discusses random numbers extensively, but doesn't mention this method. It appears in the book's second edition (1981), but is treated warily. Knuth writes, "The only reason it is difficult to recommend [it] wholeheartedly is that there is still very little theory to prove that it does or does not have desirable randomness properties". By the time of the third edition (1998) it is obvious that a great deal of effort has been expended on it, and it is almost part of the canon. I first saw the technique described in Knuth's book, *The Stanford GraphBase* [2]. He describes it in the section called `GB_FLIP`, the name of the program used to generate random links for the other programs described in the book. The generator described here comes from this book.

The basis of the generator is a bit of mathematics, the gist of which Knuth gives as follows:

The subtractive method. If m is any even number and if the numbers a_0, a_1, \dots, a_{54} are not all even, then the numbers generated by the recurrence

$$a_n = (a_{n-24} \cdot a_{n-55}) \bmod m \quad (1)$$

have a period of at least $2^{55} - 1$, because the residues $a_n \bmod 2$ have a period of this length. Furthermore, the numbers 24 and 55 in this recurrence are sufficiently large that deficiencies in randomness due to the simplicity of the recurrence are negligible in most applications.

If something has a period of n , it means that it can't be made up as an integral number of repetitions of a smaller sequence. Knuth says that $2^{55} - 1$ is the smallest the period can be, but it is plausible that it is $2^{85} - 2^{30}$. Furthermore, the low-order bits of the generated numbers are just as random as the high-order bits. These are both very large numbers.

```
<:2^55x
36028797018963967
(2^85x)-2^30x
38685626227668132516855808
```

There is a usage problem in giving a name to them. In the UK the names of numbers go up in powers of a million. Thus billion, trillion, and quadrillion are a million to the second, third, and fourth power. In the US they go up in powers of a thousand, so million, billion, trillion, quadrillion are a thousand to the second, third, fourth, and fifth powers. The names are more appropriate in the UK usage, but the size of the denominating numbers are smaller and usually more convenient in the US usage. The first large number would be about 36,029 billion in UK usage, and 36 quadrillion in US usage.

```
36028,797018,963967 (UK)*
36,028,797,018,963,967 (US)
```

The larger number would be about 39 quadrillion in the UK and 39 septillion in the US. Whichever way you call it, it's big.

* Whether using short scale or long scale nomenclature, Britons do in fact follow US usage in separating thousands by commas. (Ed.)

38,685626,227668,132516,855808 (UK)
 38,685,626,227,668,132,516,855,808 (US)

Since there are only $<:2^{31} \times$ or 2,147,483,647 positive integers, it is clear that each integer will appear many times in the chain, but the period of the chain is indeed enormously long.

Here's how the generator works: if a random link is needed, select the one indexed by `gb_fptr`, from a list `A` of 55 random numbers, and subtract one from `gb_fptr`. Thus getting a random link requires trivial computation, an indexing and a subtract, simply:

```
gb_fptr { A
gb_fptr=: <: gb_fptr
```

There must be more to it than that, you say, and you are right. The tricky problems are how to get more random links when we've used all that are in the list, and, ultimately, where did the initial list come from? The processes involve a fair amount of crabwise sidling and snakewise slithering. I'll make it as clear as I can.

What does one do when all 55 links in the list have been selected? A program `gb_flip_cycle` is used to provide a new list of 55 links, guided by recurrence (1). Knuth's C program uses two successive for loops; the first subtracts in sequence the last 24 items of `A` from the first 24; the second subtracts the last 31 from the first 31 in sequence, depending on the explicit sequencing in the for loop specification to make sure that the overlap of 7 items in the middle of the two has the proper values each time it is traversed. We could do that, too, using the shiny new control words available in J, but prefer instead to emphasize the potential for parallelization by using the following scheme: subtract the last 24 from the first 24, then the new first 24 from the second 24, and lastly the new 7 following the first 24 from the last 7.

All these subtractions are mod m , and this is easily computed at the machine level by anding the result of subtraction with `16b7ffffffffff`. In the function below it is shown at the J language level by taking the residue mod `16b80000000`, yielding the same result.

The function `gb_flip_cycle` takes as argument a list of 55 random numbers, and yields a new such list. In terms of recursion (1),

its first use replaces a_0 through a_{54} by a_{55} through a_{109} . Each of **a**, **b**, and **c** is a two-row matrix of indices to the list.

For example, **c** is the matrix

```
48 49 50 51 52 53 54
24 25 26 27 28 29 30
```

The first row gives the indices of the minuend, which are also the indices of the result. The second row gives the indices of the subtrahend. For example, the amend **c g }** subtracts a_{79} through a_{85} from a_{48} through a_{54} , yielding a_{103} through a_{109} , and these replace a_{48} through a_{54} .

```
gb_flip_cycle=: monad define
a=. 0 31 +/ i. 24          NB. 1st 24 get (1st 24) -
                           NB. (last 24)
b=. 24 0 +/ i. 24          NB. 2nd 24 get (2nd 24) -
                           NB. (new 1st 24)
c=. 48 24 +/ i. 7           NB. last 7 get (last 7) -
                           NB. (7 after 1st 24)
NB. v0 is difference mod m of selected parts of
NB. argument
v0=. 16b80000000 | [: -/ {
NB. v1 is top row of 2-row left argument
NB. -- result address
v1=. [: {. [
NB. v2 is right argument: a 55 item list
v2=. ]
g=. v0`v1`v2               NB. gerund for amend
c g } b g } a g } y       NB. amend in three
                           NB. overlapping steps
)
```

If you look closely at the numbers, you'll notice that recurrence (1) isn't faithfully followed. Instead we have:

$$a_n = (a_{n-55} - a_{n-24}) \bmod m \quad (2)$$

Knuth claims that this doesn't make the results any less random.

You are probably asking, "Yes, but where did you get the original list of random links?" This is the part that happens only once, and that's good, because it is the most complicated and time-consuming part of the whole method.

In outline, a next random link is formed from three values: the current value *next*, the previous value *prev*, and the *seed*. The current value is subtracted from the previous value, and the modified *seed* is subtracted from this, mod m .

The random links are stored as they are developed in list A, but not sequentially. The first one is stored in the 21st position, index 20. The next number found will be stored 21 to the right, in position 41. And the next one 21 further to the right, in position 62. Oops. That addition is made mod 55, so it will actually be stored in position 7. And so on. There's a trifle of Fibonacci magic here: there are 34 positions to the right of the first one, and 55 in all, and the numbers 21, 34, and 55 are consecutive Fibonacci numbers. Also, 21 is relatively prime to 55, which means that the index will take on all values from 0 to 54, inclusive, once each. Knuth comments that the successive values are thus stored as far as possible from each other, and that the initializing would be rather poor if this dispersal were not done.

The value of *seed* is not constant; it can be any one of 31 possible values. For each of the initial list of random links, a new value of *seed* is obtained by rotating its 31 rightmost bits one position to the right.

```
bwr=: [: #. _1: |. (31$2) #: ] ➡
NB. rotate bits 1-31 to the right
seed=: bwr seed
```

and then performing

```
temp=: next
next=. m | prev - next + seed
prev=: temp
```

The *seed* used by Knuth is $-314159 \bmod m$, or 2147169489. This is also the initial value of *prev*. The initial value of *next* is 1, chosen because the method requires that there be at least one odd number in the first set of links.

```

    gb_init_randx=: monad define
NB. initial list of random numbers
A=. 55 # 0
m=. 2^31
seed=. m | y
next=. 1
prev=. seed
k=. 20
whilst. k ~: 20 do.
    A=. next k } A
    seed=. bwr seed
    temp=. next
    next=. m | prev - next + seed
    prev=. temp
    k=. 55 | k + 21
end.
A
)

```

To verify that A has been properly formed, you can look at the first few links formed. These should be:

```

A=: gb_init_randx _314159
20 41 7 28 { x: A
1 2147326568 1073977445 536517481

```

Knuth explains that the first set isn't random enough to be used. Once the sequence gets far enough from its beginning, however, the initial transients become less perceptible. Thus after the initial A is formed, in order to ensure the necessary degree of randomness, a new list is formed from it by using `gb_flip_cycle` five times. When this is done, the list is ready to be used. The complete initializing procedure is thus

```

    gb_init_rand=: monad define
gb_fptr=: _2
gb_flip_cycle^:5 gb_init_randx y
)

```

The initial value of A is obtained by

```
A=: gb_init_rand _314159
```

A link in the random link chain is obtained using the function `gb_next_rand`:

```

    gb_next_rand=: monad define
  if. gb_fptr > _56 do.
    z=: gb_fptr { A
    gb_fptr=: <: gb_fptr
  else.
    A=: gb_flip_cycle A
    z=: _1 { A
    gb_fptr=: _2
  end.
  z
)

```

The links are accessed using the index value `gb_fptr`, and in reverse order. This doesn't affect the randomness of the links. Thus the first link taken would be the one at index position 54, which may be accessed (using contracurrent indexing) with `gb_fptr` set to `_1`, and then subtracting 1 from `gb_fptr`. The links are taken from the current list of links in the `if.` clause until `gb_fptr` reaches `_56`, at which time the `else.` clause is used to form a new list of links. Item `_1` of this new list is returned, and `gb_fptr` is set to `_2`.

To verify that initializing has been done correctly, check the result of `gb_next_rand`:

```

x: gb_next_rand ''
119318998

```

Unlike the linear congruential generators, subtractive generators are much less choosy about the seed value used. I believe that any value with a generous balance of 1s and 0s will do. Thus this one method gives rise to an enormous number of random link chains, each enormously long, obtainable simply by varying the seed.¹

A script combining the code in this article is given on the next page as an Appendix.²

¹ Knuth [1] announced a proof by R. P. Brent that the period of the subtractive method is exactly $2^{85} - 2^{30}$ for $m = 2^{31}$. See exercises 23 and 30 in Section 3.2.2. (*Ed.*)

² Not present in the original article. Added in the Second Edition, April 2009.

References

- [1] Knuth, D., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Third Edition, Reading, Massachusetts: Addison-Wesley, (1997), ISBN 0-201-89684-2.
- [2] Knuth, D., *The Stanford GraphBase: A Platform for Combinatorial Computing*, Reading, Massachusetts: Addison-Wesley, (1993). ISBN 0-201-54275-7.

Appendix

```

    gb_flip_cycle=: monad define
a=. 0 31 +/ i. 24      ➡
NB. 1st 24 get (1st 24) - (last 24)
b=. 24 0 +/ i. 24      ➡
NB. 2nd 24 get (2nd 24) - (new 1st 24)
c=. 48 24 +/ i. 7      ➡
NB. last 7 get (last 7) - (7 after 1st 24)
NB. v0 is difference mod m
NB. of selected parts of argument
v0=. 16b80000000 | [: -/ {
NB. v1 is top row of 2-row left argument
NB. -- result address
v1=. [: {. [
NB. v2 is right argument: a 55 item list
v2=. ]
g=. v0`v1`v2           NB. gerund for amend
c g } b g } a g } y
NB. amend in three overlapping steps
)

```

```

    bwr=: [: #. _1 |. (31$2) #: ] ➡
NB. rotate bits 1-31 to the right

```

```

    gb_init_randx=: monad define
NB. initial list of random numbers
A=. 55 # 0
m=. 2^31
seed=. m | y
next=. 1
prev=. seed
k=. 20

```



```

whilst. k ~: 20 do.
  A=. next k } A
  seed=. bwr seed
  temp=. next
  next=. m | prev - next + seed
  prev=. temp
  k=. 55 | k + 21
end.
A
)

```

```

  gb_init_rand=: monad define
gb_fptr=: _2
gb_flip_cycle ^:5 gb_init_randx y
)

```

```

A=: gb_init_rand _314159

```

```

  gb_next_rand=: monad define
if. gb_fptr > _56 do.
  z=. gb_fptr { A
  gb_fptr=: <: gb_fptr
else.
  A=: gb_flip_cycle A
  z=. _1 { A
  gb_fptr=: _2
end.
z
)

```


17 To Summarise

First published in Vector, 15, 1, (August 1998), 132-137.

Revised by Roger Hui, (April, 2009).

The key adverb in J, represented by slashdot (`/.`) is defined in the *J Dictionary* as:

`x u/.y ↔ (=x) u@# y`

that is, items of `x` specify keys for corresponding items of `y`, and `u` is applied to each collection of `y` having identical keys. For example:

```
1 2 3 1 3 2 1 </. 'abcdefgh'
+---+---+---+
|adg|bf|ce|
+---+---+---+
```

This may be clearer if we look at the separate parts.

```
=x
1 0 0 1 0 0 1
0 1 0 0 0 1 0
0 0 1 0 1 0 0
```

The first row of this has 1s in the first, fourth, and seventh positions, so when used as the left argument to copy (the dyad of `#`), and applied to `y`, yields its first, fourth, and seventh items, or `'adg'`; similarly the second row yields `'bf'` and the third row yields `'ce'`. Each of these is then boxed and the three are concatenated together, yielding

```
+---+---+---+
|adg|bf|ce|
+---+---+---+
```

The basic idea remains the same when `u` changes from box to a different monad. For example, if we replace box by tally (the monad of `#`) we get:

```
x #/. y
3 2 2
```

The same three groupings are selected, but instead of being boxed they are tallied, or counted, yielding the count of each group; three in the first, and two in the second and third.

The key adverb was not in the initial version of J. It came in later at the request of the J user community, notably Joey Tuttle.

```
acct=:1001 1002 1003
] table=:((?10#3){acct),.?10#100
1001 51
1003 83
1002 3
1002 5
1001 52
1001 67
1003 0
1003 38
1003 6
1002 41
```

```

+//./|:table
170 127 49

```

This result may not be completely satisfactory, since the amounts are not in the order of the accounts: they are in the order in which they fortuitously occur in the table. One way to remedy this is to place some dummy rows at the beginning of the table, one for each account, with the accounts in the desired order, and with the

* Results will vary, because the expression makes use of '?'.

amounts set to zero (`acct,.0`).

```
(acct,.0),table
1001  0
1002  0
1003  0
1001 51
1003 83
1002  3
1002  5
1001 52
1001 67
1003  0
1003 38
1003  6
1002 41
```

Now when we summarise the amounts will be in account order.

```
+//.//|:(acct,.0),table
170 49 127
```

We can produce a summary of accounts and amounts by prefacing the above with the list of account numbers and stitching (`,.`) the lists together.

```
acct,+.//.//|:(acct,.0),table
1001 170
1002  49
1003 127
```

This gives you the theory and the practice of the key adverb, so it's time to play, and incidentally to learn another way to use key.

How are the digits of π distributed? If the digits were distributed evenly, then the frequency of occurrence of all digits would be about 10%. J enables you to compute as many digits of π as you have room for and time for. A convenient way to obtain n digits of π is to subtract 1 (`<:`) from n , make this an extended integer (`x:`) use this as an exponent of 10 (`10^`), apply floor atop π times (`<.@o.`) and take the format ("`:`") of this:

```
dp=: monad def '":<.@o.10^x:<:y' NB. digits of pi
```

Try this on a small integer:

```
] q10=: dp 10
3141592653
```

This is correct.

Try it on a somewhat larger integer:

```
] q30=: dp 30
314159265358979323846264338327
```

Checking this against the value of π to many places in a table such as may be found in a volume of Knuth's *The Art of Computer Programming* shows that q30 is accurate, too.

Let's compute some more (q3000 may take several minutes):*

```
q100=: dp 100
q300=: dp 300
g1000=: dp 1000
q3000=: dp 3000
```

Now let's see how the digits are distributed in each of these, in order. We need a digit distribution function. This is where a new use of key comes in. In order to make the result of the digit distribution function be in the right order, we'll preface the argument with d, a list of the decimal digits in order.

```
d=: '0123456789'
```

To get the distribution we preface the formatted digits of π with d (d,y), then apply count (#) key (/.) reflexive (~) to this, giving us the distribution of d,y, then subtract 1 (<:) to adjust the count for the presence of d.

```
dd=: monad def '<:#/.,~d,y'
```

And try this out on q10, which is easy to verify by eye:

```
/:~q10
1123345569
dd q10
0 2 1 2 1 2 1 0 0 1
```

No zeros, two ones, one two, two threes, one four, two fives, one six, no sevens or eights, and one nine. Now let's see the digit distribution of each of the other lists of π digits.

```
dd q30
0 2 4 7 3 3 3 2 3 3
dd q100
8 8 12 12 10 8 9 8 12 13
```

* 65 seconds on a laptop built in 2001 (Ed.)

```

dd q300
26 30 35 31 37 27 31 19 34 30
dd q1000
93 116 103 103 93 97 94 95 101 105
dd q3000
259 308 303 266 318 315 302 287 310 332

```

These look somewhat reasonable, but it would be better to see how closely each gets to having 10% of each digit, using a function `pd`, which takes a distribution as argument, and yields the percentage of each value, rounded to the nearest one per cent. Do this by dividing the values by the sum of the values (`y%/y`), multiply this by 100 (`100*`), to get percentages, and round, getting the nearest percentage, by adding a half (`0.5+`) and taking the floor (`<.`).

```

pd=: monad def '<.0.5+100*y%+/y'
pd dd q10
0 20 10 20 10 20 10 0 0 10

```

We can compare this to `dd q10` and see that it is simply the same values multiplied by 10 to give percentages, as desired. There are so few digits to take into account that it is difficult to say whether the distribution is even or not. Trying the next distribution, of thirty values, still leaves us uncertain.

```

pd dd q30
0 7 13 23 10 10 10 7 10 10

```

There are no zeros among the first thirty digits, and a lot of threes. Probably still not enough digits.

```

pd dd q100
8 8 12 12 10 8 9 8 12 13

```

Except for the large number of nines, this is beginning to look quite even.

```

pd dd q300
9 10 12 10 12 9 10 6 11 10

```

Here, the number of sevens seems too low. Let's keep looking.

```

pd dd q1000
9 12 10 10 9 10 9 10 10 11

```

Ones seem a bit high, but I'd say this distribution is even enough.

```

pd dd q3000
9 10 10 9 11 11 10 10 10 11

```

With 3000 digits to distribute, we can say with some satisfaction that this represents an even distribution.

Before we part, let's look at a consecutive portion of these digits:

(762+i.6){q1000
999999

Hmmm. Well, yes, that's not too unusual.* In fact, if such strings *didn't* occur every now and then, it would argue against randomness.

* Eugene has selected the so-called *Feynman Point*, a six-digit sequence 999999 in the decimal expansion of π to which Richard Feynman (1918-1988) used to draw attention in his lectures to make an instructive joke about what is and what is not perceived as random. (*Ed.*)

18 Maximum Infix Sums

First published in Vector, 15, 2, (October 1998), 100-103.

Revised by Chris Burke, (April 2009).

In his book *Programming Pearls* (Addison-Wesley 1986), Jon Bentley collected a baker's dozen of his columns of the same name that had appeared in *Communications of ACM*. Column 7 therein is called "Algorithm Design Techniques", and the problem discussed in there is the subject of this column.

Here's the problem:

Given a vector x of numbers (positive, negative, or zero), define a function f that yields the maximum of the sums of all the infixes. For example, if

x
31 _41 59 26 _53 58 97 _93 _23 84

then $f\ x$ is 187, the sum of 59 26 _53 58 97. If all the numbers are positive the maximum infix is x , with sum $+x$. When all inputs are negative the maximum infix is the empty vector, with sum 0.

How many infixes are there in a vector? An infix can start at any point and end at any point. The number of infixes starting at the very beginning is $\#x$, one for each possible ending point. The number of infixes beginning at the next position is $(\#x) - 1$, at the next is $(\#x) - 2$, and so on; the total is thus the triangular number $-: (\#x) * (1 + \#x)$, or, expressed functionally, $-:@(*>:)\#x$, one of the series 1 3 6 10 15... The number of infixes for $\#x$ where x is a power of 10 are:

$\#x$	$-:@(*>:)\#x$
1	1
10	55
100	5050
1000	500500
10000	50005000
100000	5000050000
1000000	500000500000

So even for a vector of length 1,000, the number of infixes to consider is greater than half a million.

Bentley gives as a bad example an algorithm (written in Pascal) that tests the sum of each infix, and shows that a computer that takes an hour for `#x 1000` will take 39 days for `#x 10000`.

```
MaxSoFar := 0.0;
for L := 1 to N do begin
  for U := L to N do begin
    Sum := 0.0;
    for I := L to U do
      Sum := Sum + X[I];
    {Sum now contains the sum of X[L..U]}
    MaxSoFar := max(MaxSoFar, Sum);
  end;
end;
```

He goes on to discuss faster algorithms, but which still take quadratic time. He describes then a subquadratic algorithm which uses $O(N \log N)$ time. He tells how he and a colleague thought this was probably the best possible. They described this at a meeting at Carnegie Mellon University, and someone in the audience designed a much improved linear time algorithm in less than a minute. Several APL colleagues of mine have voiced their suspicions that the designer knew APL.

The code (again written in Pascal) for the linear-time algorithm is:

```
MaxSoFar := 0.0;
MaxEndingHere := 0.0;
for I := 1 to N do begin
  {Invariant: MaxEndingHere and MaxSoFar are
   accurate for X[1..I-1]}
  MaxEndingHere := max(MaxEndingHere+X[I], 0.0);
  MaxSoFar := max(MaxSoFar, MaxEndingHere);
end;
```

This can be translated into a J verb:

```
mis=: 3 : '>./(0>.+)/\y,0' NB. maximum infix sum
```

We append 0 to the end of the list argument (`y,0`), because the last value on the list may be negative, then do a suffix scan (`/\.`) using the verb 'the maximum of zero and the sum so far' (`0>.+`). This produces a list of sums formed according to the rule that if a negative sum is encountered it is replaced by zero. Finally, the maximum of all the sums is yielded (`>./`). To see how this works we use `mis0`, which is like `mis` but without the maximum-over, on a short list:

```

mis0=: 3 : '(0>.+)/\ .y,0'
mis0 y=: _100 2 3 4 _100 5 6 _7 8 9 _100
0 9 7 4 0 21 16 10 17 9 0 0
mis y
21

```

The maximum infix is 5 6 _7 8 9, with sum 21.

The need for appending a zero may be demonstrated by using `mis1`, which is like `mis0` but without the appended zero:

```

mis1=: 3 : '(0>.+)/\ .y'
mis1 y
0 9 7 4 0 12 7 1 8 0 _100

```

Jon Bentley showed the advantage of linear over quadratic algorithms by writing his slow algorithm in fine-tuned Fortran and running it on a Cray-1 supercomputer, and the linear algorithm on a Basic interpreter running on a Radio Shack TRS-80 Model III microcomputer. The runtime for an argument of length N was $3.0N^3$ nanoseconds for the slow algorithm on the Cray, and $19.5N$ milliseconds ($19,500,000N$ nanoseconds) for the fast algorithm on the TRS-80. The linear algorithm on the slow machine caught up with the slow algorithm on the fast machine at $N=2500$, and for $N=1,000,000$, the slow algorithm/fast machine took 95 years (estimated) and the fast algorithm/slow machine took just 5.4 hours.

The moral of the story is something like, “It’s foolish to improve performance by getting a faster machine; one’s money is better spent on finding a faster algorithm.” This isn’t always possible, of course. Machine designers are always finding ways to build faster machines, but it takes a very rare skill to devise faster algorithms.

This brings me to a confession. The J version of the algorithm wasn’t translated from Pascal (which is a language I don’t understand), but from Arthur Whitney’s K. Many of you will know of the existence of two different APL successors, J and K; some of you will know of their progenitors: Ken Iverson and Roger Hui for J, and Arthur Whitney for K. There are connections among these three. All are from Alberta, in Canada (Roger Hui by way of Hong Kong); all worked for I. P. Sharp Associates in Toronto (Whitney also worked for IPSA in Australia, Hong Kong, and Singapore), and all are good friends. I might also say that all have

first-rate brains. That they have taken divergent paths in pushing the APL idea into the next millenium might lead you to think that their relationship might be strained, and they do indeed have a rivalry in that regard; but it is a friendly, though determined one. I use both J and K, and believe that each language is excellent, but also that each has features that I wish the other had.

The K algorithm, which I learned from Arthur Whitney (we live just a mile or so apart in Palo Alto, California) is:

`f : |/0(0|+)\x}`

In K the vertical bar (|) signifies the dyad ‘larger’. So |/ is ‘the largest of all, or maximum’. This is applied to a scan using the hook 0|+, where 0| is the monad ‘larger of 0 and the argument’, and + is the dyad ‘plus’, of course. In K the scan (like over) begins at the left. The 0 preceding (0|+) is the initial value of the scan, and serves the same purpose as the appended 0 at the end of the J function `mis`.

Since I have both J and K on my machine (K can be downloaded from the website www.kx.com) I was motivated to time `mis` and `f` on similar length arguments. The results are interesting.

The arguments are named `xn`, where 10^n is the length of the list.

	J	K
<code>x1</code>	0.00017	0.000386
<code>x2</code>	0.0005	0.0024
<code>x3</code>	0.0077	0.0212
<code>x4</code>	0.082	0.089
<code>x5</code>	0.87	0.532
<code>x6</code>	--	5.091

J is faster for arguments up to 1,000 items. and there is a tie for arguments 10,000 items long, but K is faster for larger arguments, and, indeed, can handle much larger arguments than J can. I’ve reported this difficulty with J and perhaps by the time you read this it will have been remedied.

Endnote (Ed.)

The J timing problem has now been fixed (April, 2009) and performance is an order of magnitude better.

K is faster for smaller arguments, and slower for larger. The following updated timings are in milliseconds:

	J	K
x1	0.0129	0.0052
x2	0.0482	0.0441
x3	0.392	0.419
x4	3.77	4.19
x5	37.96	42.7
x6	393	476

19 Crosswords and Life

First published in Vector, 15, 3, (January 1999), 99-106.

Revised by Oleg Kobchenko, (April, 2009).

Making Shift

To refresh your memory about J's shift verb, here is what the online *J Dictionary* says:

The phrase `x |.!.f y` produces a shift: the items normally brought around by the cyclic rotation are replaced by `f` unless `f` is empty (`0=#f`), in which case they are replaced by the normal fill defined under `{.` (take):

```
t=: 'abcdefg'
2 _2 |.!. '#' 0 1 t
cdefg##
##abcde
```

The right shift is the dyadic case of `|.!.f` with the left argument `_1`. For example:

```
|.!. '#' t
#abcdef
```

This article uses the shift verb in numbering the squares of crossword puzzles and in Conway's Game of Life. The use of the shift verb is similar in both: it applies to tables along both the columns and the rows. It also applies to the neighbours of an atom: those reachable by a one-square rook move in the case of the crossword puzzle, and those reachable by a one-square queen move in the case of Life.

All of the shifts are of amount one. A shift of one is *ahead* if the last item is shifted out, and a fill item is introduced at the beginning; that is, the direction of shift is toward the larger index. A 'shift one ahead' verb is thus:

```
soa=: |.!. ''
```

Notice that the fill item is empty (`''`), so that `soa` can be used with nouns of any type, with the default fill for that type being used. In our cases we shall be working with numeric data, so the fill item will be the number zero.

```
L=: 2 3 4 5 6
soa L NB. shift the last item out
0 2 3 4 5
```

```

      ] T=: 1 2 3 +/ L
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
      soa T      NB. shift the last item out
0 0 0 0 0
3 4 5 6 7
4 5 6 7 8
      soa"1 T    NB. shift the last item of each item
(row) out
0 3 4 5 6
0 4 5 6 7
0 5 6 7 8

```

Similarly, a shift of one is *back* if the first item is shifted out, and a fill item is introduced at the end; that is, the direction of shift is toward the smaller index. A 'shift one back' verb is thus:

```

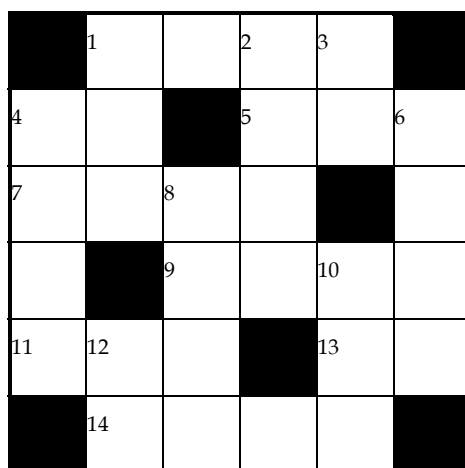
      sob=: 1&soa
      sob L      NB. shift the first item out
3 4 5 6 0
      sob T      NB. shift the first item out
4 5 6 7 8
5 6 7 8 9
0 0 0 0 0
      sob"1 T    NB. shift the first item of each item
(row) out
4 5 6 7 0
5 6 7 8 0
6 7 8 9 0

```

A Crossword Problem

A crossword puzzle (the kind I am familiar with, that appear in the newspapers in the United States) consists of two tables of numbered clues, labelled 'Across' and 'Down', together with a usually square diagram containing regularly spaced rows and columns which divide it into smaller squares, some black, but most white, in a usually symmetrical pattern. Some of the white squares contain numbers, and these indicate the beginning of an infix of squares going across or down, where the letters of a word are to be written, and these are keyed to the clues. The numbers are in ravel, or row-normal order, beginning with 1.

Here is a small crossword puzzle diagram, with its clues:



Across

1. Competent
4. Direction
5. Australian bird
7. Ran away
9. Damn euphemism
11. Bath, for instance
13. That is
14. Leave out

Down

1. Common abbreviation
2. Helen's mother
3. Printer's measure
4. Phantasmic celestials
6. Tempt
8. Cheese
10. Small thing
12. Italian river

The pattern of a crossword puzzle can be represented by a matrix in which a white square is represented by a one, and a black square by a zero. For example:

] M=: #: 2b0111110 2b110111 2b1111101 2b101111 ➡
2b111011 2b011110

```

0 1 1 1 1 0
1 1 0 1 1 1
1 1 1 1 0 1
1 0 1 1 1 1
1 1 1 0 1 1
0 1 1 1 1 0

```

A professional composer of such puzzles undoubtedly knows from experience which squares are to contain a clue number. An inexper-

enced computer has to be taught. We'll show how to teach a computer to number the squares, with the help of J's shift instruction.

In Exercise 1.3.2-23 of his book *Fundamental Algorithms*, Donald Knuth gives the rule as follows:

A square is numbered if it is a white square and either (a) the square below it is white and there is no white square immediately above, or (b) there is no white square immediately to its left and the square to its right is white.

Notice how Knuth carefully yet a bit awkwardly avoids saying 'black square above' and 'black square to the left', and instead says 'no white square'. I think this is because of the white squares in the top row and leftmost column. What is above or to the left of them? Shades of Jim Brown's empty array jokes! How can you tell whether what is above a white square in the top row is not a white square?

The problem is solved by shifting all of the one-square rook move squares to coincide with that square. For the border squares, this ensures that zeros, signifying 'black' squares, are shifted to coincidence. This is done in two steps, producing two matrices of the same size as the puzzle diagram matrix.

The verbs to solve the crossword numbering problem are:

```
f=: soa < sob
X=: ] *. f +. f"1
```

The use of `f"1` produces a one for each atom in the matrix in which the square to its left is less than the square to its right. This is true only if the square to the left is black (0) and the square to the right is white (1). This identifies the potential squares where an across word can begin.

The following use of the verb `f` produces a one for each atom in the matrix in which the square above it is less than the square below it. This is true only if the square above is black and the square below is white. This identifies the potential squares where a down word can begin.

The verb XORs these two results, producing a matrix showing all squares satisfying either of these tests (the same square can be 1 in both tests). This combined square is ANDed with the original square, yielding a result which has a 1 only where there is also a 1 in the orig-

inal, thus definitely identifying squares to be numbered as the beginning of either an across or a down word.

These steps are summarized as follows:

```

      M      ;    (f M)      ;    (f"1 M)      ; (]*.f+.f"1)M
+-----+-----+-----+-----+
|0 1 1 1 1 0|1 1 0 1 1 1|1 1 0 0 0 0|0 1 0 1 1 0|
|1 1 0 1 1 1|1 0 0 0 0 1|1 0 0 1 0 0|1 0 0 1 0 1|
|1 1 1 1 0 1|0 0 1 0 0 0|1 0 0 0 0 0|1 0 1 0 0 0|
|1 0 1 1 1 1|0 0 0 0 1 0|0 0 1 0 0 0|0 0 1 0 1 0|
|1 1 1 0 1 1|0 1 0 0 0 0|1 0 0 0 1 0|1 1 0 0 1 0|
|0 1 1 1 1 0|0 0 0 0 0 0|1 1 0 0 0 0|0 1 0 0 0 0|
+-----+-----+-----+-----+

```

The verb `X` encapsulates these steps. We obtain the final result by applying `X` to `M`:

```

] c=: X M
0 1 0 1 1 0
1 0 0 1 0 1
1 0 1 0 0 0
0 0 1 0 1 0
1 1 0 0 1 0
0 1 0 0 0 0

```

Knuth's exercise asks for a display of the puzzle using black and white squares made of plusses, minuses, and stiles. To complete the exercise we give with no further ado ...

```

c1b=: [: +/ [: *./\ ' ' " _ = ]  ➡
NB. count leading blanks

CWD=: monad define
NB. y is boolean matrix giving a crossword puzzle
NB. pattern where 0 represents a black square
NB. and 1 a white square;
NB. yields crude numbered crossword puzzle diagram.
bw =. <' ' +', ' ' +', ':' +++++' ➡
NB. scalar white square
bb =. <'+++++', '+++++', ':' +++++' ➡
NB. scalar black square
g =. [: , [: ,. / [: > ] ➡
NB. ravel stitch insert open
a =. , X y ➡
NB. mark across and down squares
b =. (c1b |.)"1 ":",.>:i.+a ➡

```

```
NB. format & left adjust numbers
  c =. <"2 b(<0;0 1)}"1 2>bw
NB. insert numbers in blanks
  d =. cws0(bw;((,y)#a);<c)
NB. insert blank white squares
  e =. cws0(bb;((,y);<d)
NB. insert black squares
  f =. g"1 ($y)$e
NB. stitch open of reshape
  '+' ,. '+' , (3 5*$y)$, f
NB. reshape and border
)
```

When this is applied to M we get:

```
CWD M
+++++
+++++1  +  +2  +3  +++++
+++++  +  +  +  +++++
+++++
+4  +  ++++++5  +  +6  +
+  +  +++++  +  +  +
+++++
+7  +  +8  +  +++++  +
+  +  +  +  +++++  +
+++++
+  ++++++9  +  +10  +  +
+  +++++  +  +  +  +
+++++
+11 +12  +  ++++++13  +  +
+  +  +  +++++  +  +
+++++
+++++14  +  +  +  +++++
+++++  +  +  +  +++++
+++++
```

Conway’s Game of Life

In Knuth’s *The MetaFont Book*, Addison Wesley, 1986, Exercise 13.24, p. 121, and Answer 13.24, p. 245 are as follows:

Exercise 13.24: In John Conway’s “Game of Life” pixels are said to be either *alive* or *dead*. Each pixel is in contact with eight neighbors. The live pixels in the (n+1)st generation are those who were dead and had exactly three neighbors in the nth generation, or those who were alive and had exactly two or three live neighbors in the nth generation. Write a short MetaFont program that displays successive generations on your screen.

Answer 13.24: (We assume that `currentpicture` initially has some configuration in which all pixel values are zero or one; one means “alive.”)

```
picture v; def c = currentpicture enddef;
forever: v := c; showit;
  addto c also c shifted left + c shifted right;
  addto c also c shifted up + c shifted down;
  add to c also c - v; cull c keeping (5 , 7);
endfor.
```

(It is wise not to waste too much computer time watching this program.)

It is not apparent, but Knuth’s approach assumes a flat universe, where, as is customary, if things are pushed beyond the edges, they fall off the surface. Many APL approaches, using the rotate verb, assume a toroidal universe, where the display is assumed to connect both horizontally and vertically, and if things are pushed beyond the edges, they wrap around, staying on the surface, and appearing on the opposite edge.

Assuming that you don’t speak MetaFont, I’ll explain the logic behind this program. It encodes the values of the pixel and its eight neighbours to indicate a pixel which is to be alive or dead in the next generation. The encoding gives a weight of one to the pixel and a weight of two to each of its neighbours. If we compute the sum of the pixel plus twice the sum of its neighbours, we can arrive at any of the eighteen values from zero through seventeen: zero for a dead pixel or one for a live pixel, plus 0, 2, 4, 6, 8, 10, 12, 14, or 16 for twice the sum of its neighbours, depending on whether the number of alive neighbours is 0, 1, 2, 3, 4, 5, 6, 7, or 8. Then “dead and exactly three live neighbours” is $(0+(2*3))$, or 6, and “alive and exactly two or three live neighbours” is $(1+(2*2))$, or 5, and $(1+(2*3))$ or 7, so we get the next generation by replacing each sum equal to 5, 6, or 7 by one, and all others by zero.

Knuth forms a new picture as the sum of the picture and the picture shifted one column left and the picture shifted one column right.

He adds this new picture to the new picture shifted down one and the new picture shifted up one, next doubles the value of each pixel, then subtracts the original pixel, giving us the value we desire. It only

remains to see whether this is one of the life-giving values. That's what the cull verb does: it culls the chaff from the result, leaving only those having the value 5 or 6 or 7.

J verbs to do what Knuth's MetaFont program does are:

```
g=: ] + soa + sob
h=: ] -- [: +: [: g g"1
L=: h e. 5 6 7"_
```

The verb `g` will be used in somewhat the same way as the verb `f` in the crossword puzzle problem. It is used first (`g"1`) to form the sum of each column with the columns on either side, then applies `g` to this, forming the sum of each row with the rows on either side, doubles (`+:`) this, then subtracts (`] --`) the original picture. The verb `h` encapsulates this. The verb `L` uses `h` to go through these steps, then determines which atoms of the result are equal to either 5 or 6 or 7.

Given a picture:

```
picture=: 0 0 0 0 0,0 1 1 1 0,0 0 1 0 0, ➡
0 1 1 0 0,:0 0 0 0 0
```

Its original value and the next 5 steps of Life are:

```
q=: L^(i.6) picture
<"2 q
```

0 0 0 0 0	0 0 1 0 0	0 1 1 1 0	0 1 0 1 0	0 0 1 0 0	0 0 0 1 0
0 1 1 1 0	0 1 1 1 0	0 1 1 1 0	0 1 0 0 1	0 0 0 1 1	0 0 0 1 0
0 0 1 0 0	0 0 0 0 0	0 0 0 1 0	0 0 0 1 0	0 0 0 0 0	0 0 0 0 0
0 1 1 0 0	0 1 1 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0

Endnote (Ed.)

Alternatively, to see an extremely crude animation of Conway's Game of Life in the J session, insert this line into a new ijs window (Ctrl+N):

```
] picture=: L picture
```

and press Ctrl+Shift+W repeatedly to run it.

20 New Model Computer

First published in Vector, 15, 4, (April 1999), 106-112.

Revised by Oleg Kobchenko, (April, 2009).

Donald Knuth began writing his series called *The Art of Computer Programming* in 1962, and devised a mythical computer called MIX with which to illustrate computer language programs, and as a machine to be used for exercises. Half a lifetime has gone by since then — half a human lifetime, three or four computer lifetimes. His MIX design is now obsolete. A new machine, called MMIX, will replace it in subsequent revisions of his volumes. A preliminary writeup of this new machine is given in Knuth's web site.

His home page address is <http://sunburn.stanford.edu/~knuth>

Briefly, MMIX operates on 64-bit words. It has 256 general-purpose 64-bit registers that can hold either fixed point or floating point numbers, or characters, or arbitrary binary data. Most instructions have the form OP X Y Z, where OP, X, Y, and Z are each 8-bit bytes. If OP is the ADD instruction, for example, the meaning is "X=Y+Z", that is, "Set register X to the contents of register Y plus the contents of register Z." There are 256 op codes that fall into a dozen or so categories. The virtual memory available to an application is 2^{64} bytes. There are no input-output instructions; files are handled as memory-mapped data. There is no operating system at the moment for MMIX. Knuth writes, "Whenever I have been asked if I will be writing a book about operating systems, my reply has always been 'Nix.' Therefore the name of MMIX's operating system, NNIX, should come as no surprise."

The purpose of this column is to discuss a few J-related aspects of this machine. I am attempting just now to write a J model of it. Time will tell if and how well I do this.

On this machine numbers are 64 bits long, though the design tolerates shorter numbers. For floating point numbers it uses the IEEE/ANSI standard 64-bit form, and accommodates the 32-bit forms only to transform them to the 64-bit form in computations. More interestingly, integer numbers are also 64 bits long. Thus

an unsigned integer can take on any value from zero through 18,446,744,073,709,551,615, or approximately $1.8e19$. Signed integers range between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. One of the reasons I chose J as simulation language is its extended integers, which allow me to use these large numbers directly, with the least amount of fuss: an occasional terminal 'x' on a number, or 'x:' preceding a number.

Many years ago I heard Luther Woodrum, the one who gave APL a good grade (he designed and implemented the grade primitives in APL\360) ask, "When will computer designers put in a maximum instruction?" In almost all computers, finding the maximum of two numbers involves a branch instruction. Something like this:

```
condition is x is less than y
branch to Label if condition is true
maximum is x
branch to exit
Label: maximum is y
exit:
```

One of the great motivations of computer designers is to reduce the number of branches required by programs executing at the machine level. Branches really slow things down. I recently attended, along with Larry Breed (the key designer and implemetor of APL\360), a talk Knuth gave on MMIX. I had read the preliminary MMIX writeup and hadn't found a maximum instruction, so after the lecture, in the question period, I asked how one found a maximum on MMIX. Knuth fumbled a bit, and gave an explanation which I didn't hear very well. Larry, however seemed to understand and accept it, so I didn't pursue the issue then, but the next day I phoned Larry and asked him for more. He explained that there was an identity that could be used to obtain maximum at the machine level without a branch, namely:

$$a \max b \text{ is } b + 0 \max a - b$$

I said to Larry that that looked a bit circular, didn't it, defining maximum in terms of itself? Larry assured me that the circularity is only apparent. He said also that, before retiring from IBM a few years ago, he had been instrumental in getting the architects of an IBM computer (I think the RS6000) to add a "difference or zero" instruction, primarily to handle maximum, and that now Knuth

had devised a similar feature for MMIX. It's called 'zero or set if negative', written `ZSNI rX, rY, Z`, which acts as follows: If register Y is negative, register X is set to Z, otherwise nothing happens. To see how this works in finding maximum, look at the three cases below. The two values are in A and B and the result is to appear in B.

		case 1		case 2		case 3		
		A=8	B=3	A=3	B=8	A=3	B=3	
SUB	A, A, B	5	3	-5	8	0	3	A=A-B
ZSNI	A, A, 0	5	3	0	8	0	3	A=0 max A
ADD	B, A, B	5	8	0	8	0	3	B=A+B

Somewhere below the machine instruction level there is a branch, but the machine is happy nevertheless, because the three instructions above execute very rapidly, and in a pipelined machine the pipeline is undisturbed, since no instruction-level branch is needed.

The next thing I'd like to mention is exemplified by a frustrating aspect of IEEE floating point as implemented by two different systems: the little-endian and the big-endian school. In the little-endian implementations the bytes are numbered from left to right, 7 through 0, and the bytes in the big-endian school are numbered 0 through 7. This leads to strange things like, in J, looking at the value of a floating point number on a Macintosh and on an Intel machine gives two different pictures, one reversed from the other. Another bit of explanation I need to give you is that Knuth uses the prefix # in front of a number expressed in hexadecimal. One of the 'bit fiddling' instructions is 'multiple or', denoted by MOR. The explanation given for it is:

Suppose the 64 bits of registers Y and Z are indexed by pairs such that the bytes are numbered from 0 to 7 and the bits within the bytes also by 0 through 7; then the MOR operation replaces bit ij of register X by the bit

$$y_0z_{i0} \text{ or } y_1z_{i1} \text{ or } \dots \text{ or } y_7z_{7i}$$

Thus, for example, if register Z contains #0102040810204080, MOR reverses the order of the bytes in register Y, converting between little-endian and big-endian addressing.

What Knuth has designed, if the above is opaque to you, is a special case of the good old-fashioned 'or dot and' inner product. When I came to describing this instruction in my simulation, in addition to standard housekeeping, all I had to write was:

$$U =: B +. / . *. A$$

where B and A are 8-by-8 boolean matrices representing the 64 bits in the two operands.

If I get anywhere with this simulation, I'll probably report on it in a future column.

Endnote (*Ed.*)

For a sample A and B to use with the above expression, try:

```
A =: ? 8 8 $ 2
B =: ? 8 8 $ 2
```

21 New Big Deal

*First published in Vector, 16, 1, (July 1999), 113-119.**

Revised by Kip Murray, (April, 2009).

Chris Burke was displeased, to begin with. He had tried J's deal with a small left argument and a very large right argument, and this took a perceptibly long time. He tried it with a larger argument and was told he had run out of space. He tried it with a still larger argument, and was told he had exceeded deal's limits. He wrote to Roger Hui about these distressing circumstances, showing him several examples:

```
(time=:6!:2) '1 ? 1e7' NB. this takes too long
0.93
```

```
1 ? 1e8
|out of memory
| 1 ?1000000000
```

```
1 ? 1e9
||limit error
| 1 ?10000000000
```

Roger's first thought was how to get around this limitation. He said that if the left argument is much smaller than the right Chris would be better off just doing $x\$y$; because deal allocates a boolean vector of length y to compute unique answers in the result of $x?y$. Thus a very large y would give the results Chris noted. He forwarded Chris's message to me, copying Chris, with the message "Perhaps Eugene can comment."

I checked the literature on my shelf but couldn't find any worthwhile "selection without replacement" algorithms. I then remembered the very early days of APL\360, before deal was made a primitive—some time in 1966, perhaps. I had written a defined function to perform deal, using an algorithm that was quite fast. I remembered it vaguely, and thought that with it I might be able to do better with J. I wrote one, tried it out on a dozen or so cases, then communicated it to Chris and Roger:

* The original attribution was as follows: "by Eugene McDonnell, with a major contribution from Roger Hui."

```

deal=: dyad define
NB. experimental deal for small x
NB. and very large y
count=: 0
t=. i.0
k=. 1.1 NB. adjust as you see fit
NB. maybe make it a function of y
u=. >. k * x
whilst. (# t) < x do.
t=. ~. ? u # y
count=: count + 1
end.
x {. t
)

```

As you can see, I was uncertain about the factor 1.1. I had put a counter in to see how often more than one execution of the whilst section was needed. In the few dozen cases I tried, there were none. Happily, the timings showed a large improvement over current deal for Chris's cases: the cases that were slow were much faster, and the range of the right argument was significantly extended. Here are the timings I experienced:

```

time '1 ? 1e7'
4.07
100 time '1 deal 1e7'
0
1000 time '1 deal 1e7'
0.00044
1000 time '1 deal 1e8'
0.00044
1000 time '1 deal 1e9'
0.00044
1000 time '100 deal 1e7'
0.00082
count
1
1000 time '100 deal 1e8'
0.00076
count
1
1000 time '100 deal 1e9'
0.00083
count
1
ts=: 6!:2 , 7!:2@] NB. time and space
100 ts '1 deal 1e7'
0 2240

```

```

      100 ts '1 deal 1e8'
0 2240
      100 ts '1 deal 1e9'
0.0005 2240
      100 ts '100 deal 1e7'
0.0005 4288
      100 ts '100 deal 1e8'
0.0005 4288
      100 ts '100 deal 1e9'
0.0005 4288

```

Roger thought this was neat. He implemented my algorithm, invoked when $x < 0.01 * y$. He rewrote my algorithm, simplifying it:

```

deal=: dyad define
u=. >. 1.1 * x
while. x># t=. ~. ? u # y do. end.
x {. t
)

```

His C implementation looked like this:

```

static A bigdeal(m,n)I m,n;{A t,x,y;
  RZ(x=sc((I)floor(1.11*m)));
  RZ(y=sc(n));
  do{RZ(t=nub(roll(reshape(x,y))));}while(m>AN(t));
  R vec(INT,m,AV(t));
}
/* E.E. McDonnell circa 1966, small m and large n */

```

But he worried that this would run into the birthday problem, which gets its name from its most celebrated instance, that the odds are in your favour if you bet that in a group of 23 or more people at least two of them will have the same birthday. The greater the number of people, of course, the higher the probability that this will occur. With more than 365 people, the probability is certain, or 1, since the pigeonhole principle dictates that if you have x pigeonholes and y pigeons, with x less than y , at least one of the pigeonholes will have more than one pigeon in it. What Roger wanted to know was, what is the value of x as a function of y where the probability of a duplicate appearing was 0.5. If he had this information, he could make the multiplier more accurate. Perhaps 1.11 wasn't good enough.

I wrote the following:

```

Hui_test=: dyad define
tests=: 0
successes=: 0
whilst. 1000 > tests =. >: tests do.
successes=: successes + *./~:?x$y
end.
<. 100 * 0.001 * successes
)

```

I ran this test^{*} for y in all of the hundreds, thousands, ten thousands, hundred thousands, millions, and 10,000,000 and 20,000,000. To make it easier to digest I'll only show the results for $y=10^2$ 3 4 5 6 7. The other results are consistent with these.

y	x
100	12
1000	37
10000	116
100000	370
1000000	1180
10000000	3740

I thought this looked roughly like a square root relationship so tried a few manoeuvres:

```

%:y
10 31.6228 100 316.228 1000 3162.28
%:1.4*y
11.8322 37.4166 118.322 374.166 1183.22 3741.66

```

This was quite a good fit, and shows:

```

y = (x^2) % 1.4
x = %: 1.4 * y

```

to a close approximation. I communicated these results to Roger. He studied this and was able to apply some more theory to it, and wrote back to me: in choosing m items from a universe of n items,

^{*} i.e. Hui_test was run with each given value of y (right argument) and trial values of x (left argument) until it returned 50 or close. Thus for $y=100$

```

11 Hui_test 100
52
12 Hui_test 100
51
13 Hui_test 100
45

```

So $x=12$ gives the closest to 50 (percent) and is the value to be used. (Ed.)

the probability of all the m items distinct is $(*/n-i.m) \% (n^m)$.
 The numerator is the number of ways of choosing m distinct items;
 the denominator is the number of ways of choosing m items. Thus:

```
(*/n-i.m) \% (n^m)                                (a)
(*/n-i.m) \% (* / m $ n)
*/ (n-i.m) \% (m $ n)
*/ (n-i.m) \% n
f=: [: */ ] %~ i.@[ -- ]

22 23 f"0 ] 365    NB. the classic birthday problem
0.524305 0.492703
```

The first m , for which $m f n$ is less than 0.5, is:

```
1 + (0.5 > */ \ n %~ n - i.n) i. 1
f1=: >: @ (i.&1) @ (0.5&>) @ (* \) @ ([ %~ ➡
[ - i.)
f1 365
23
n=: , (,10^1 2 3 4 5)*/1 2 4 8
m=: f1"0 n
n,.m ,. %: 1.4*n
10      5 3.74166
20      6 5.2915
40      8 7.48331
80     11 10.583
100     13 11.8322
200     17 16.7332
400     24 23.6643
800     34 33.4664
1000    38 37.4166
2000    53 52.915
4000    75 74.8331
8000   106 105.83
10000  119 118.322
20000  167 167.332
40000  236 236.643
80000  334 334.664
100000 373 374.166
200000 527 529.15
400000 745 748.331
800000 1054 1058.3
```

For extremely large values of n , Roger saw that the function `f1` is impractical to compute, and he concluded that a method based on

root finding on the original function f , using my approximation of $m =: \%: 1.4 * n$ as an initial guess, would be more suitable.

```
1e7 3724
1e8 11775
1e9 37234
2e9 52656
```

The end result was that Roger found that the rule I had originally suggested of switching to the “roll” algorithm for $m > n$ when $m < 0.01 * n$ does run into the birthday problem, and he replaced it with the more accurate rule I had found following his suggestion. There the matter rests, with one small postscript. As I studied Roger’s mathematics, particularly the phrase $(*/n - i . m)$ in line (a) above, I recalled J’s “falling factorial” function. The *J Dictionary* defines this as follows:

The fit conjunction applies to \wedge to yield a stope defined as follows:
 $x \wedge ! . k \ n$ is $*/x + k * i . \ n$. In particular, $\wedge ! . _1$ is the falling factorial function.

Let me elaborate on this. Think of the caret (\wedge) as being defined in the first instance as denoting a function of three arguments: a base, a count, and a step. Then $\text{caret}(\text{base}, \text{count}, \text{step})$ is the product over $\text{base} + \text{step} * \text{integers count}$. For example,

```
'base count step'=: 3 5 7
i. count
0 1 2 3 4
step * i. count
0 7 14 21 28
base + step * i. count
3 10 17 24 31
*/ base + step * i. count
379440
```

```
caret=: monad define
NB. general caret function
NB. y is a list of three values:
NB. base
NB. count
NB. step
'base count step' =. y
*/ base + step * i. count
)
caret(3 5 7)
379440
```


This generality hides the fact that there are really just three variations of significant interest, steps having values `_1`, `0`, and `1`. These three provide falling factorials, steady factorials (powers) and rising factorials. Each of these has to do with a product over count number of values, beginning with base, and continuing with values increasing by step. See what results come with a base of 5 and a count of 3, with each of the three significant step sizes:

```

    caret 5 3 _1
60
    5 * 4 * 3
60
    caret 5 3 0
125
    5 * 5 * 5
125
    5 ^ 3
125
    caret 5 3 1
210
    5 * 6 * 7
210

```

The case with step zero is the default case of `caret`, and is the power function. We can use the falling and steady factorial cases to write a more compact version of Hui's function `f`:

```

probdupes=: dyad define
NB. Probability of no duplicates when drawing with
NB. replacement from among the 1st count integers
    base =. x
    count =. y
    (base ^!._1 count) % (base ^!.0 count)
)

NB. simplified version of probdupes
pd=: ^!._1 % ^

NB. version exploiting family resemblances
pdx=: [: %/ ^!._1 0

    365 probdupes 23
0.492703
    365 pd 23
0.492703
    365 pdx 23
0.492703

```

These functions fail when the values of the falling factorial get very large, causing its value to exceed the maximum real value, and thus to be represented by machine infinity. When this happens, the steady factorial (power) value will already be machine infinity, and the quotient of two infinities is *Not a Number*, or NaN. To get around this problem, use extended integers as arguments, and extended precision inverse on the result. When the result is smaller than the smallest nonzero number, a result of zero* is forced:

```

      365 pd 200          NB. result is NaN error, from _ % _
|NaN error: pd
|  365      pd 200

      x:^:_1 [ 365x pd 200x    NB. result of zero is forced
0

```

* With J602 running on an iMac (2009) the value returned is `1.61013e_30` not 0, i.e. nearly zero but not quite. (*Ed.*)

22 We'll Cross That Bridge When We Come To It

First published in Vector, 16, 3, (January 2000), 126-130.

Revised by Gilles Kerouac, (April 2009).

Björn Helgason, from Iceland, submitted this problem to the Internet J Forum:

Four people, A, B, C, and D, come to a bridge at night, with only a flashlight to guide them. The time each one takes to cross the bridge is: A in 1 minute, B in 2 minutes, C in 5 minutes, and D in 10 minutes. The bridge will only support two of them at a time, and the time to cross is, of course, that of the slower walker. The flashlight must be carried on any crossing. They want to get across the bridge as quickly as possible. Since they have a palm-top computer with J installed, they write a program that tells them what the minimum time is, and in what order they should go forth and back over the bridge. Your problem is to write an equivalent program. To help you get started, the program found that the minimum time is seventeen minutes.

The first response to Björn's problem said that there was no need for a program to be written, because it could be solved in one's head; and complained, furthermore, that seventeen minutes was impossible; it was demonstrable that the minimum time was nineteen minutes: the fastest one, A, going across first with B and the flashlight, leaving B on the other side, coming back with the flashlight to go across with C, then coming back with the flashlight again to go across with D: two plus one plus five plus one plus ten, making nineteen, and that the order was unimportant so long as A was the constant companion. It couldn't be done more quickly, since the fastest man was always the one to go back. However, Helgason kept insisting that seventeen was the minimum, and offered to send a private message to the complainer giving such a solution.

After a fair amount of back and forth between Helgason and the disbeliever, the issue was resolved when Kirk Iverson finally submitted a solution from Toronto which gave the seventeen-minute solution for the problem. He admitted that he wasn't sure whether the program would give the minimum answer for all

possible combinations of number of people per crossing and times for each to cross, but it would handle many cases. (Yes, Kirk is related to Ken; he's a nephew.) The disbeliever was converted, of course, and made a handsome apology.

See whether you can arrive at a solution, either to this specific problem, or to the more general one solved by Kirk. When you're satisfied that you either can solve the problem, or have thrown up your hands in frustration, or else are just too lazy to give it any more thought, you can go on to read Kirk Iverson's solution. From here on, it is his text, slightly amended, with additional comments by me shown in square brackets.

First, the "compiled" code, in case anyone wants to run this without seeing what it does:

```
NB. ---- copy into ijs window and execute -----
".(noun define)-.LF
bridge:(<./@(>@{2&{}}){(.@),(1&{@],&.><@[]),(2&{@]-.&.><@[]),3&{
.@],<@[]})^:(*#@(>@{1&{}}))@((((>@{.@[-. ]);>@{:@[]}).@([[-.[-.
]])/))@(>@{0&{}}{([<.#@])}.)]"1(, : | .)@(/:~)@(>@{1&{}})([>@{~(>@{
2&{}}@]<&(<./)>@{:@[]]+.#@(>@{:@[]]=#@(>@{1&{}}@[])])(. @],(1&{@]-
.&.><@[]),(2&{@],&.><@[]),3&{.@],<@[]])^:(*#@(>@{1&{}}))^:_.(( {
.;};.'_'_) :.((+/@:(>./@>);)]@{3&{.}))@,
)
NB. -----
```

[I speculate that Kirk first wrote a series of verbs to accomplish the objective, then obtained this unreadable mess by applying the fix adverb (*f.*) to the main verb. Doing this replaced all the subordinate verbs by their definitions, yielding the aforesaid mess. He probably then obtained the character string corresponding to the fixed function, using the *5! : 5* form of the foreign conjunction, and displayed this in a field 62 characters wide, giving the six lines shown above, and copied them.]

The line following the first *NB.* line contains three items that are defined in a script loaded when a *J* session starts: the variable *noun* has the value 0; the adverb *define* has the value *: 0* (explicit definition script form) and the variable *LF* is the linefeed character, defined as *0{a..* The *(noun define)* in the first line after the comment line (beginning with *NB.*) permits entry of multiple lines of text. Kirk pasted in the six lines, then typed a right parenthesis on the next line, and hit enter, thus causing entry to terminate, and the *(noun define)* was replaced by the six lines shown. The *-.LF* removed the linefeeds from this text,

and the ". " executed the line, causing a verb named `bridge` to be defined.

I copied the `bridge` verb, and pasted it into a J session, and it worked as advertised. The left argument to `bridge` is the maximum number of people who can walk on the bridge at the same time; the right argument is a list of the length of time each person needs to cross. The result is a boxed list of total elapsed time, followed by all moves:

```
2 bridge 1 2 4
+-+-----+-+-----+
|7|1 2|1|4 1|
+-+-----+-+-----+
```

The result signifies 7 minutes total time; 1 and 2 cross; 1 returns; 4 and 1 cross.

This is your last chance to try figuring out how to solve our problem, because the solution Kirk obtained is now going to be shown:

```
NB. -----
NB. Crossing the bridge.

NB. Rules
NB. Move all people from one side of bridge to other.
NB. Each person has an individual time it takes to cross.
NB. At most MAXLOAD number of people on bridge.
NB. Torch must always be with a group on the bridge.
NB. Speed of a group is the speed of the slowest member.
NB. Strategy
NB. Overall strategy is to pick a good group to go over, and
NB. then have the fastest person to return with the torch. Wash,
NB. rinse, repeat. I call the guy to return the torch the "runner".
NB.
NB. We attempt to move the slowest people together as a group,
NB. rather than split them up to slow down all groups. We move
NB. the slowest ones over whenever there is a suitable runner
NB. to return, i.e., none of this slow group will have to return.
NB.
NB. If there is no good runner, we select the fastest to go over
NB. and act as runners. We only include enough runners which are
NB. necessary to bring in the rest of the people.
NB. Notes
NB. Iteration is done by repeatedly applying a function to
NB. the full set of data until it results in everyone moved.
NB. Data is maxload;unmoved;moved;move0;move1;move2;...
NB. Verbs to access pieces from the data

maxload=: >@(0&{) NB. maximum number of people on the bridge
unmoved=: >@(1&{) NB. people not yet moved
moved=: >@(2&{) NB. people already moved
moves=: 3&}. NB. record of all moves
more=: *@#unmoved NB. are there more to move?
NB. move/unmove move people across bridge
NB. x is list of people to move; y is data
move=: {.@] , (1&{@] less each <@[] , (2&{@] , each <@[] , 3&{@] , <@[]
unmove=: {.@] , (1&{@] , each <@[] , (2&{@] less each <@[] , 3&{@] , <@[]
```

NB. Strict <less> Similar to -. but removes from x only the number
 NB. of matching items found in y All items in y are expected to be
 NB. found in x, and the items in the result are in the same order as
 NB. they appear in x Universe is ~.x, and is catenated in front of
 NB. y, therefore count (#/.~) of each returns the counts of the
 NB. respective items. The difference (incremented to compensate for
 NB. the catenation of the universe onto y) is used to copy items from
 NB. the universe.

```
less=: universe #- [ >:@-&count universe , ]
universe=: ~.@[
count=: #/.~
```

NB. Pick the group to go across. Runners are the fast group to
 NB. shuttle the rest over; waddlers are the slower group put together
 NB. to capitalize on the slowest of the bunch.

```
groups=: sort@unmoved (runners ; waddlers) ]
runners=: [ maxtake~ required <. maxload@]
required=: 1 >:@>. #@[ <:@>.% <:@maxload@]
waddlers=: maxload@] maxtake |.@[
```

```
pickslow=: nogoodrunner +. lastgroup NB. 0-runners; 1-waddlers
nogoodrunner=: moved@] <:&fastest slower
lastgroup=: #@slower = #@unmoved@]
```

NB. fastest/slowest in a group

```
fastest=: <./
slowest=: >./
slower=: >@{:@[
```

```
maxtake=: ([ <. #@]) {. ] NB. take, but don't overtake
```

```
pick=: groups (pickslow >@{ [ ] ]
```

NB. forward - pick group to go and move them

```
NB. return - pick fastest runner and move back, if more to move
forward=: pick move ]
```

```
return=: (fastest@moved unmoved)^:more
```

NB. iterate repeats the forward/return action until there are no more.

```
iterate=: return@forward^:more^:_
```

NB. assemble the argument into the data structure; inverse

NB. computes total trip time from moves, and returns that along
 NB. with the moves.

```
assemble=: ({. ; }. ; '""_) :. ((+/@:(slowest@>) ; ])@moves)
```

NB. maxload play speeds

```
bridge=: (iterate&.assemble)@,
```

NB. -----

And now for our problem:

```
2 bridge 1 2 5 10
+---+---+---+---+---+
|17|1 2|1|10 5|2|2 1|
+---+---+---+---+---+
```

The total time is 2+1+10+2+2, or 17, as was required. An alternative would have 2 return the first time, and 1 the second time, and this also is minimum.

Kirk's J implementation consists of two dozen or so verbs, all in tacit form, so this is a functional programming solution. The verb `iterate` continues until there is no more to be done, using power to the limit, denoted by infinity (`^:_`).

As this article goes to press, it is recognized that Kirk's solution is not guaranteed to give the minimum result for all cases; for example, it fails for $1\ 10\ 11\ 12$; but it was the first solution that gave the minimum result for the given case $1\ 2\ 5\ 10$.

23 An Open and Shut Case

First published in Vector, 16, 4, (April 2000), 99-106.

Revised by Ric Sherlock, (April 2009).

My 11-year old granddaughter, Amy Powers, brought home a problem the other day which her mathematics teacher had given to her 6th-year class. I thought it pedagogically fruitful, and advanced beyond any mathematics I had learned at her age. It demonstrates how much teaching has changed for the better in the more than sixty years since I was in the 6th year. A broader variety of subject matter beyond the elementary algebra I was taught and a better method of teaching mathematical topics have made the subject more interesting and, I am convinced, easier to learn.

Some background: in many schools in the United States, lockers are provided in which the students may store their textbooks and other belongings. These are usually placed in long rows, and some inspired teacher was inspired to create the *Locker Problem* to give a concrete base on which to help teach some facts about divisors.

The Locker Problem

In a high school (years 9 to 12) there are 1,000 lockers placed next to each other along a hall. During winter break, the custodians at the school clean the lockers and paint fresh numbers on each locker door. The lockers are numbered from 1 to 1,000. When the 1,000 students arrive back from their vacation, they decide to celebrate returning to school by working off some energy. Here is what they do: Student 1 runs down the row of lockers and inverts every door. (To invert a door means to open it if it was closed, and to close it if it was open.) Student 2 runs down the row of lockers and inverts doors 2, 4, 6, 8, and so on to the end of the line. Next student 3 inverts the doors of lockers 3, 6, 9, 12, and so on to the end of the line. This pattern continues until each of the 1,000 students has had a turn to run down the hall.

When the students are finished, which doors are open?

Amy set about answering this question by making a table:

											1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2		C		C		C		C		C		C		C		C	
3			C			0			C			0			C		
4				0				0			C					0	
5					C				0					0			
6						C				0							
7							C							0			
8								C								C	
9									0								
10										C							
11											C						
12												C					
13													C				
14														C			
15															C		
16																0	
17																	C

The first row of all 0s shows that after student 1 has finished, all the doors are open. The next row, for student 2, shows that the even numbered doors are closed. The third row, for student 3, shows that every third door has changed its state: if it had been open, it is now closed, and if closed, is now open. I've added row and column stubs, and have darkened the 0s along the diagonal, and the corresponding row and column numbers. Because doors 1, 4, 9, and 16 were open, Amy guessed that the doors that were open were those of students whose numbers were squares. Was she right?

The next step might be difficult for a sixth-grade student to have arrived at. It is to count the number of students who inverted each door. Each of the 0s and Cs in Amy's table is a divisor of the number it lies beneath. For example, the 0s and Cs under 12 are inversions by the six students numbered 1, 2, 3, 4, 6, and 12; thus, the number of divisors of 12 is six. The number of divisors of a number varies irregularly. Counting the divisors of our seventeen numbers gives the table:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2

At first glance, there doesn't seem to be any pattern to the number of divisors, except that there is a tendency for the number to incr-

ease; however, I've put in bold type those integers and their divisor counts where the count is odd, since only those doors will be open that have been inverted by an odd number of students. In Amy's chart the doors that are open are numbers 1, 4, 9, and 16. Her conjecture was that the doors that are open are those whose numbers are squares. I made the further conjecture that squares and only squares have an odd number of divisors. Both conjectures are true. If a number is a square, it has an odd number of divisors; conversely, if a number has an odd number of divisors, it is a square. How can we show this?

Just studying the number of divisors doesn't reveal any pattern. I shall make a great leap here, and go immediately to a description of the Prime Factors Exponent Numbers (PFENs). Kenneth Iversen's book *Algebra, an Algorithmic Treatment* (Addison-Wesley, Menlo Park, California, 1972), gives a description of this system in section 16.2. In this number system a positive integer is represented by a list of non-negative integers, where the integer in column i represents the exponent to which prime i is to be raised, going from left to right. The primes corresponding to indices 0 1 2 3 4 5 are 2 3 5 7 11 13. For example, if, in the PFEN for a number, column 3 has the value 5, the third prime, 7, is to be raised to the 5th power, and so 0 0 0 5 represents 16807 ; the decimal integer corresponding to the PFEN is the product of the results of raising each prime to the corresponding power. The PFEN forms for the first 17 positive integers are:

n	PFEN n
1	
2	1
3	0 1
4	2
5	0 0 1
6	1 1
7	0 0 0 1
8	3
9	0 2
10	1 0 1
11	0 0 0 0 1
12	2 1
13	0 0 0 0 0 1
14	1 0 0 1
15	0 1 1
16	4
17	0 0 0 0 0 0 1

The PFEN for 12 is 2 1, signifying that the decimal number corresponding can be obtained by $\ast/2 \ 3^2 \ 1$, that is, $\ast/4 \ 3$, or 12. The PFEN for 1 is the empty list, since 1 has no prime components; its value is the product over the empty list, or 1. A J verb for producing the PFEN corresponding to a decimal number is given by:

```
pfd=: _&q:      NB. PFEN from decimal
pfd 300
2 1 2
pfd 16807
0 0 0 5
```

The first example shows a number having prime factors 2, 3, and 5 with exponents 2, 1, and 2. The second example shows a number having prime factors 2, 3, 5, and 7 with exponents 0, 0, 0, and 5. Since any integer to the zero power is 1, any number of zero exponents do not alter the value of the product.

I have to be reminded from time to time that J has built-in inverses for a great many verbs; I was plodding through defining *dfp*, the verb inverse to *pfd*, the hard way late one night, and woke up the next morning to the realization that all I needed was the inverse adverb (^:_1).

```
dfp=: pfd^:_1    NB. decimal from PFEN
dfp 2 1 2
300
dfp 0 0 0 5
16807
```

There's no need for the user to know how an inverse works—it is enough merely to accept the presence of an inverse as a blessing. If you do want to know, however, you can see what the inverse looks like in detail by using the basic conjunction *b.* as follows:

```
pfd b. _1
(p:@i.@# */ . ^ )"1 :.(_&q:)
```

Studying this shows that the inverse function works by taking the inner product, with product (**/*) as the left verb and power (*^*) as the right verb, that is (**/ . ^*), applied between a list of primes (*p:@i.@#*) on the left and a conforming list of exponents (*]*) on the right.

The PFEN numbers make the calculation of the product of two numbers easy. To multiply two numbers, add their PFENs.

Thus:

```

    72 * 90
6480
    pfd 72 90
3 2 0
1 2 1
    3 2 0 + 1 2 1
4 4 1
    dfp 4 4 1
6480

```

A verb to multiply PFENs can be defined:

```

    multp=: +/@,:
    3 2 0 multp 1 2 1
4 4 1

```

We're almost at the point of being able to show why squares have an odd number of divisors. One more detail is wanting, and that is, how to determine the number of divisors of a number. We could count the number of integers not greater than a given integer that have a give a residue of zero for that integer. To find that there are four divisors of eight, for example, one could write:

```

    i=: >:@i.
    i 8
1 2 3 4 5 6 7 8
    (i 8)|8
0 0 2 0 3 2 1 0
    0=(i 8)|8
1 1 0 1 0 0 0 1
    +/0=(i 8)|8
4

```

This method becomes unwieldy for large integers. A more compact method would be welcome.

If a number n has prime factors with exponents e , any number which is a divisor of n will have the same prime factors, with exponents which are less than or equal to e . Here I use q : with negative infinity as left argument. This gives another representation of an integer, where only the primes which have an exponent greater than zero are given, together with their positive exponents:

```

    pfd2=: __&q:
    pfd2 666
2 3 37
1 2 1

```

This signifies that 666 is composed of $(2^1) * (3^2) * (37^1)$, or $2 * 9 * 37$.

The inverse is given by:

```
dfp2=: pfd2^:_1
dfp2 pfd2 666
666
```

We can enumerate the divisors of 666 by taking all combinations of 1 2 with 1 3 9 with 1 37. A divisor will thus be one of the numbers generated as follows:

```
(2^i.2)*/(3^i.3)*/(37^i.2)
1 37
3 111
9 333
2 74
6 222
18 666
```

This array shows all the divisors. There are twelve altogether, and the twelve comes from the product over one plus the exponent of each prime: $*/ 1 + 1 2 1$, or $*/2 3 2$, or 12. Thus we can compute the number of divisors of a number by taking its PFEN, adding one to it, and taking the product:

```
*/ >: 1 2 1
12
```

All the powers that each prime in the composition of the number can take will be given by a table such as the one below, essentially all the numbers in the radix given by $1 + \text{PFEN } n$:

```
pfx=: ] #: [: i. */
pfx 1 + 1 2 1
0 0 0
0 0 1
0 1 0
0 1 1
0 2 0
0 2 1
1 0 0
1 0 1
1 1 0
1 1 1
1 2 0
1 2 1
```

```

      2 3 37 ^"1 pfx 1 + 1 2 1
1 1 1
1 1 37
1 3 1
1 3 37
1 9 1
1 9 37
2 1 1
2 1 37
2 3 1
2 3 37
2 9 1
2 9 37
*/ |: 2 3 37 ^"1 pfx 1 + 1 2 1
1 37 3 111 9 333 2 74 6 222 18 666

```

But we've gone a step too far; we don't need or want the values of the divisors, merely how many divisors there are. In reaching this point, however, we've found out how to arrive at this number: take the product over one plus the PFEN. We define the square verb `squrp2` and the number of divisors verb `nd2`:

```

      squrp2=: 1 2 * ]
      pfd2 666
2 3 37
1 2 1
      squrp2 pfd2 666
2 3 37
2 4 2
      dfp2 squrp2 pfd2 666
443556
      *: 666
443556
      nd2=: 13 : '*/ >: {: y'
      nd2
[: */ [: >: {:
      nd2 pfd2 666
12

```

We see that 666 has 12 divisors. How many does its square have?

```

      nd2 squrp2 pfd2 666
45

```

So the square of 666 (443556) has an odd number of divisors. Perhaps now we can see why. If we take any number, having any arbitrary PFENa, consisting of some mixture of odd and even numbers (zero is even), and add it to itself, (thus producing a

square) we'll obtain a PFENb in which *all* the numbers are even, since even plus even is even ($2n+2n$ is $4n$, an even number), and odd plus odd is even ($2n+1 + 2n+1$ is $4n+2$, an even number), and it is the square of the number given by PFENa. If we take the PFEN of 443556, or 2 4 2, and add 1 to each exponent, giving 3 5 3, we have a list of all odd numbers; taking the product of this, to yield the number of divisors of the square number PFENb, gives 45, an odd result (the only way a product can be odd is if *all* of the multiplicands are odd – any even multiplicand means the product will be even), so that the square PFENb has an odd number of divisors. This is a completely general result, and means that all square numbers, and only square numbers, have an odd number of divisors.

Amy only tested the first dozen or so numbers. I wrote a set of J verbs to allow testing all thousand locker numbers:

```
i=: integers      =: [: >: i. NB. +ve integers thru y
z=: gauge         =: = i NB. gauge 4 is 0 0 0 1, etc.
c=: cycles        =: $ z NB. recycle gauge y times
t=: table         =: c"0 i NB. make square table

tk=: table 1000
$tk
1000 1000

dc=: +/tk          NB. counts students touching a locker
$dc
1000
17{.dc
1 2 2 3 2 4 2 4 3 4 2 6 2 4 4 5 2

p1=: >: I. 2|dc NB. lockers open should all be squares
$p1
31
p1
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 ➡
324 361 400 441 484 529 576 625 676 729 784 841 900 961

NB. squares divisor count; all should be odd
sqdc=: nd2"2 pfd2"0 p1
sqdc
1 3 3 5 3 9 3 7 5 9 3 15 3 9 9 9 3 15 3 15 9 9 3 21 5 9 ➡
7 15 3 27 3
```


Here are the rest of the questions Amy had to answer. See if you can answer them:

- A. What do you notice^{*} about the lockers that were touched by exactly two students? (Try: $>: I. 2=dc$)
- B. What do you notice about the lockers that were touched by exactly three students?
- C. What do you notice about the lockers that were touched by exactly four students?
- D. What was the first locker touched by both student 6 and student 8?
- E. What do you notice about the student numbers of the students that touched both locker 24 and locker 36?
- F. Which students touched both locker 100 and 120? What do you notice about their student numbers?

Answers on page 354.

^{*} Notice first that the Locker Algorithm implements the *Sieve of Eratosthenes*, though not in the most efficient way. The 1000 lockers stand for the *natural numbers*, $\{n\}$, and each student m stands for a candidate factor of n . There are students for all factors, including 1 and n itself, not just for primes, as would result from the classical form of the *Sieve*). If therefore locker n is touched by just 2 students, n has only two factors, 1 and n , signifying n must be a prime number. (Ed.)

24 Blists in OEIS

First published in Vector, 17, 1, (July 2000), 110-120.

Revised by Oleg Kobchenko, (April 2009).

This article discusses a kind of list I call a *blist*. The first part defines a blist, and covers material that is well known in combinatorial circles, and reported on by me in an earlier article [1], and also gives an actual use of J's Weighted Taylor Coefficients adverb `t :`. The second part breaks new ground, providing a tabulation that hasn't been seen before.

Part 1: What is a blist?

A basic list, or blist, is a list of length n with at least one of each of the items of `i.k`, where $1 <: k$ and $k <: n$. For example, `0 2 1 0` is a blist, since it has at least one each of `i.3`, but `1 0 1 3` is not, since it has a three but no two. There is a many-to-one correspondence between the infinite number of arrays of n items and the corresponding finite number of blists of length n . The finitude of the number of blists of length n comes from the finitude of their permitted items. Since J's grade functions are omnivorous, the grade of any rank array can be found, and any array can be sorted, whether scalar or structured, boolean, integer, real, complex, literal, or boxed, and thus the blist of any array can be determined. The blist of an array can be determined by the function:

```
blist=: ] i.~ [: /:~ ~.
```

This finds the indices of the items of the array in the sorted nub of the array. For example, given the list:

```
] list=: ? 10 # 15
7 12 0 0 7 10 0 5 1 6
```

Its nub is

```
~.list
7 12 0 10 5 1 6
```

and its ordered nub is

```
/:~.list
0 1 5 6 7 10 12
```

and its indices in the ordered nub, that is, its blist, are

```
list i.~ /:~ ~. list
4 6 0 0 4 5 0 2 1 3
```

and this has each of the values in `i.7` at least once.

A blist has the useful property that it has the same ordering relations as infinitely many other, more complex, lists and arrays, and thus can be substituted for those other lists and arrays in discussions of such properties. For example, an array and its blist have the same upgrade:

```
list
7 12 0 0 7 10 0 5 1 6
bl list
4 6 0 0 4 5 0 2 1 3
/: list
2 3 6 8 7 9 0 4 5 1
/: bl list
2 3 6 8 7 9 0 4 5 1
```

Other common properties of arrays and their blists are the same, for example their cycle structure, the number of operations needed to sort them, and their number of runs (up or down).

BLT is a brute-force function to give tables of all the blists of a given length.

```
BLT=: 3 : 0
~. bl"1 (y#y)#: i.y^y
)
```

There is only one blist of length 1, since the only permitted item is 0.

```
BLT 1
0
```

The blists of length two are:

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

The blists of length three are:

```
BLT 3
0 0 0
0 0 1
0 1 0
0 1 1
```

```

0 1 2
0 2 1
1 0 0
1 0 1
1 0 2
1 1 0
1 2 0
2 0 1
2 1 0

```

The number of blists of the first three orders can be counted easily: 1, 3, and 13. On the other hand, the function BLT soon runs out of space on my computer, requiring $4 \cdot n \cdot n^n$ bytes to build the table from whose rows the blists are selected, and I can't use BLT beyond $n=7$. The space in bytes required for the tables for the first few values is given by:

```

j=:13 : '4*y*y^y'
j 1 2 3 4 5 6
4 32 324 4096 62500 1119744

```

and for a few larger values:

```

, .j 7 8 9 10 11 12x
      23059204
      536870912
      13947137604
      400000000000
      12553713506884
      427972821516288

```

Although it is difficult to determine the values of blists of length n for large n , the number of such blists is much easier to find. The answer to exercise 5.3.1-3 in Knuth's *Searching and Sorting* volume gives a variety of ways for doing this. We can write a function F to give the number of blists of length n , based on Gross's formula $\sum_{k \geq 1} k^n / 2^{1+k}$, $n \geq 1$. A version of Gross's formula in J is easy to write:

```
Gross=: 13 : '+/(x^y)%2^>:x'
```

The formula implies an infinite number of values of k are required, but in practice I find that using the first 101 positive integers suffices.

```

k=: >:i.101
F=: k&Gross

```

The number of possible blists for arrays of ranks 1 through 15 are:

n	F n	n	F n	n	F n
1	1	6	4683	11	1622632573
2	3	7	47293	12	28091567595
3	13	8	545835	13	526858348381
4	75	9	7087261	14	10641342970443
5	541	10	102247563	15	230283190977853

The terminal digits of the values repeat in the pattern 1 3 3 5, so that if $4|n$ is 1 2 3 0, then $10|F n$ is 1 3 3 5, respectively, for positive n . This series is number A000670 in N. J. A. Sloane’s magnificent website, *On-Line Encyclopedia of Integer Sequences* (OEIS) [2].

I shall be referring to Sloane’s OEIS numbers frequently in what follows.

This sequence of numbers arises naturally in a variety of areas, in addition to sorting, including trees with $n+1$ leaves, combination locks, compositions of numbers, and left arguments to APL’s transpose dyad, but it is the sorting topic that is most interesting. It allows one to say in exactly how many ways an arbitrary list of length n can be arranged. For example, three items A, B, and C of any value can be arranged in thirteen ways, depending on their size interrelationships, using the relations = and < and the convention that $A=B<C$ means $(A=B) * . (B<C)$. Next to each relation list I’ve placed the corresponding blist, to show the kinship of the two forms.

A=B=C	0 0 0
A=B<C	0 0 1
A=C<B	0 1 0
A<B=C	0 1 1
A<B<C	0 1 2
A<C<B	0 2 1
B=C<A	1 0 0
B<A=C	1 0 1
B<A<C	1 0 2
C<A=B	1 1 0
C<A<B	1 2 0
B<C<A	2 0 1
C<B<A	2 1 0

Gross’s formula, even after increasing the number of terms in the left argument, begins to lose accuracy after length 14. To obtain accurate values for P_n , the number of blists of length n , one can use the identity: $2P_n = \sum_k (k!n) * P_{n-k}$, for $n > 0$.

Instead of obtaining just one value, we obtain a list of all the values up to the one we're seeking, given that the first value is 1, and that all successive values can be appended to this value by a sum of the products of the list and a conforming list of binomials. For example, assuming we have the list 1 1 3, we can get the next two longer lists by:

```

      1 1 3 , +/ 1 1 3 * ((i.3)!3)
1 1 3 13
      1 1 3 13 , +/ 1 1 3 13 * ((i.4)!4)
1 1 3 13 75

```

The function LPA encapsulates this strategy:

```

LPA=: 13 : 'y,+/y*(i.!)]#y'
LPA 1
1 1
LPA LPA 1
1 1 3
LPA LPA LPA 1
1 1 3 13
LPA^:3[1
1 1 3 13

```

To produce the list of the first n terms, one would write $LPA^:n\ 1$. Because the terms grow large quite rapidly, it is necessary to use extended arguments if terms of high degree are wanted. We can thus obtain arbitrarily large values.

```

LPA=: 13 : 'y,+/y*(i.!)]#y'
,.28 29 30 31{LPA^:31[1x
6297562064950066033518373935334635
263478385263023690020893329044576861
11403568794011880483742464196184901963
510008036574269388430841024075918118973

```

Approximate values can be obtained by the function L, built around the powers of the logarithm of 2 (this isn't in Knuth, I found it accidentally by playing with the series):

```

L=: 13 : '(!y)%+:(^.2)^>:y'
L 8
545834.99790748546
L 9
7087261.0016229022

```

Rounding to the nearest integer, this function will give accurate results up to 13,

```
<.0.5+L 13
526858348381
```

but is off by one for 14:

```
<.0.5+L 14
10641342970444
```

We know this can't be right, since if $4 \mid 14$ is 2, then $10 \mid L \ 14$ must be 3, not 4.

Another way to get the number of blists for a given n is to use its exponential generating function (egf). I look back wistfully on myself at the age of 19 learning the calculus necessary to understand exponential generating functions, but in the 55 years since the knowledge has somehow departed from me. Now I can't tell you how to derive it, but will merely state it. If you have the necessary mathematical background to understand it (which I don't any more), you can read the answer to exercise 7.44 in Knuth et al's *Concrete Mathematics*. In common mathematical notation the egf for the number of blists is $1/(2-e^n)$, and the J version can be written directly from this:

```
paegf=: %@(2:-^)
```

The values don't seem to have much of a pattern:

```
paegf i.11
1 _1.3922 _0.18556 _0.055293 _0.019012 _0.00683
_0.0024911 _0.00091355 _0.00033569 _0.00012344
_4.5404e_5
```

However, if you apply J's Weighted Taylor Coefficients adverb to it, it becomes a marvel:

```
(paegf t:) i.11
1 1 3 13 75 541 4683 47293 545835 7087261
102247563
```

Lastly, and something else I discovered by playing with the series, if we divide the $n-1$ th value by the n th, and multiply this by n , we get a number which more and more closely approaches the natural logarithm of 2.

Compare this with the machine-precision value of the logarithm of 2:

Part 2: How many blists of length n begin with k ?

CL=: [: #/.~ [: {."1]

I began building an upper triangle matrix (analogous to the Pascal triangle), where an entry in row i , column j gives the number of blists of length $j+1$ that begin with integer i .

```

      CL BLT 1
1
      CL BLT 2
2 1
      CL BLT 3
6 5 2
      CL BLT 4
26 25 18 6
      CL BLT 5
150 149 134 84 24
      CL BLT 6
1082 1081 1050 870 480 120
      CL BLT 7
9366 9365 9302 8700 6600 3240 720

```

The last one took a looong time. With these I was able to create table t1:

```

      t1
1 2 6 26 150 1082 9366
0 1 5 25 149 1081 9365
0 0 2 18 134 1050 9302
0 0 0 6 84 870 8700
0 0 0 0 24 480 6600
0 0 0 0 0 120 3240
0 0 0 0 0 0 720

```

Portions of this table appear in OEIS. Its sum is A000670. The first row is Series A000629. The second row is one less than the first row, and is Series A002050. The third row is the first row minus 2^n , and is twice Series A002051. The lowest counterdiagonal is $!n$. The penultimate counterdiagonal is A038720. I massaged the numbers in various ways, the fruitful one being to take its first difference, providing a new last row of all zeros to preserve the data:

```

      ] t2=:2-/\t1,0
1 1 1 1 1 1 1
0 1 3 7 15 31 63
0 0 2 12 50 180 602
0 0 0 6 60 390 2100
0 0 0 0 24 360 3360
0 0 0 0 0 120 2520
0 0 0 0 0 0 720

```

This is series A028246 from OEIS. It looks promising, but I want to remove the factorials from the rows:

```

] t3=:t2%|i.#t2
1 1 1 1 1 1 1
0 1 3 7 15 31 63
0 0 1 6 25 90 301
0 0 0 1 10 65 350
0 0 0 0 1 15 140
0 0 0 0 0 1 21
0 0 0 0 0 0 1

```

And this brings me to familiar territory. It is the table of the Stirling numbers of the second kind, also called *subset* numbers, and is series A008277 from OEIS. It is usually displayed transposed from the table above, and with an added first row and first column.

```

((1+#t3){.1),.0,|:t3
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 1 0 0 0 0 0
0 1 3 1 0 0 0 0
0 1 7 6 1 0 0 0
0 1 15 25 10 1 0 0
0 1 31 90 65 15 1 0
0 1 63 301 350 140 21 1

```

The reason these are called subset numbers is that entry (i, j) gives the number of ways to partition a set of i items into j nonempty parts. Thus, the value 7 in row 4, column 2, says there are 7 ways to put 4 items into 2 nonempty parts:

$(abc, d); (abd, c); (acd, b); (bcd, a); (ab, cd); (ac, bd); (ad, bc)$

```

+/t3
1 2 5 15 52 203 877

```

These are the Bell numbers (series A000110), which give the total number of ways of placing n distinct objects in n boxes. This is to be expected, since the subset number in item $(n; k)$ gives the number of ways to partition a set of n things into k nonempty subsets. The Bell numbers thus summarize the subset numbers.

But now I know how to create *my* table of numbers. I can use the nonrecursive function `s2nr` from Iverson's *Concrete Math Companion* to generate the subset numbers.

This may be a bit mysterious at first, so I'll show you how its parenthesized central portion works.

```
s2nr=: |: @ (^/~ %. ^!._1/~) @ i."0
] v0 =. i.7x
0 1 2 3 4 5 6
```

Form the table of powers t4

```
] t4 =. ^/~ v0
1 0 0 0 0 0 0
1 1 1 1 1 1 1
1 2 4 8 16 32 64
1 3 9 27 81 243 729
1 4 16 64 256 1024 4096
1 5 25 125 625 3125 15625
1 6 36 216 1296 7776 46656
```

and the table of falling factorials t5

```
] t5 =. ^!._1/~ v0
1 0 0 0 0 0 0
1 1 0 0 0 0 0
1 2 2 0 0 0 0
1 3 6 6 0 0 0
1 4 12 24 24 0 0
1 5 20 60 120 120 0
1 6 30 120 360 720 720
```

and make these the left and right arguments to matrix divide, yielding the table of subset numbers.

```
] t6=: |: t4 %. t5
1 0 0 0 0 0 0
0 1 0 0 0 0 0
0 1 1 0 0 0 0
0 1 3 1 0 0 0
0 1 7 6 1 0 0
0 1 15 25 10 1 0
0 1 31 90 65 15 1
```

I now can produce the table of leading digits versus length of splits. The first step is to build a table of subset numbers.

```
] a =. s2nr 10x
```

```

1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0
0 1 3 1 0 0 0 0 0
0 1 7 6 1 0 0 0 0
0 1 15 25 10 1 0 0 0
0 1 31 90 65 15 1 0 0
0 1 63 301 350 140 21 1 0
0 1 127 966 1701 1050 266 28 1 0
0 1 255 3025 7770 6951 2646 462 36 1

```

Drop the leading row and column, then transpose.

```

] b =. |: 1 1 }. a
1 1 1 1 1 1 1 1 1
0 1 3 7 15 31 63 127 255
0 0 1 6 25 90 301 966 3025
0 0 0 1 10 65 350 1701 7770
0 0 0 0 1 15 140 1050 6951
0 0 0 0 0 1 21 266 2646
0 0 0 0 0 0 1 28 462
0 0 0 0 0 0 0 1 36
0 0 0 0 0 0 0 0 1

```

Multiply row i by factorial i .

```

] c =. b * ! i. # b
1 1 1 1 1 1 1 1 1
0 1 3 7 15 31 63 127 255
0 0 2 12 50 180 602 1932 6050
0 0 0 6 60 390 2100 10206 46620
0 0 0 0 24 360 3360 25200 166824
0 0 0 0 0 120 2520 31920 317520
0 0 0 0 0 0 720 20160 332640
0 0 0 0 0 0 0 5040 181440
0 0 0 0 0 0 0 0 40320

```

Sum from the bottom up.

```

] d =. +/ \. c
1 2 6 26 150 1082 9366 94586 1091670
0 1 5 25 149 1081 9365 94585 1091669
0 0 2 18 134 1050 9302 94458 1091414
0 0 0 6 84 870 8700 92526 1085364
0 0 0 0 24 480 6600 82320 1038744
0 0 0 0 0 120 3240 57120 871920
0 0 0 0 0 0 720 25200 554400
0 0 0 0 0 0 0 5040 221760
0 0 0 0 0 0 0 0 40320

```

And this is the table I wanted to be able to create.

I've told you the series numbers in OEIS of parts of my table. What about the table itself? I'm pleased and proud to be able to tell you that when I emailed information about it to Neil Sloane, proprietor of OEIS, he agreed it was new, and assigned the number A054255 to it, with credit to me. I feel as if I've gained a speck of immortality. We now have blists in OEIS. You can look it up!

Reference

- [1] McDonnell, E. E., How Shall I Transpose Thee? Let Me Count The Ways. *APL Quote Quad*, 8, 1, (1977-09).
- [2] Sloane, N. J. A., *On-Line Encyclopedia of Integer Sequences*.
<http://www.research.att.com/~njas/sequences/>

25 Someone Just Moved! Who Was It? Or, Apter's Puzzle

First published in Vector, 17, 2, (October 2000), 116-130.

Revised by Kip Murray, (April 2009).

The Puzzle

Stevan Apter recently proposed this puzzle to the online K group: Given a list of distinct items, and a second list changed by moving only one item of the original, find which item has been moved.

There was a fair amount of discussion about this, and a number of proposed solutions, a surprising number of which were erroneous. This paper gives a solution that I believe works properly in all cases. The greater part of the paper, however, discusses the reverse problem: Given a solution, find the list that gave rise to it.

Rotated Infix Permutations

First, to solve Apter's problem: Given the two lists A and B:

```
A=: 3 1 4 5 9 2 6
B=: 3 1 5 9 2 4 6
A ,: B
3 1 4 5 9 2 6
3 1 5 9 2 4 6
```

tell which item in A was moved in order to produce B. It couldn't have been the 5 or 9 or 2 because they form an infix in the order of the original. The 4 (shown underlined), which had preceded 5 and 9 and 2 is the one that is out of order: it is now at their right: it has been moved from index 2 to index 5.

The problem is simplified if the items are replaced by their indices. Since the items are distinct, A can be replaced by the identity permutation, and B by the indices of its items in A. For example:

```
C=: i. # A
D=: A i. B
C ,: D
0 1 2 3 4 5 6
0 1 3 4 5 2 6
```

C and D contain all the information needed to solve the original A and B problem. In fact, D is all that is needed: the identity permutation C can be understood. In what follows, I'll assume that a

problem list is given in this D form. Thus an argument to the solution program might be:

D
0 1 3 4 5 2 6

Comparing D to C shows that some of the items remain in their original positions, but others have been moved. The following shows the items that have been moved in bold. These form a rotated infix; because of this I'll call D a *rotated infix permutation*, which for convenience will be abbreviated to *rip*.

0 1 3 4 5 2 6

The nonzero items of D-C show which have been moved:

D-C
0 0 1 1 1 _3 0

Those which have been displaced by the move produce a 1 in the difference; the one moved produces _3. But an item can be moved to a lower position as well as to a higher. Suppose we are given to solve:

E
0 1 5 2 3 4 6

The items in bold again represent a 1-rotation, but this time it is to the right. If we subtract C from this we get:

E-C
0 0 3 _1 _1 _1 0

Here the value corresponding to the moved item is positive value, and the displaced items produce _1s. Considering both cases, it is evident that the moved item is determined by finding the maximum magnitude in the difference between the list to solve and the identity permutation. There are two difficulties to discuss. The first difficulty: suppose we move an item just one position to the right or left: what results?

F=: 0 1 3 2 4 5 6
F-C
0 0 1 _1 0 0 0

The magnitudes of F-C show two possible results. It's unclear whether the 3 has been moved to position 2, or the 2 has been moved to position 3. Either one can be chosen. The rule used here is: Among equal maxima, choose the one occurring first. This is easiest because of the way J's *Index of primitive* (i.) is defined,

because it gives the index of the first occurrence of the item sought. This primitive has right of seniority over J's *Index of Last* primitive (*i* :), which is a relative newcomer; *Index of* antedates even APL: it is described in Iverson's book *A Programming Language* [1] in section 1.16 *Ranking*, page 31:

The *rank* or *index* of an element $c \in b$ is called the *b index of c* and is defined as the smallest value of *i* such that $c = b_i$.

Thus, for the rip F above, 3 will be identified as the item that has been moved—even though it might in fact have been produced by moving 2 to position 3.

The second difficulty arises from the possibility, not excluded in Apter's statement of the problem, of moving an item from its original position to its original position. For example, the list:

```
] G=: i.7
0 1 2 3 4 5 6
```

if proposed as a problem to solve, might be the result of moving any of the seven items back to its original position. What we get if we subtract C from G, (G-C), is 0 0 0 0 0 0 0.

Again following the rule *among equal maxima, choose the first*, I would find 0 as the one having been moved (to 0).

Canonical Specifications

A rip such as D, E, F, or G can be represented by a list of three integers: its length L, the initial position I of the moved item, and its final position F. For example, the rip D is specified by 7 2 5; E by 7 5 2; F by 7 3 3; and G by 7 0 0. I'll call such a list the rip's *canonical representation*, or *casp*.

The function CfR below solves Apter's problem, yielding the casp from the rip:

```
CfR=: monad define"1
NB. casp from rip
R=. y
F=. (i.>.)|(-i.@#)R
L=. #R
I=. F{R
L,I,F
)
```

Applying it to the sample rips:

```

      CfR D
7 2 5
      CfR E
7 5 2
      CfR F
7 3 2
      CfR G
7 0 0

```

For aficionados of the one liner:

```
OLCfR=: # , (([ , ~ {) ~ (i. >. /) @ ([: | ] - [: i. #)) ~
```

A function inverse to CfR would take a casp L, I, F and yield the rip which gave rise to it. It can be described informally like this:

lay out a row of L numbered blocks

```
0 1 2 3 4 5 6
```

remove the Ith block (which has the number I on it) and set it aside

```
0 1  3 4 5 6           2
```

collapse the remaining blocks over the empty space

```
0 1 3 4 5 6           2
```

separate these into two parts, the first F and the rest

```
0 1 3 4 5  6           2
```

and insert the removed block in the space made (which is F)

```
0 1 3 4 5 2 6
```

This function RfC which, given a casp, yields a rip, is:

```

      RfC=: monad define"1
      'L I F'=. . y
      M=. I-.~i.L
      (F{.M),I,(F}.M)
    )

```

Here are some uses of RfC:

```

      RfC 7 2 5
0 1 3 4 5 2 6

```

```

      RfC 7 5 2
0 1 5 2 3 4 6
      RfC 7 3 2
0 1 3 2 4 5 6
      RfC 7 0 0
0 1 2 3 4 5 6
    
```

Roughly speaking, CfR and RfC are inverses, and thus we'd like to be able to say:

```

D -: RfC CfR D
S -: CfR RfC (S=: CfR D)
    
```

The function MC below, given a positive integer argument, yields a 3-column table of all the casps for rips of that length. For a list of length n , there are n^2 rips, one for each initial position versus each final position. A variation of the odometer function is required. Given a length L , form: $i . * : L$, then the (L,L) representation of each, and finally, prefix L to each, ending with an L^2 by 3 table:

```

MC=: 13 : 'y,.(2#y)#:i.*:y'
] C3=: MC 3
3 0 0
3 0 1
3 0 2
3 1 0
3 1 1
3 1 2
3 2 0
3 2 1
3 2 2
    
```

Next, I'll get the corresponding rips:

```
R3=: RfC C3
```

And finally, put C3 next to R3 so we can see the correspondences:

```

2 2 2 6 2 2": C3,.R3
3 0 0      0 1 2
3 0 1      1 0 2
3 0 2      1 2 0
3 1 0      1 0 2
3 1 1      0 1 2
3 1 2      0 2 1
3 2 0      2 0 1
3 2 1      0 2 1
3 2 2      0 1 2
    
```

The number of distinct results is not L^2 , but less than that. When I and F are the same the results are the same: each of 3 0 0, 3 1 1, and 3 2 2 yields 0 1 2. From these three solutions we get only one cusp, so we can reduce the initial 9 by 2. In general, the diminishment coming from this case is $L-1$. Next, the results are the same when I and F are adjacent numbers, as in 3 0 1 and 3 1 0, and 3 1 2 and 3 2 1. From these four casps we get only two rips, so we can reduce the total by another 2. In general, the diminishment coming from this case is also $L-1$, one for each adjacent pair. The 9 results are thus reduced by another 2, leaving just 5. The general formula for the number of distinct rips is $L^2-2(L-1)$, or, simplified, L^2-2L+2 , which gives the J polynomial function $2 _2 \ 1\&p..$

The function NR below, given an integer argument, yields the number of distinct rips of that length:

```
NR=: 2 _2 1&p.
( ],.NR)>:i.10
1 1
2 2
3 5
4 10
5 17
6 26
7 37
8 50
9 65
10 82
```

The Anatomy of Rips

I've been in the habit of checking sequences such as the second column above against the entries in Neil Sloane's invaluable collection, which appeared in print first in his *A Handbook of Integer Sequences* (Academic Press, 1973). Recently the book has been supplemented by an ever growing collection on his web site, *On-Line Encyclopedia of Integer Sequences*, at:

<http://www.research.att.com/~njas/sequences/>

The series 1 2 5 10... is ID Number A002522, which describes it in offset 0 as n^2+1 . My series is offset 1, which implies the formula previously given, namely n^2-2n+2 . It also notes that this sequence

5
10 13
17 20 25
26 29 34 41
37 40 45 52 61

I began my study of rips by finding the *anagram index* of a fair number of them, using J's A. primitive, and listing on a sheet of paper those of length seven in order by length of rotated infix, and within that by highest maximum I and F. Here are those with rotated infix of length 3, and the associated anagram index (I've underlined the disordered infix items).

I<F										
L	I	F	rip							A.
7	4	6	0	1	2	3	<u>5</u>	<u>6</u>	<u>4</u>	3
7	3	5	0	1	2	<u>4</u>	<u>5</u>	<u>3</u>	6	8
7	2	4	0	1	<u>3</u>	<u>4</u>	<u>2</u>	5	6	30
7	1	3	0	<u>2</u>	<u>3</u>	<u>1</u>	4	5	6	144
7	0	2	<u>1</u>	<u>2</u>	<u>0</u>	<u>3</u>	4	5	6	840

I>F													
L	I	F	rip										A.
7	6	4	0	1	2	3	<u>6</u>	<u>4</u>	<u>5</u>			4	
7	5	3	0	1	2	<u>5</u>	<u>3</u>	<u>4</u>	6			12	
7	4	2	0	1	<u>4</u>	<u>2</u>	<u>3</u>	5	6			48	
7	3	1	0	<u>3</u>	<u>1</u>	<u>2</u>	4	5	6			240	
7	2	0	<u>2</u>	<u>0</u>	<u>1</u>	3	4	5	6	1440			

The obvious way of entering the data thus gathered was to identify the rows with I and the columns with F, and to put the anagram index in item for I, F in row I, column F. This was unsatisfactory, since the numbers were going in what seemed the wrong direction, so instead I made tables where the rows were numbered by how far the right edge of the infix was from the right end of the table, and the columns were numbered by the

length of the infix. Thus the anagram index of 30 where the L I F is 7 2 4, which has the infix 2 spaces from the right edge and has length 3 would be entered at row 2 column 3 of the I<F table.

	I<F						
	1	2	3	4	5	6	7
0	0	1	3	9	33	153	873
1	0	2	8	32	152	872	
2	0	6	30	150	870		
3	0	24	144	864			
4	0	120	840				
5	0	720					
6	0						

	I>F						
	1	2	3	4	5	6	7
0	0	1	4	18	96	600	4320
1	0	2	12	72	480	3600	
2	0	6	48	360	2880		
3	0	24	240	2160			
4	0	120	1440				
5	0	720					
6	0						

Studying these tables convinced me that I could see the rule determining the value of an entry. For the I<F table, column 1 would always be zero; an infix of length one meant that I and F were the same, so the result would be the identity permutation, which is permutation 0 for permutations of all lengths. Column 2 would always be $!(1+\text{row number})$. Column j for $j>2$ would be sums of $(j-1)$ successive items of column 2. For example, the entries in column 3 would be $1+2$, $2+6$, $6+24$...; in column 4 would be $1+2+6$, $2+6+24$, $6+24+120$,... and so forth. This can be shown using J's Infix adverb, $x\ u\ y$, where the function u (in our case $+/\$) is applied over successive length- x infixes of y :

```
] fs=: !i.10
1 1 2 6 24 120 720 5040 40320 362880
  2+/\fs
2 3 8 30 144 840 5760 45360 403200
  3+/\fs
4 9 32 150 864 5880 46080 408240
  4+/\fs
10 33 152 870 5904 46200 408960
```

For the I>F table, the entries in column 1 and 2 would be the same as in the I<F table, for the same reasons. Column j for $j>2$

would be $(j-1)*(j-2)$ drop column 2. For example, the entries in column 3 would be 2*1 drop 1 2 6 24 120...; in column 4 would be 3*2 drop 1 2 6 24 120...; and so forth.

I was able to verify my conjectures for both tables after a few experiments, and wrote two functions that would give me the value for any entry in either table:

```
ILF=: 13 : ' +/!(x+1)+i.y-1' "0
IGF=: 13 : ' (y-1)*!(y-1)+x' "0
3 ILF 4
864
3 IGF 4
2160
```

The `by`, `over`, and `tab` functions shown below come from J's Help | Phrases | 2.D*

```
d16=: by=: ' &@,.@[,.]
d17=: over=: ({.;.).@":@,
a18=: tab=: 1 : '[ by ] over u/'
(i.7) ILF tab 1+i.7x
```

	1	2	3	4	5	6	7
0 0	1	3	9	33	153	873	
1 0	2	8	32	152	872	5912	
2 0	6	30	150	870	5910	46230	
3 0	24	144	864	5904	46224	409104	
4 0	120	840	5880	46200	409080	4037880	
5 0	720	5760	46080	408960	4037760	43954560	
6 0	5040	45360	408240	4037040	43953840	522955440	

```
(i.7) IGF tab 1+i.7x
```

	1	2	3	4	5	6	7
0 0	1	4	18	96	600	4320	
1 0	2	12	72	480	3600	30240	
2 0	6	48	360	2880	25200	241920	
3 0	24	240	2160	20160	201600	2177280	
4 0	120	1440	15120	161280	1814400	21772800	
5 0	720	10080	120960	1451520	18144000	239500800	
6 0	5040	80640	1088640	14515200	199584000	2874009600	

* where `tab` is now shown as `table` (Ed.)

In both ILF and IGF the arguments are modified by adding or subtracting 1. I wondered whether I could get any further insights by writing versions of ILF and IGF in which the arguments were not offset.

```
ILFx=: 13 : '+/!x+i.y'"0
ILFx tab~i.7x
+-----+
| |0| 1| 2| 3| 4| 5| 6|
+-----+
|0|0| 1| 2| 4| 10| 34| 154|3422
|1|0| 1| 3| 9| 33| 153| 873|7489
|2|0| 2| 8| 32| 152| 872| 5912|54116
|3|0| 6| 30| 150| 870| 5910| 46230|54117
|4|0| 24| 144| 864| 5904| 46224| 409104|54118
|5|0| 120| 840| 5880| 46200| 409080| 4037880|
|6|0| 720| 5760| 46080| 408960| 4037760| 43954560|
+-----+
4 142 1048

IGFx=: 13 : 'y*!y+x'"0
IGFx tab~i.7x
+-----+
| |0| 1| 2| 3| 4| 5| 6|
+-----+
|0|0| 1| 4| 18| 96| 600| 4320|1563
|1|0| 2| 12| 72| 480| 3600| 30240|18931
|2|0| 6| 48| 360| 2880| 25200| 241920|52571
|3|0| 24| 240| 2160| 20160| 201600| 2177280|52520
|4|0| 120| 1440| 15120| 161280| 1814400| 21772800|52557
|5|0| 720| 10080| 120960| 1451520| 18144000| 239500800|52521
|6|0| 5040| 80640| 1088640| 14515200| 199584000| 2874009600|
+-----+
4 142 52849 52560 52578 52648
```

I've put numbers to the right of those rows and at the bottom of those columns which correspond to entries in Sloane's *Encyclopedia*. The *Encyclopedia* doesn't contain entries for all the rows and columns of ILFx and IGFx, but they are easily obtained. The functions below give the first x items of the indicated row or column:

```
NB. x items of row y of ILFx
ILFI=: 13 : '+/!y+i.x:x'
NB. x items of column y of ILFx
ILFJ=: 13 : 'y+/\!i.<:y+x:x'
NB. x items of row y of IGFx
IGFI=: 13 : '(!y+i.10x)*i.x:x'
NB. x items of column y of IGFx
IGFJ=: 13 : 'y*!y+i.x:x'
```

And here they are, applied to 10 items of row 3 and column 3 for each table:


```

10 ILFI 3
6 30 150 870 5910 46230 409110 4037910 43954710 ➡
522956310
10 ILFJ 3
4 9 32 150 864 5880 46080 408240 4032000 43908480
10 IGFI 3
0 24 240 2160 20160 201600 2177280 25401600 319334400 ➡
4311014400
10 IGFJ 3
18 72 360 2160 15120 120960 1088640 10886400 ➡
119750400 1437004800

```

Sloane contains tables and triangles as well as linear sequences. Table ILF x is closely related to Sloane's sequence 54115:

```

      1
    1 1
  1 2 3
1 6 8 9
1 24 30 32 33
1 120 144 150 152 151

```

The formula for this triangular array T is given as:

Triangular array generated by its row sums:

$T(n,0) = 1$ for $n \geq 1$,

$T(n,1) = r(n-1)$,

$T(n,k) = T(n,k-1) + r(n-k)$ for $k=2, 3, \dots, n$, $n \geq 2$,

$r(h)$ = sum of the numbers in row h of T .

The rows of this triangle are derived from ILF x by reading the counterdiagonals from left to right. We can get the counterdiagonals in J by using J 's *Box* and *Oblique*:

```

, .7{ .</ . | :ILF $x$ /~i .7
+-----+
|0|
+-----+
|0 1|
+-----+
|0 1 2|
+-----+
|0 2 3 4|
+-----+
|0 6 8 9 10|
+-----+
|0 24 30 32 33 34|
+-----+
|0 120 144 150 152 153 154|
+-----+

```

Here it is in triangular form:

						0
				0	1	
			0	1	2	
		0	2	3	4	
	0	6	8	9	10	
0	24	30	32	33	34	
0	120	144	150	152	153	154

Table IGFx is closely related to Sloane’s 51683:

					1
			2	4	
		6	12	18	
	24	48	72	96	
120	240	360	480	600	
720	1440	2160	2880	3600	4320

The rule for this triangle is given in 1-offset as Table A:
 $a(n,k)=n!*k$; they are just integer multiples of factorials.
For example, $a(5,3) = 5!*3$, or $120*3$ or 360 .

The *J Phrases* book gives only a limited number of functions concerning rips.

```
NB. Phrases from 7.A
NB. Rotate last three to the left
m7=: 3&A.
NB. Rotate last three right
m8=: 4&A.
NB. Rotate last x to the left
d9=: ([: +/[:![:].[:i.[) A. ]
NB. Rotate last x to the right
d10=: (!@[ - !@<:@[) A. ]
```

The functions given here expand the possibilities significantly. They are of dubious practical value, because the factorials grow so large so quickly that they really aren’t practical to generate rips of any great length. For that, the function RfC is far more practical.

Contracurrency

All the while I was working on this material it had been bothering me that the Anagram index grew with respect to the end of the rip, not the beginning. For example,

```
A. 1 2 0
3
```

A. 0 2 3 1
 3
 A. 0 1 3 4 2
 3
 A. 0 1 2 4 5 3
 3

The Anagram index is the same regardless of the length of the rip, when the infix is the same distance from the *right* end. When the infix is the same distance from the *front* end, it varies:

A. 1 2 0
 3
 A. 1 2 0 3
 8
 A. 1 2 0 3 4
 30
 A. 1 2 0 3 4 5
 144

J supports contracurrent indexing, which is right-end oriented: the last item has contracurrent index _1, the penultimate has _2, and the antepenultimate has _3, regardless of the number of items. The previous tables were labelled by length of infix and distance from right edge, with the direction of 1-rotation used to distinguish two separate tables, as produced by the functions ILF and IGF. The function below produces a table labelled by contracurrent indices:

RfMC=: 13 : '(-y)]\A.RfC MC y'

This forms the table of all casps for rips of length *y*, then obtains the rips, next obtains the Anagram indices, and last, reshapes this into a square table.

```
] q=: |.->:i.8
_8 _7 _6 _5 _4 _3 _2 _1
q by q over RfMC 8
```

	_8	_7	_6	_5	_4	_3	_2	_1
_8	0	5040	5760	5880	5904	5910	5912	5913
_7	5040	0	720	840	864	870	872	873
_6	10080	720	0	120	144	150	152	153
_5	15120	1440	120	0	24	30	32	33
_4	20160	2160	240	24	0	6	8	9
_3	25200	2880	360	48	6	0	2	3
_2	30240	3600	480	72	12	2	0	1
_1	35280	4320	600	96	18	4	1	0

Entry (i,j) in this table is the Anagram index of a rip of any length in which item i has been moved to index j , where i and j are contracurrent indices.

Here are functions to convert between direct casps and contracurrent casps (ccasps):

```
NB. contracurrent casp from direct
CCfD=: --~`,`:3"1
```

```
NB. direct casp from contracurrent
DfCC=: (|@<. + 0: , ,)/"1
```

Here is a little experiment with the above two functions which shows the limitations of the ccasp form:

```
MC3=: MC 3
MCC3=: CCfD MC3
MC3x=: DfCC MCC3
MCC3x=: CCfD MC3x
format=: 2 2 2 5 3 5 2 2 5 3
format ": MC3, .MCC3, .MC3x, .MCC3x
3 0 0    _3 _3    3 0 0    _3 _3
3 0 1    _3 _2    3 0 1    _3 _2
3 0 2    _3 _1    3 0 2    _3 _1
3 1 0    _2 _3    3 1 0    _2 _3
3 1 1    _2 _2    2 0 0    _2 _2
3 1 2    _2 _1    2 0 1    _2 _1
3 2 0    _1 _3    3 2 0    _1 _3
3 2 1    _1 _2    2 1 0    _1 _2
3 2 2    _1 _1    1 0 0    _1 _1
```

A ccasp needs only two items: the contracurrent indices of the item moved and where it is moved to. This loses the information of the length of the rip involved, and so the conversion from ccasp back to casp is not exact: the length imputed is the magnitude of the minimum item. For example, 3 2 1 is converted to _1 _2, but the back conversion is 2 1 0, since the magnitude of the minimum item _2 is 2. However, 2 1 0 is converted exactly back to _1 _2.

Reference

[1] Iverson, K. E., *A Programming Language*. Wiley, (December 1962). ISBN 0471430145.

26 Four Cubes Redux

First published in Vector, 17, 3, (January 2001), 113-120.

Revised by Bill Lam, (April 2009).

*A Festschrift for Kenneth Iverson
on his 80th birthday,
17 December 2000.*

I recently cleaned out the chest of drawers in my bedroom, in the course of which I got rid of many frayed and yellowed handkerchiefs, never-worn T-shirts, paper thin undergarments, and, much to my pleasant surprise, I excavated a set of four coloured cubes, an inch and a quarter to the side, a modern version of a puzzle dating back at least a century, under various names; the set I had is marketed under the name *Instant Insanity* in the United States by Parker Brothers, the purveyor of many other games, most notably Monopoly. The cubes' faces are coloured with one of the four colours blue, green, red, or white, in some mixture. You are asked to stack the cubes one above the other in such a way that each side of the stack contains a face with each of the four colours, in some order.

In 1981 I had written a pamphlet on the puzzle called *The Four Cube Problem*[1], subtitled "a case study in Basic, APL, and functional programming." In the pamphlet a prize-winning Basic solution and an APL solution written by me were compared.

The Basic program had 91 non-comment lines, and the APL had nine; the average length of a Basic line was 31 characters; of the APL line 21 characters; Basic used 18 variables, APL none; Basic had 21 loops, APL none. The Basic program was written for and executed on the Sorcerer computer, which I suspect was a fairly early desktop computer, and so its execution time of 3 minutes and 5 seconds can't be fairly compared with the APL program, which took 0.7 seconds to execute on one of the largest and fastest commercial computers available in 1981, the Amdahl V8 computer in the I. P. Sharp machine room in Toronto.

The APL solution emphasized the *functional programming* approach introduced by John Backus[2], employing the direct definition form by K. E. Iverson as a way of facilitating the

definition of functions and their use in exposition. The solution used one constant function, one dyad, and seven monads.

The APL solution from my 1981 pamphlet could, I believe, be easily translated into Dyalog APL:

```
I:C S (Bω[1;A])O S(Bω[2;A])O S(Bω[3;A])O G Bω[4;X]
X:3 4ρ 3 5 4 6 1 3 2 4 1 6 2 5
G:2 1 3 4ϕ(1,ρω)ρω
B:(Rω)∘.= 'BGRW'
R:(∨/<ϕω)^.=ϕω)≠ω
A:(ι24)ϕ4≠X,[1]ϕX
O:((1↑ρω)≠α),[2](((1↑ρα),3ρ1)×ρω)ρω
S:(^/^/^/ω=<[2]ω)≠ω
C:'BGRW'[ω+.×ι4]
```

This solution took as argument a 4 by 6 character matrix representing the cubes, with the initial letters of the face colours given in the order top, bottom, left, right, front, and back; the colours were blue, green, red, and white, abbreviated by 'bgrw'. The result yielded was a three-dimensional array of shape $n, 4, 4$, where n is the number of solutions for the given set of cubes. A belt in a cube consists of a circuit of four faces; there are three basic belts in a cube: left, front, right, back; top, left, bottom, right; and top, back, bottom, front. The last cube is taken as the base of the stack, and only its three basic belts are considered, since rotating or reversing these belts will not provide any essentially different solutions. The face numbers of the basic belts are given by the constant X . For the other cubes, a total of 24 belts are considered, obtained by rotation and reversal of the basic belts in all possible ways. These are given by the function A .

The faces are represented by a boolean vector of length four containing a single 1 in the position corresponding to the colour:

```
Blue:      1 0 0 0
Green:     0 1 0 0
Red:       0 0 1 0
White:     0 0 0 1
```

A boolean representation was chosen in order to economize on space. This conversion is done by B , and the inverse by C . In the display below you can see the sizes of the intermediate arrays formed during execution. The largest needs space for $*720\ 4\ 4\ 4$ or 46,080 items. That is how many bytes are needed for a character representation. If bits are substituted, only one-

eighth, or 5,760 bytes are needed. Integers would require four times, or 184,320 bytes: a prohibitive size for many systems of that day.

The development of the result, and the shapes of the intermediate arrays are shown below:

```

I:C S (Bω[1;A])O S(Bω[2;A])O S(Bω[3;A])O G Bω[4;X]
      ←2 4 4→      ←2 4 4→      ←2 4 4→      ←3 4→→  1
      ←16 4 4→→    ←24 4 4→→    ←24 4 4→→    ←3 4 4→→  2
                                      ←3 1 4 4→→  3
                                      ←-----72 2 4 4-----→  4
                                      ←-----28 2 4 4-----→  5
                                      ←-----672 3 4 4-----→  6
                                      ←-----45 3 4 4-----→  7
                                      ←-----720 4 4 4-----→  8
                                      ←-----1 4 4 4-----→  9
                                      ←-----1 4 4-----→ 10

```

At the right of line 1 the shape of the belts of the bottom cube is 3 4. In line 2 the boolean conversion has been made, and the shape is 3 4 4. In line 3 the function G has added a dimension of length 1; this is needed so that each belt from the next cube can be stacked on each existing stack.

Function O does this stacking; function S removes stacks containing duplicate face colours. This process continues with the remaining cubes, and finally C converts the boolean vector forms into colour letters.

This brief synopsis is a necessary prelude to the description of the J solution.

The function I in APL has this equivalent in J:

```
i=: (0&{) (s@o) (1&{) (s@o) (2&{) (s@o x) (3&{)
```

J's *insert (/)* adverb is defined this way in the *J Dictionary* on gerund left arguments:

m/y inserts successive verbs from the gerund m between items of y, extending m cyclically as required. Thus, +`*/i.6 is 0+1*2+3*4+5.

This suggests that i could be rewritten like this:

```
g=: (s@o)`(s@o)`(s@o x)/
```

Whereas in both APL's **I** function and J's (irrelevant) **i** function it is necessary to specify index values to select rows of the argument, the gerund form of **g**, using J's *insert* primitive (**/**) depends on the nature of insertion to go through the rows of the argument in the proper order, from the bottom up.

The remaining functions are detailed as follows:

```
b=: 0 2 1 3 0 4 1 5 2 4 3 5
x=: (3 1 4$b) &{
o=: [: ,/ ([: ~. [: a [) , "1 2/ ]
a=: (i.24) |."1 [: [ 4: # [: (] , |."1) (3 4$b) { ]
s=: ] #~ 2: > [: >."1 [: , "2 [: +/"3 ] =/ [: ~. ,
```

The constant **b** is used here in order to keep the length of displays within the confines of the page width. The **b**'s in functions **x** and **a** should be replaced by the value of **b**. Thus there are actually only five functions needed: **g**, **x**, **o**, **a**, and **s**, and, indeed, **g** could be replaced by its body, reducing it to only four functions. The function **R** of the APL version is replaced by the J primitive *nub* (**~.**). The functions **B** and **C** of the APL solution convert between the character and the boolean list representations used. These are not needed in the J solution because it is designed to be generic, permitting any form of representing the cubes. The nine APL functions use 151 tokens. The five J functions use 106 tokens.

The function **x** applies to the bottom row of the argument, and produces the three basic belts in the form of 1 by 4 tables. Because it is used with the insert primitive, there is no need to specify the index value 3; this comes about because of the nature of insertion. This is the seed needed to start the building of the candidate piles. The result of each step is a three-dimensional array of shape **m, n, 4**, where **m** is the number of successful piles so far, initially 3; and **n** is the height of each pile, starting at 1, and ending at 4.

The effect of **s@o** is to insert the dyad **o** between two arrays **c** and **d**, where **c** is shape **k** by 24, where **k** is at most 24, and at least 1; and **d** is the result of the previous step, with shape **m, n, 4** as described above. Each of the **k** rows of **a** is placed on each of the **n** by 4 tables, and then these are joined to form a single table of **k** times **m** items. As noted above, the winnowing of the 24 belts selected by function **a** is accomplished by the J *nub* primitive (**~.**) within **o**, ensuring that there are no duplicate solutions.

The result of `o` is passed on to `s`. I rather like `s`; notice the nice way the rank operator gets used: first rank 3, then rank 2, then rank 1. It uses the phrase `] =/ [: ~.`, to form the outer product equal of the argument with the nub of its ravel. This gives a four-dimensional array of `k` boolean tables, with a table for each of the rows in each solution. It sums the 3-dimensional arrays, which effectively counts the number of appearances in each column of each colour. The tables are raveled, and a mask is formed with a 1 for each row which contains no item greater than 1, and applied to the argument, which gets rid of all of the candidate solutions containing duplicate colours, leaving only those which remain as candidates for final solutions. Here is a sample of how `s` works, using `t`, containing two 3-high stacks:

```

    $t=: 2 3 4 $ 'brwgwwgrrgbgbrwgwwgrrbbw'
2 3 4
  t
brwg
wwgr
rgbg

brwg
wwgr
rbbw
    $t0=: t = / ~. , t
2 3 4 4
    $t1=: +/"3 t0
2 4 4
  t1
1 1 1 0
0 1 1 1
1 0 1 1
0 1 0 2

1 1 1 0
1 1 1 0
1 0 1 1
0 1 1 1
    ] t2=: , "2 t1
1 1 1 0 0 1 1 1 1 0 1 1 0 1 0 2
1 1 1 0 1 1 1 0 1 0 1 1 0 1 1 1
    $t3=: >./"1 t2
2
    ] t4=: 2>t3
0 1

```

This solution can be improved a bit. Suppose I replace the function `x` with a similar function `y` which has the property that it is

executed only if its argument has shape `,6,.` This can be done using J's function power primitive. Then we could do without the `gerund` form altogether. Here is function `y`:

```
y=: (3 1 4$b)&{^:(,6) -: $)
```

Now we can solve the problem with:*

```
(s@o y)/
```

That's all there is to it. J has helped to give a solution significantly shorter than the APL solution.

Now that I had a generic solution, I was able to test it on four different representations: a single character, a symbol, an integer, and a boxed name. All of these are scalars. Here are the four representations of the one-solution set of cubes:

```
] nu=: 4 6 $ 3 1 0 0 2 2 3 2 2 1 3 0 3 3 1 1 ➡
2 0 3 0 1 1 3 2
3 1 0 0 2 2
3 2 2 1 3 0
3 3 1 1 2 0
3 0 1 1 3 2
] ch=: nu{ 'bgrw'
wgbrrr
wrrgwb
wggrb
wbggwr
] sy=: nu{ s: ' blue green red white'
`white `green `blue `blue `red `red
`white `red `red `green `white `blue
`white `white `green `green `red `blue
`white `blue `green `green `white `red
] bn=: nu{ <:_1 '`blue `green`red `white'
+-----+-----+-----+-----+-----+-----+
|white|green|blue |blue |red  |red  |
+-----+-----+-----+-----+-----+-----+
|white|red  |red  |green|white|blue |
+-----+-----+-----+-----+-----+-----+
|white|white|green|green|red  |blue |
+-----+-----+-----+-----+-----+-----+
|white|blue |green|green|white|red  |
+-----+-----+-----+-----+-----+-----+
```

* as is done on page 211. (Ed.)

And the solutions for each:

```
(s@o y)/ nu
1 0 3 0
2 2 1 3
0 1 2 1
3 3 0 2
```

```
(s@o y)/ ch
gbwb
rrgw
bgrg
wwbr
```

```
(s@o y)/ sy
`green `blue `white `blue
`red `red `green `white
`blue `green `red `green
`white `white `blue `red
```

```
(s@o y)/ bn
+-----+-----+-----+-----+
|green|blue |white|blue |
|red  |red  |green|white|
|blue |green|red  |green|
|white|white|blue |red  |
+-----+-----+-----+-----+
```

A set of cubes which gives 22 solutions is:

```
`white `blue `red `blue `white `green
`red `blue `red `blue `green `white
`red `blue `red `green `green `white
`green `white `white `red `blue `red
```

I measured the time needed to produce solutions for each of these representations using two different cube sets, one which had only one solution, and one which had 22 solutions.

Here is a tabulation of the times taken using each of the two test cubes:

	one solution	23 solutions
integer	0.012	0.047
single character	0.010	0.036
symbol	0.012	0.047
boxed name	0.058	0.254

The table shows that the symbol datatype is competitive in time with the single character and integer data types, which is what we hoped would happen. Symbols are a much more efficient datatype to work with than are boxed names. It also shows, which should not be a surprise to anyone, that I have on my desk a computer that is seventy times faster and has much larger memory and disk storage than did the large roomful of computer and disk packs that had seemed so very powerful twenty years ago.

References

- [1] McDonnell, E. E., *The Four Cube Problem*. APL Press, Palo Alto, (1981).
- [2] Backus, J., Can programming be liberated from the von Neumann style? A Functional Style and its Algebra of Programs. *Comm. ACM* 21, 8, (1978-08).

27 Erdős Numbers and Pierce and Engel Expansions

First published in Vector, 17, 4, (April 2001), 111-122.

Revised by Bill Lam, (April 2009).

*If you wish in the world to advance,
Your merits you're bound to enhance,
You must stir it and stump it,
And blow your own trumpet,
Or, trust me, you haven't a chance!*

W. S. Gilbert, Ruddigore

Introduction

This paper discusses a way in which mathematicians are connected to each other, much like the *six degrees of separation* of the play and film of that title by John Guare, or the *Bacon numbers* associated with the film actor Kevin Bacon. It then discusses a paper written by two of these interconnected mathematicians that gives two ways of representing a rational number that were new to me. The paper also had some personal relevance. In discussing the subject matter of the paper I'll define a number of adverbs and a conjunction, the first serious use I've made of these J possibilities. Finally, I'll apply the verbs I define in connection with the almost 4000-year old Egyptian mathematics found on the Rhind papyrus.

Erdős Numbers

I heard a talk a few years ago given by the mathematician Ronald Graham, of the Bell Laboratories, in the course of which he discussed Erdős numbers. Graham had been a frequent collaborator with the mathematician Paul Erdős and even had a room set aside in his home for him. This was to facilitate visits by this eccentric nomad, who, in his later life, drifted from one collaborator to another, with only his suitcase and his mind, greeting his next host with the words, "My brain is open!" [1]. The web site:

<http://www.oakland.edu/enp>

is devoted to Erdős numbers. The next paragraph is copied from that source:

Most practicing mathematicians are familiar with the definition of one’s Erdős number ... Paul Erdős (1913–1996), the widely-travelled and incredibly prolific Hungarian mathematician of the highest caliber, wrote hundreds of mathematical research papers in many different areas, many in collaboration with others. His Erdős number is 0. Erdős’s co-authors have Erdős number 1. People other than Erdős who have written a joint paper with someone with Erdős number 1 but not with Erdős have Erdős number 2, and so on. If there is no chain of co-authorships connecting someone with Erdős, then that person’s Erdős number is said to be infinite.

Here are some data that I gathered from this web site, giving the number of people known to have each of the first several Erdős numbers (the site is updated periodically so the data will change from time to time):

Erdős number	0	1	2	3	4	5
1st kind	1	502	5713	26422	62136	66158
2nd kind	1	229	1969	8602	22668	36112

The row labelled “2nd kind” refers to a more stringent classification, described in the web site as follows:

The entire discussion so far has been based on linking two mathematicians if they have written a joint paper, whether or not other authors were involved. A purer definition of the collaboration graph (in fact, the one that Paul Erdős himself seemed to favor) would **put an edge between two vertices if the mathematicians have a joint paper by themselves, with no other authors ...** Let C' denote the collaboration graph under this more restrictive definition, and let us call the associated path lengths “Erdős numbers of the second kind” (and therefore call traditional Erdős numbers “Erdős numbers of the first kind” when we need to make a distinction).

Since those with Erdős number 2 got their numbers from writing with someone with Erdős number 1, we can get the average *promiscuity number* of the Erdős 1 authors by dividing their total into the total of Erdős 2 authors. The average number of co-auth-

ors for those with Erdős number 1 of the 1st kind is 11.4 and for those of the 2nd kind is a more selective 8.6. All sorts of people have Erdős numbers. You may be surprised to learn that Bill Gates has Erdős number 4, that Sir Francis Crick has Erdős number 7 and that his double helix collaborator Jim Watson has Erdős number 8.

This is the background you need in order to make sense of this message I received from Roger Hui about a year ago:

Subj: Erdős Number
 Date: 12/18/99 6:36:06 AM Pacific Standard Time
 From: RHui@Interlog.Com (Roger Hui)
 To: EEMcD@AOL.Com (E.E. McDonnell)
 CC: KEI@Interlog.Com (Kenneth E. Iverson)

Apparently my Erdős number is 2, having co-authored a paper[2] with Shlomo Moran (during my grad school days in the early 1980's), who co-authored a paper with Erdős himself[3]... Neither you nor Ken are in [the] list [of people with Erdős numbers 0, 1, or 2]. Therefore,... both you and Ken have a Erdős number of 3 or less, having co-authored a paper with me.[4]

This was a surprise, since, although I have written a lot of semi-mathematical papers, I am not really a mathematician. Since there were three co-authors of the paper I wrote with Roger, I had an Erdős number 3 of the first kind, but not of the second kind. Similarly Shlomo Moran has an Erdős number 1 of the first kind, but not of the second, since there were three co-authors of his paper with Erdős.

Thus matters stood until a few months ago, when Roger sent me another message on the same subject:

Subj: Erdős Number
 Date: 11/15/00 9:57:21 AM Pacific Standard Time
 From: rhui@ADAYTUM-CAN.COM (Roger Hui)
 To: EEMcD@AOL.Com (E.E. McDonnell)

According to <http://www.acs.oakland.edu/~grossman/erdoshp.html> Jeffrey Outlaw Shallit wrote a joint paper[5] with Erdős in 1991. Since you wrote at least one paper with Shallit ("Extending APL to Infinity")[6], that makes you an Erdős 2.

Since Erdős died in 1996, I had now reached the highest I can get on the Erdős graph. What is more, since my paper with Shallit had just the two of *us* as authors, and Shallit's with Erdős had just the two of *them*, I now had Erdős number 2 of both the 1st and 2nd kind, and Shallit's Erdős number 1 is also of the 1st and 2nd kind. Among my fellow number twos are such luminaries as Albert Einstein, G. H. Hardy and Donald Knuth. As I chat with my fellow 2's, I appreciate the rather better class of thinkers they are, and wonder now that I was able at all to tolerate those arriviste 3's with whom I rubbed shoulders. I know personally only one other number two, and that is my friend Charles Brenner, about whom you may have read in my letter in the *Technical Correspondence* of the January 2001 *Vector*, vol. 17, No. 3, p. 112. He is today best known as *the forensic mathematician*. See his very interesting website at www.dna-view.com. Charles was just sixteen in 1961 when he and his father, the mathematician Joel Lee Brenner, collaborated on a paper[7]. In 1987 the elder Brenner collaborated on a paper with Erdős[8], thereby promoting his son Charles's Erdős number from infinite to 2. This raises a question. His father's collaboration with Erdős involved a total of six co-authors, so his father's Erdős number 1 is of the first kind only. Charles and his father were the *only* co-authors of their paper. How does this rank Charles? I'd say that the sins of the fathers should *not* be visited on the sons, so that Charles's number 2 is, like mine, of the first *and* second kind. The promiscuity number of Joel Lee Brenner is 37; of Jeffrey Outlaw Shallit is 47; and of Shlomo Moran is 54, all three of them well above average.

I got to know Jeffrey Shallit when he was a young teenager with a gift for mathematics. He was a regular visitor to the IBM Philadelphia Scientific Centre when I worked there. When he went to college at Princeton University, he wrote his bachelor's paper on the mathematical aspects of my design of the complex floor function[9]. Jeff came to work for me at I. P. Sharp Associates in Palo Alto in the summer of 1979, after graduating from Princeton and just before beginning graduate work at the University of California at Berkeley. It was then that he and I wrote our joint paper on APL and the infinite. I wrote the part on infinite values, and Jeff worked out the details of infinite-sized arrays, including a very nice way of displaying them using a diagonal transformation. Upon getting Roger Hui's second message I emailed Jeffrey,

who is now a full professor at Waterloo University in Ontario, Canada, saying how grateful I was to him for making this possible, but said that I didn't feel worthy of the honour. He answered:

Subj: Re: thanks for raising my Erdős number
 Date: 11/19/00 5:35:42 AM Pacific Standard Time
 From: shallit@graceland.uwaterloo.ca (Jeffrey Shallit)
 To: Eemcd@aol.com

Au contraire, it's I who should be grateful to you. The paper that I wrote with Erdős was on expansions of the form

$$1/a - 1/(ab) + 1/(abc) - \dots$$

which I learned about from you in your APL article on "Spirals and Time"[10]. If I hadn't had the opportunity to learn from you in Philadelphia and later in Palo Alto, I would never have explored this interesting topic!

Best, Jeff

This, unlike the essentially shallow glamour of my Erdős number, was something that I felt I could take legitimate pride in. The "Spirals and Time" article was a very early one of my articles. It gives me no little gratification when someone tells me they've enjoyed one. This message gave me an exceptionally large boost because Jeff had actually learned from it something useful to him professionally, and may even, as he suggests, have led to his involvement with Erdős. By the way, it had another satisfying repercussion, which I heard about from someone in Denmark who had shown my article to his fiancée. The article noted that the Gregorian calendar intercalation scheme had leap years in a 4-100-400 year cycle, but could be made exactly accurate if it were extended to a 4-100-400-3200-86400 cycle. The fiancée told Henry that on reaching the conclusion, where I noted that 86400 was also the number of seconds in a day, she "had an intellectual orgasm."

Notice that successive terms in the Gregorian sequence are multiples of the preceding term. That is, 100 is 25×4 and 400 is 4×100 . In the extension suggested in my article, 3200 is 8×400 and 86400 is 27×3200 . The sequence of multipliers 4 25 4 for the Gregorian sequence and 4 25 4 8 27 can each be used to determine the average length of the year in each system. In the next section I'll discuss how this may be done. For now, I'll only point out that

the current method of adding leap *seconds* to a year is sufficient to make it unnecessary to do any extensions to the Gregorian sequence, so discussions (like this) concerning changing or extending the current Gregorian scheme are purely academic.

Pierce and Engel Expansions

That brings us, at last, to the *At Play With J* part of this paper.

The expansion in Shallit's message lies behind the design of the Gregorian calendar.

$$1/a - 1/(ab) + 1/(abc) - \dots$$

In the case of the Gregorian calendar the values *a*, *b*, and *c* are 4, 25, and 4.

$$'a \ b \ c' =: 4 \ 25 \ 4$$

The product scan (**/*) of this list gives the cumulative products, in this case defining the intervals in years when to intercalate and when not: every fourth year but not every hundredth year unless it is also a four-hundredth year.

$$\begin{aligned} &] \ m =: */\ a, b, c \\ 4 \ 100 \ 400 \end{aligned}$$

We reciprocate these values:

$$\begin{aligned} &] \ n =: \% \ x: m \\ 1r4 \ 1r100 \ 1r400 \end{aligned}$$

The alternating sum of this gives the part of a day *p* which, when added to 365, gives the average length of the year in calendar days in the Gregorian calendar.

$$\begin{aligned} &] \ p =: -/ \ n \\ 97r400 \end{aligned}$$

It also gives the number of leap years (97) in the cycle (400). One gets 97 this way:

$$\begin{aligned} &400 \% 4 \ 100 \ 400 \\ 100 \ 4 \ 1 \\ &- / 100 \ 4 \ 1 \\ 97 \end{aligned}$$

The number of days in 400 years gives the length of the Gregorian cycle in days, which keeps repeating as the millennia roll on.

$$\begin{aligned} &400 * 365 + 97r400 \\ 146097 \end{aligned}$$

Shallit's *curriculum vitae* lists three papers, each bearing on the topic of *Pierce expansions*. These are not widely known, but they and the companion *Engel expansions* are the topics of the rest of this article. What are they? They are algorithms for converting a rational number into a series of integers, which, much like a continued fraction, give a way of representing a rational. The algorithm for Pierce expansions is described by Shallit in his paper on their metric theory as follows:

[Pierce expansion algorithm]: Given a real number x in $(0, 1]$, this algorithm produces the sequence of a_i such that $x = \{a_1, a_2, \dots\}$.

P1. [Initialize]. Set x_0 to x , set i to 1.

P2. [Iterate]. Set a_i to floor $(1/x_{i-1})$; set x_i to $1 - a_i x_{i-1}$.

P3. [All done?]. If $x_i = 0$, stop.

Otherwise set i to $i + 1$ and return to P2.

Jeff points out that if x is a rational p/q , step P2 replaces p by $q \bmod p$, that this is less than p , and so eventually x will become 0, and the algorithm will terminate.

Several friends of mine asked if I could find out what Shallit's joint paper with Erdős was about. It dealt with Pierce expansions, but also with something called Engel expansions. All I could find out about them was that they involved sums of reciprocals, as opposed to alternating sums, but I was unable to work out the details on my own, so asked Jeffrey for help. His reply:

From: shallit@graceland.uwaterloo.ca (Jeffrey Shallit)
To: Eemcd@aol.com

Do you read Maple? Here is a maple program to compute the engel expansion of x up to the first n terms:

```
engel := proc(x,n) local xp, z, k;
xp := x;
z := [];
k := 0;
while ((k <= n) and xp <> 0) do
k := k+1;
y := ceil(1/xp);
z := [op(z),y];
xp := y*xp - 1;
od;
z;
end;
```

Basically you take ceiling of $1/x$, and that's the next output. Then you multiply that by your current x and subtract 1. Then continue.

Jeff

I'm not familiar with Maple, but I did manage to feel my way through this code. Having digested it, I concluded that the Pierce and Engel expansions were closely related. The Pierce list of integers implies an alternating sum, and the Engel list of integers implies a direct sum. This simplified my work in turning them into J, since one pattern would do for both.

To begin with, I changed Jeff's approach in two important ways: first, to simplify termination control, I would work only with rational arguments; second, instead of having essentially two main variables, a continually modified rational and a continually lengthening list of integers, I would combine the two, beginning with a scalar rational, and successively modifying this in two ways: first appending a tail, and then modifying the head. The tail extension would be obtained by applying an integer function to the reciprocal of the current head value, and appending this; the *floor* ($<.$) for Pierce expansions, and the *ceiling* ($>.$) for Engel expansions. The head modification would be obtained by applying a subtracting function to the product of the head and the tail, and replacing the head with this; the *one minus* ($-.$) function for Pierce expansions, and the *minus one* ($<:$) function for Engel expansions. Similarly, in computing the inverses to these functions, obtaining the rational from a list of integers, the same pattern would be used, with minus ($-$) for Pierce contractions, and plus ($+$) for Engel contractions. This is summarized in the following table:

	Pierce	Engel
Tail	$<.$	$>.$
Head	$-.$	$<:$
Inverse	$-$	$+$

First I defined an adverb which would serve for both tail extensions:

```
BT=: 1 : '( , u @ % @ { . ) y '
```

where *u* stands for the appropriate integer function to be used in its place, floor or ceiling. It works by applying the appropriate integer function (*u*) to the reciprocal (%) of the head ({ .), then appending it (,). Here's how this works with each form of expansion:

```
<. BT 97r400
97r400 4
>. BT 97r400
97r400 5
```

Next comes an adverb for both head modifications:

```
BH=: 1 : '((0 } ~)u @ ({ . * { :))y '
```

This replaces the head (0 } ~) with the appropriate subtracting function (*u*) applied to the product of the tail and the head ({ . * { :). For example:

```
-. BH <. BT 97r400
3r100 4
<: BH >. BT 97r400
17r80 5
```

Next, these are combined in a conjunction that will allow a step function to be defined for both expansions:

```
BS=: 2 : '(u BH)@(v BT)y '
```

Here the *u* and *v* stand for the left and right function arguments to be used. For example:

```
-. BS <. 97r400
3r100 4
<: BS >. 97r400
17r80 5
```

A control structure is needed to allow the steps to be applied as often as necessary. This requires a sequence of two uses of the power conjunction; the first to control termination, with a right argument which gives the signum of the head (* @ { .). This will be *one* for any nonzero head value (I assume the argument is always positive), which allows the function to be applied; when the head eventually becomes *zero*, as it must since it is continually

being reduced, the function will not be applied, and the result will be the same as the argument. The second use of the function power conjunction will cause the steps to be applied to the limit, that is, until two successive results are equal. The convention proposed by Iverson[11] is that positive infinity (∞) be used to describe application of a function to the limit. Nicely enough, this proposal was seconded by Shallit and me in our paper on infinities, and it is now part of J.

```
CS=: 1 : 'u (^:(* @ {.))^: _'
```

We can use CS with both steps:

```
(- . BS <.)CS 97r400
0 4 33 100
(<: BS >.)CS 97r400
0 5 5 16
```

These results show that the head is indeed zero. The zero is extraneous, so now we define two functions that yield just the needed Pierce and Engel expansion from a rational. We only have to behead ($\{.$) the results we just got:

```
PR=: 3 : '}. @((- . BS <.)CS)y'
ER=: 3 : '}. @(<: BS >.)CS)y'

PR 97r400
4 33 100
ER 97r400
5 5 16
```

Let's define the functions inverse to PR and ER and check whether each expansion contracts to 97r400. The method is essentially the same for both, so again we define an adverb that applies the appropriate subtraction function to insert ($u /$) between the result of reciprocating ($\%$) and product scanning ($* / \backslash$) our lists of integers:

```
RB=: 1 : 'u / * / \ % y'
```

and this makes the inverses easy to define:

```
RP=: - RB
RE=: + RB
```

So now we use each of them:

```
RP PR 97r400
97r400
RE ER 97r400
97r400
```

So the expansions contract properly. We saw above that the list $4 \ 25 \ 4$ contracted to $97r400$, and now have verified that the list $4 \ 33 \ 100$ does as well. In other words, both intercalation schemes will give an exact Gregorian year. The difference is that the cycle for the present Gregorian year is 400 years; for a $4 \ 33 \ 100$ calendar the cycle is 13,200 years. For each, the resulting average year length is 365.2425 days.

By the way, the result $97r400$ is given rather than the decimal equivalent 0.2425 because the results of PR and ER are both rationals, the same type as their arguments. See what happens if one just types in the numbers:

```
RP 4 33 100
0.2425
RE 5 5 16
0.2425
```

The functions PR and ER will work properly only when applied to rational arguments.

Engel Expansions and Gypsy Math

The very ancient document called the *Rhind papyrus* includes a table of all fractions of the form $2/n$ from $2/3$ through $2/101$, and for each gives a list of from two to four unit fractions that sum to it. For example,

```
2r3   = 1r2 + 1r6
2r61  = 1r40 + 1r244 + 1r488 + 1r610
```

The Engel expansions of rationals of the form $2/p$ for the first four odd primes give the same results as those listed in the Rhind papyrus:

```
BE=: [: % */\

(BE @ ER)"0 [ 2r3 2r5 2r7 2r11
1r2 1r6
1r3 1r15
1r4 1r28
1r6 1r66
```

This is not generally true, however. For example, the Rhind values for $2r21$ are $1r14$ and $1r42$, whereas the rationals given by the Engel expansion are $1r11$ and $1r231$. However, I am now in a

position to bring some unfinished business to a close, that I've left in abeyance since June, 1981. In my last column as Recreational APL editor for *APL Quote Quad*, in a section called "Gypsy Math" I wrote:

The Rhind papyrus ... shows, for each odd integer from 3 to 101, several integers whose reciprocals sum to $2 \div \omega$. For example, $(2 \div 17) = + / \div 9 \ 153$. Write a function F such that, for odd positive argument ω ,

$(2 \div \omega) = + / F \omega$

$2 = \rho F \omega$

$\sim \omega \in F \omega$

Thus, $F \ 17 \leftrightarrow 9 \ 153$.

I'm impressed by the fact that twenty years ago I knew how to give unit fraction results for Rhind fractions such as $2r17$ without knowing anything about Engel expansions. However, the Engel expansion is completely general, and will handle any positive rational less than 1. Our Egyptian predecessors probably used a simpler formula which we would write in J notation as:

```
Egypt=: [: */\ (,~ -:@>:)
Egypt"0 [ 3 5 7 11
2 6
3 15
4 28
6 66
```

Postscript

Erdős, by-the-bye, has a Bacon number of 4. Schechter explains how this came about in his book on Erdős:

A mathematician and sometime actor named Gene Patterson appeared briefly in the 1993 documentary about Erdős, *N is a Number*. Patterson also had a role in *Box of Moonlight* with John Turturro, who was in *The Color of Money* with Tom Cruise, who appeared in *A Few Good Men* with Kevin Bacon.

And Jeffrey Shallit is the proud possessor, in addition to his Erdős number of 1, of an *Elvis number* of 3. If you can't guess what this is, you can read all about it at his web site:

<http://www.math.uwaterloo.ca/~shallit>

References

- [1] Schechter, Bruce, *My Brain Is Open, The Mathematical Journeys Of Paul Erdős*. Simon & Schuster, New York, (1998).
- [2] Ibarra, O., Moran, S., Hui, R. K. W., A Generalization of the Fast LUP Matrix Decomposition Algorithm and Applications. *Journal of Algorithms* 3, (1982), 45-56.
- [3] Erdős, P., Linial, N., Moran, S., Extremal Problems on permutations under cyclic equivalence. *Discrete Math.* 64, (1987), 1-11.
- [4] Hui, R. K. W., Iverson, K. E., McDonnell, E. E., Whitney, A. T., "APL\ ". *APL90 Conf. Proc.*, 192-200.
- [5] Erdős, P., Shallit, J., New bounds on the length of finite Pierce and Engel series. *Séminaire de Théorie des Nombres de Bordeaux* 3, (1991), 43-53.
- [6] McDonnell, E. E., Shallit, J., Extending APL to Infinity. *APL80*, Noordwijkerhout, 123-132.
- [7] Brenner, C. H., Brenner, J. L., The popularity of small integers as primitive roots. *Numer. Math.* 4, (1962), 336-342.
- [8] Brenner, J. L., Beasley, L. P., Erdős, P., Szalay, M., Williamson, A. G., Generation of alternating groups by pairs of conjugates. *Period. Math. Hungar.* 18, (1987), 259-269.
- [9] McDonnell, E. E., Complex Floor. *APL Congress 73*, Copenhagen.
- [10] McDonnell, E. E., Spirals & Time. *APL Quote Quad* 7, 4, (Winter 1977), 20-22.
- [11] Iverson, K. E., Operators and functions. RC 7091, IBM Corp., Yorktown Heights, NY, (1978).

28 Boggle

First published in Vector, 18, 1, (July 2001), 91-102.

Revised by Tracy Harms, (April 2009).

Boggle

I suppose many of you have played as a child with a set of blocks, wooden cubes about an inch and a half on a side, with pictures, letters, numbers, and designs on the faces. Did you ever set the alphabet faces in a row to spell a word? Suppose you had such a set of blocks in which all of the faces had letters on them, and that you had a tray divided by partitions to form rows and columns, giving cells into which the blocks just fitted. Now if you jumble up your blocks in a bag, then take out a block at a time and with your eyes closed, put it securely in one of the cells at random until they are all full, you will find when you then open your eyes that the letters that are face up will be in all different orientations. Now if you are given the task of finding among these as many words of four or more letters formed among blocks that are connected either by an edge or a corner within three minutes, you will have some idea of how to play a word game I very much like.

The game comes in two forms; the original game, the one I cut my teeth on, is called *Boggle*TM, and is played on a 4-by-4 tray. The other game, which I now favour, and which appeared several years after Boggle was introduced, is called *Big Boggle*TM, and is played on a 5-by-5 tray. The blocks are miniature, about five-eighths of an inch on a side and made of plastic, as is the tray. There is also a clear plastic dome which fits snugly over the tray, permitting the whole to be turned upside down, shaken, and turned upright, so that with a little jiggling each block nestles upright in one of the cells. The game comes with a three-minute egg-timer and a little sheet of instructions. The letters on each of the sixteen Boggle cubes and the 25 Big Boggle cubes are given by the columns of the tables below. They are shown as lower-case here, but in the game they appear as capitals. The letter shown as 'q' is actually the di-graph 'qu', and if used in a word counts as two letters:

a a a a c d d d e e e e e h h
a b c f o i e e i e e h i l i l
e b h f o m i l s g i o r m n
e j o k t o l r t h n t s t n n
g o p p t t r v t n s v s t q r
n o s s w u x y y w u w t y u z

a a a a a a a b c c c c d d d e e e f g i n o
f d e a e a e a j c e e e h d h n m i i o p o o
i e e a g e e f k e i i i l l h s o i p r r o o
r n e f m e g i q n l i p n n n l s t i r r r t t
s n e r n e m r x s p l s o o o s t t s v r u t
y n m s n e u s z t t t t r r t r u t t y w y w u

Here are the rules supplied with the game:

Object: To list, within 3 minutes, as many words of the highest point value as you can find among the random assortment of letters in the cube grid.

Preparation: Each player should have a pencil and a piece of paper. Drop the letter cubes into the dome and place the grid, open side down, over the dome. Turn the domed grid right-side up, vigorously shake the cubes around, and maneuver the grid until each cube falls into place. Then, as one player removes the dome, another player starts the timer.

Playing: When the timer starts, each player searches the assortment of letters for words of *four letters or more*. When you find a word, write it down.

Words are formed from *adjoining letters*. Letters must join in the *proper sequence* to spell a word. They may join horizontally, vertically or diagonally to the left, right, or up-and-down. No letter cube, however, may be used more than once within a single word.

Type of words allowed: The only words that are allowed are those that can be found in a standard English dictionary. You may look for any type of word — noun, verb, adjective, adverb, etc. Plural nouns are acceptable as are all verb tenses. Words within words are also allowed, e.g., *master*: mast, aster.

Type of words not allowed: Proper nouns, abbreviations, contractions, hyphenated words, and foreign words that are not in an English dictionary.

Scoring and winning: When the timer runs out, everyone must stop writing. Each player in turn then reads aloud his or her list of words. Any word that appears on more than one player’s list must be crossed off *all* lists, including that of the reader. The same word found by a player in different areas of the grid may not be counted for multiple credit.

After all players have read their lists, each player scores his or her remaining words, differing point values accorded to the words according to their lengths, as follows:

no. of letters	4	5	6	7	8+
Points	1	2	3	5	11

The winner is a) the player whose words have earned the most points, or b) the first to reach 50 points, 100 points or whatever score is considered by all to be a reasonable target.

I usually play until at least one player has reached or passed 100 points. I've played the game with three or four players, but prefer the two-person game. My wife and I have established several additional house rules. Since I frequently wrote down spurious words, my wife insisted that there be a penalty for such. Thus, any word may be challenged. If it is not found in the dictionary the player is given a score of -1 for it. If it *is* in the dictionary, the player gets an additional point for it. We began by using our huge unabridged *Merriam-Webster* dictionary, but this, my wife claimed, gave me an unfair advantage, since I frequently wrote down archaic Scottish words and the like that were in this dictionary but not in smaller ones. We then began using the *Concise Oxford Dictionary (COD)*, but had to give up on that, too, since it favoured English words and English spellings. My wife was tired of me putting down words like *twee* and *nous* that are unknown on our side of the Atlantic. We now use the *American Heritage Dictionary (AHD)*, Ken Iverson's favourite. Here is a sample grid:

```

t i n e
n i n t
o c n a
r e t l

```

These letters are shown in normal position; in practice they can have any of the four possible orientations. Try your hand at finding words in this grid. Remember, you have just three minutes. You can see the words I found, unaided by computer, at the end of this paper.

The Problem

Twenty years ago, when I was Recreational APL editor of *APL Quote Quad*, I received a letter from Robert Ashworth, of Carbon-dale, Illinois, asking if I could write a column on the Boggle game. I was agreeable to the extent of posing these problems to my readers (this was before Big Boggle was born, so assumes the smaller Boggle situation):

- 1. Find the number of paths of length 4 in the grid.
- 2. Find the number of paths of length 5.
- 3. Write a suite of functions that, given a 4-by-4 character table, finds all words of length 4 and 5, using the rules of Boggle. Assume the existence of two tables w4 and w5 containing all the acceptable words of length 4 and 5, respectively.

I thought this was the most difficult problem I'd ever proposed. Furthermore, at the time I had only vague ideas of how to go about solving it. Recently, nagged by this unfinished business, I revisited the problem, with a degree of success.

A Boggle path is a sequence of distinct connected cells, connected in the Boggle sense. An interior cell is connected to the eight surrounding cells. An edge cell, not on a corner, is connected to the five surrounding cells. A corner cell is connected to the three surrounding cells. It may be suitable at times of to use the cell's list indices:

```
] m=: i.4 4
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
```

and at other times their row-column indices:

```
] q=: <"1 (4 4# : m)
+---+---+---+---+
|0 0|0 1|0 2|0 3|
+---+---+---+---+
|1 0|1 1|1 2|1 3|
+---+---+---+---+
|2 0|2 1|2 2|2 3|
+---+---+---+---+
|3 0|3 1|3 2|3 3|
+---+---+---+---+
```

Normalizing any 3-by-3 portion of the row-column index grid by subtracting the central item from each of the items, shows that two cells are connected if the maximum magnitude of the difference of their row-column indices is 1.

```

      (] -&. > (<1 1)"_ { ] )3 _3{.q
+-----+-----+-----+
|_1 _1|_1 0|_1 1|
+-----+-----+-----+
|0 _1 |0 0 |0 1 |
+-----+-----+-----+
|1 _1 |1 0 |1 1 |
+-----+-----+-----+

```

The only way I know to find the number of paths for different cases is by constructing the paths. So to solve problems a and b above implies finding the paths themselves.

An easy but expensive way is to find a superset of the paths, by taking all the combinations of sixteen things taken four at a time, that is, $4!16$ or 1,820, then get each of the 24 permutations of every combination, giving $(!4)*(4!16)$ or 43,680 four-item lists, then select from the table all rows giving Boggle-connected paths. The verb `comb` is due to Roger Hui, and it and its component parts are given at the end of this paper.

The function `paths` has syntax `r=: n paths k` and gives the paths of length `n` in a `k`-by-`k` grid.

```

paths=: dyad define
NB. find all the paths of length x in a grid of
NB. size (y*y). px is a table of all the
NB. permutations of length x
    px=. (i.!x)A.i.x
NB. cxn is a table of all the combinations of
NB. y things taken x at a time.
    cxn=. x comb *: y
NB. pc is a table of all the permutations of each
NB. of the combinations.
    pc=. ,/px{"2 1 cxn
NB. (i{mpc) is 1 if (i{cxn)is Boggle-connected
NB. and 0 otherwise.
    mpc=. y okf pc
NB. cxno is all paths of length x in a y*y grid.
    cxno=. /:~ mpc#pc
)

```

This is the `okf` function:

```
okf=: 13 : '1=>./,|2-/\'(2#x)#:'y'"1
```

Given a list `y`, this takes the row-column representation `((2#x)#:y)` of each item, the difference of pairs of successive representations `(2-/\\)`, their magnitudes `(l)`, ravel these `(.)`, finds their maximum `(>./)`, compares this to 1 `(1=)`, yielding 1 if the list is Boggle-connected, and 0 otherwise. For example, the number of paths of length 3 in a 4-by-4 grid is given by:*

```
#3 paths 4
408
```

Here are four successive items from `pc` after running `#4 paths 4`:

```
37 38 39 40{pc
2 0 4 1
2 1 0 4
2 1 4 0
2 4 0 1
```

And here is the result of applying `4 okf` to each of these rows:

```
4 okf 37 38 39 40{pc
0 1 1 0
```

If you look again at `m` you can verify that `2 0 4 1` and `2 4 0 1` are not Boggle-connected, but `2 1 0 4` and `2 1 4 0` are:

```
m
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
```

Don't bother to use this version of `paths`. It takes an unbearably long time as the path lengths get just a little larger. It is intended only to let you know how my thinking was going.

An easier way to get the paths is to build them up starting with the n^2 paths of length 1, and extending these only with promising items. At this point I showed my astuteness by sending a message to Roger Hui explaining what I was doing, and asking him if anything bright in a combinatorial way occurred to him. I received, in rapid succession, four replies, written while he was babysitting his son Nicholas as he was in the middle of moving from Toronto

* Executing `paths` needs the verb `comb` to be already defined. See page 239. Furthermore to make `pc` accessible in the next code example, the definition of `paths` must be changed to assign `pc` globally (`=:`), not locally (`=.`):

```
pc=: ,/px{"2 1 cxn
```


to Vancouver. I'll pass over the first three messages because, as usual with Roger, one idea suggested a better idea. Here is his last effort:

```
rimb    =: _1: ,. (_1: , ] , _1:) ,. _1:
tileb   =: 3 3 &(,;._3)
nborsb  =: (4 1 4#1 0 1)&#"1 @ (,/ ) @ tileb @ rimb ➡
@ i. @ ,~
initb   =: ,. @ i. @ *:
extendb =: 8&#@] ,. [: , {:"1@] { [
testb   =: *./"1@(0&<:) *. i.@{:@$ -: "1 i."1~
stepb   =: (testb # ] )@extendb
pathb   =: 4 : '(nborsb x) stepb^(y-1) initb x'
```

The `pathb` dyad has syntax `z=: n pathb k`, and yields a table with `k` columns, with rows giving all the paths of length `k` to be found in an `n`-by-`n` grid. It begins by using `initb` to form a seed table having all possible starting cells of paths, namely, a single column with values `i. *: n`. Each use of its `step` dyad, beginning with the seed table, extends its argument, a table of paths of length `k-1`, to form the table of paths of length `k`. The `stepb` dyad requires as its left argument a table with eight columns and as many rows as there are cells in the grid. Each row contains a list of all the neighbours of each cell. In the case of edge cells, which have only five neighbours, and corner cells, with only three, the lists are filled out with `_1` values. The `nborsb` monad builds this table by forming an `n`-by-`n` grid with `(i. @, ~)` and using the `rimb` monad to border this with `_1`s. This is tessellated into raveled 3-by-3 squares with the `tileb` monad. `,` and `(,/)` makes the individual tables into one large table. To complete the table the central column is deleted with `(4 1 4#1 0 1)&#"1`.

Here is what this table for the 4-by-4 case looks like:

nborsb 4							
_1	_1	_1	_1	1	_1	4	5
_1	_1	_1	0	2	_4	5	6
_1	_1	_1	1	3	5	6	7
_1	_1	_1	2	_1	6	7	_1
_1	0	1	_1	5	_1	8	9
0	1	2	4	6	8	9	10
1	2	3	5	7	9	10	11
2	3	_1	6	_1	10	11	_1
_1	4	5	_1	9	_1	12	13
4	5	6	8	10	12	13	14
5	6	7	9	11	13	14	15
6	7	_1	10	_1	14	15	_1
_1	8	9	_1	13	_1	_1	_1
8	9	10	12	14	_1	_1	_1
9	10	11	13	15	_1	_1	_1
10	11	_1	14	_1	_1	_1	_1

The table for the 5-by-5 case is similar, but with 25 rows instead of 16.

At each step, the last item of each of the current paths is used to select the appropriate row from the neighbours table, that row is replicated eight times, and each neighbour is appended to one of the eight replicated rows, using the `extendb` dyad. The `testb` monad removes illegal rows from this extended table, ensuring that no item is duplicated, and no `_1`s are present in the result. This process continues for `k-1` steps, at the end of which we have obtained the desired table.

The `pathb` function executes `4 pathb 4` in 0.02 seconds, and `4 pathb 5` in 0.09 seconds on my 233MHz computer. With its help I found the answers to problems 1 and 2:

```
#4 pathb 4
1764
#4 pathb 5
6712
```

The time for my `pathb` function to do `4 pathb 4` is 43.4 seconds. I don't have the patience to try cases involving longer paths.

Now it's time to solve problem c. I have carried about with me for about 15 years word lists from the *American Heritage Dictionary*. The original data came from the publisher Houghton-Mifflin on a single file on a magnetic tape which Joey Tuttle had mounted on a tape drive in the Toronto machine room of I. P. Sharp Associates

and read into the Amdahl V8 then in use there. He processed this file in several ways, the one of most use to me being sorted lists of words all of the same length, which I have named Words02 through Words26. These are now resident on a Macintosh computer in my home. With the tools you've seen developed, you could probably arrive at a solution yourself. Assume that you have the *b*, the ravel of the letters in the grid, *p* the table of paths, and *w* the list of words, all for a given length. Then

$$r =: /: \sim \sim. (w \ e. \ p \ \{ \ b \) \# \ w$$

will give *r*, a sorted table of all the words of a given length, with no duplicates, satisfying a legal path on the grid.

Before showing the results I obtained using the phrase above, I give first the words I found unaided by computer in this grid:

t i n e
n i n t
o c n a
r e t l

ante	alter	cental
cent	inner	lancer
coin	inter	lancet
core	lance	lanner
lane	later	recoin
late	nance	rennet
nice	nicer	rental
nine	octal	tanner
rent	renal	tannin
tine		tinner
tint		

All of these are in *COD*. All but "nance" and "recoin" are in *AHD*.

And here are those found by the computer:

<i>anne</i>	alter	alnico
ante	ancon	cental
cent	anent	cetane
cero	anion	encina
coin	<i>conti</i>	encore
core	<i>creon</i>	innate
<i>etna</i>	enate	lancer
icon	inane	lancet
lane	inner	lanner
late	inter	latent
neon	lance	nocent
<i>nero</i>	later	octane
nice	nicer	octant
nine	renal	rectal
<i>nore</i>	<i>renan</i>	rennet
once	renin	rennin
<i>reni</i>	rente	rental
rent	tater	tanner
<i>tate</i>	tinct	tannic
tent		tannin
tine		tenant
tint		tinner

The results of me versus the computer are: 4-letter words, 11 vs 16; 5-letter words, 9 vs 16; and 6-letter words, 10 vs 22. The second table shows a defect of my word collection. The tape we got from Houghton Mifflin includes biographical and geographical entries, and these have not been removed. I’ve put in italics those words, which, by Boggle rules, are not legal words. Also, *COD* has some words not in *AHD* and vice-versa. For example, “cero” (a western Atlantic fish) is in *AHD*, but not in *COD*, and “recoin” is in *COD* but not *AHD*.

There are longer words in the list. How many of you found “continent”? And “continental”? And the 16-letter behemoth “intercontinental”? Of course, I contrived this case, choosing a suitable looking word from my Word16 file, in order to fill a 4-by-4 grid completely.

Here is an incomplete table of the number of paths of given lengths in grids from size two to five:

paths=: 0 : 0

The number of paths of length k connecting distinct adjacent points in a square grid of $n * n$ points:

$k \backslash n$	2	3	4	5
1	4	9	16	25
2	12	40	84	144
3	24	160	408	768
4	24	496	1764	3768
5		1208	6712	17280
6		2240	22672	74072
7		2984	68272	296360
8		2384	183472	
9		784	436984	

I've been unable to find the law of this table. Row 1 are the squares, of course. Row 2 is four times the alternate triangular numbers (3 10 21 36). Beyond that deponent witnesseth not. The columns headed 2 and 3 are complete, but even with Hui's much more efficient functions, larger cases for columns 4 and 5 are still unattainable with my computer (and my patience).

There are two ways the task can be made more efficient: 1) instead of using `pathb`, which necessitates going back to the beginning for each case computed for a given grid size, the previous results can be stored, allowing case k to be computed by using the `stepb` function on the $k-1$ result; and 2) by taking advantage of the symmetries of the square grid. The number of paths of a given length proceeding from any corner point are the same; similarly for symmetrically located edge points and interior points. The symmetries are evident if the number of paths beginning from each grid point are displayed. For example, here are the number of paths from each point for a 4-by-4 case and a 5-by-5 case:

```
4 4$#/.~{. |:4 pathb 5
322 435 435 322
435 486 486 435
435 486 486 435
322 435 435 322
```

```
5 5$#/.~{. |:5 pathb 6
1874 2752 2998 2752 1874
2752 3524 3672 3524 2752
2998 3672 3784 3672 2998
2752 3524 3672 3524 2752
1874 2752 2998 2752 1874
```

In general, for a grid of side n , containing n^2 points, there are only $g(n)$ distinct numbers of paths, where g is

```
g=: 2&!.@>:@>.@-:
```

Here are the results of g applied to grid sizes 1 through 8:

```
(,:g) 1+i.8
1 2 3 4 5 6 7 8
1 1 3 3 6 6 10 10
```

So that, for the 4-by-4 case, only three values have to be computed, not 16, and for the 5-by-5 case, only six, not 25. The total number of cases for a given path length k in a grid of size n can be obtained by an inner product:

```
4 4$#/.~{.!:4 pathb 4
75 109 109 75
109 148 148 109
109 148 148 109
75 109 109 75

4 8 4 +/ . * 75 109 148
1764

#4 pathb 4
1764
```

When the actual paths are needed, the additional cases can be obtained from the abbreviated tables by using the appropriate indices to select from them, the indices being the permutations obtained from the ravel of the rotations and reversals of the table $i.(n.n)$.

Here is how this is done:*

```
NB. get neighbours table
n4=: nborsb 4
NB. form monad to step previous case by bonding
NB. steps left argument
f=: n4 & stepb
```

* Both suites: Hui's (for `comb`), and Rotate/reflect (for `d8`), are needed to make this code sample work. See the Appendix on page 239.

```

NB. initial case for abbreviated situation
] ip=: ,. 0 1 5
0
1
5
NB. Get all paths for abbreviated argument
$p42=: f ip
16 2
NB. Only 16 cases of 84 obtained using full arg
$4 pathb 2
84 2
NB. Use phrase 7.b.d8 to obtain desired permuted
NB. lists of all cell numbers
] allpcn=: , "2 (i.8)d8"0 2 i.4 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
12 8 4 0 13 9 5 1 14 10 6 2 15 11 7 3
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
3 7 11 15 2 6 10 14 1 5 9 13 0 4 8 12
12 13 14 15 8 9 10 11 4 5 6 7 0 1 2 3
15 11 7 3 14 10 6 2 13 9 5 1 12 8 4 0
3 2 1 0 7 6 5 4 11 10 9 8 15 14 13 12
0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15
NB. Get all len-2 paths (with possible duplicates)
$q42=: ,/p42{"1/allpcn
128 2
NB. Get ordered set of all length-2 paths
$s42=: /:~ ~. ,/ q42
84 2
NB. Compare ordered set to long-winded but
NB. accurate set
(4 pathb 2) -: s42
1

```

Appendix

Hui's combinations suite:

```

startc =: i.@-.@-
countc =: <:@[ ! <:@[ + |.@startc
indexc =: ;@:((i.-])&.>)
recurc =: (countc#startc) ,. (indexc@countc{comb ➡
&.<:)
testc =: *@[ *. <
basisc =: i.@(<: , [)
comb =: basisc`recurc @. testc

```

Rotation/reflection suite (from *J Phrases*, 7.B.d8):

m0=:]	NB. Identity
m1=: m6@m7	NB. Three-o'clock rotation
m2=: m4@m6	NB. Six-o'clock rotation
m3=: m4@m7	NB. Nine-o'clock rotation
m4=: .@]	NB. Horizontal reflection
m5=: m2@m7	NB. Counterdiagonal reflection
m6=: ."_1@]	NB. Vertical reflection
m7=: : @]	NB. Diagonal reflection
d8=: m0`m1`m2`m3`m4`m5`m6`m7 @. [
NB. i d8 y gives mi y (all rotates and reflects)	

29 The Counterfeit Coin Problem

First published in Vector, 18, 3, (January 2002), 93-103.

Revised by Bill Lam, (April 2009).

The Counterfeit Coin Problem, unlike many mathematical puzzles, is not a creation of ancient times nor of the 19th century, and does not appear in the classical works of Loyd, Dudeney, Ball, or Kraitchik. It springs from a problem posed by E. D. Schell in the January 1945 issue of the *American Mathematical Monthly*:

You have eight similar coins and a beam balance. At most one coin is counterfeit and hence underweight. How can you detect whether there is an underweight coin, and if so, which one, using the balance only twice?

Try solving this. I give my solution at the end of this paper.

Most of the solutions I have seen for this kind of problem give an initial allocation of the coins to the balance's pans, and on the basis of the weighing give another allocation, and so on. In 1978 J. G. Mauldon published an IBM Research Report RC 7476, in which he gave a solution in which the weighings were predetermined, not a result of a previous weighing, entitled "Strong Solutions for the Counterfeit Coin Problem". His statement of the problem is generalized from Schell's. I'll call this the first problem:

Given C coins, of which it is suspected that one (at most) is counterfeit (either underweight or overweight), it is required, in at most W weighings on an ordinary beam balance, to identify the counterfeit (if present) and to determine whether it is heavy or light.

He defines a strong solution to which "The choice and distribution of coins for each weighing is to be independent of the other weighings." In support of his method, he proves the theorem that "if the pair (W, C) admits a solution at all, then it admits a strong solution." Thus, no other solution can be better than his strong technique. His method is array oriented, and he gives a suite of APL direct definition functions which give a solution to the problem, acknowledging "his indebtedness to the encouragement and valuable advice of Dr. Kenneth Iverson." He also gives solutions for a second problem, where we are given that exactly one coin is counterfeit, and we are not required to specify whether it is heavier or lighter, but merely to identify it, and a third problem in

which in addition to the given set of coins, we are allowed to incorporate into the weighings an arbitrary number of coins known to be of standard weight. I'll only discuss the first problem. His solutions to all three problems are similar.

He defines a solution as a table of weighings showing the allocation of coins at each weighing. In table (A) are his solutions for all nine cases for which W is 3. These are for values of C from 4 through 12, inclusive.

C = 4, 7, 10				C = 5, 8, 11				C = 6, 9, 12			
0	0	1	2	1	1	0	2	2	0	1	2
0	2	1	0	1	2	2	1	0	0	1	2
1	0	0	2	2	1	2	1	0	1	2	0
0	1	2	0	0	1	2	0	0	1	2	0
1	2	0	0	2	1	0	2	2	1	2	0
2	0	1	1	0	0	2	2	1	2	0	0
0	1	2	0	1	2	0	1	2	0	1	2
1	2	0	1	2	0	1	2	0	1	2	0
1	2	0	2	0	1	2	0	1	2	0	1
0	1	2	0	1	2	0	1	2	0	1	2
1	2	0	1	2	0	1	2	0	1	2	0
1	2	0	2	0	1	2	0	1	2	0	1

(A)

Mauldon calls the solution for twelve coins *maximal*. A maximal solution is one in which C is the largest number of coins admitting a solution for a given W . Solutions for less than the maximal number of coins he calls *submaximal*. He tackles maximal solutions first, for these are used in forming submaximal solutions as well.

Each solution has three rows, one for each of the three weighings, and four through 12 columns, one for each of the coins. Each row gives an allocation of the coins as being either set aside, or put in the left pan, or put in the right pan, represented by 0, 1, or 2, respectively. Notice that in each weighing the number of coins placed on the left pan is the same as the number placed on the right pan, that is, the solutions are *balanced*. For example, in the four-coin case the first weighing sets aside coins 0 and 1, puts coin 2 in the left pan, and coin 3 in the right pan. When $3 \mid C$ is one, that is, for C of 4, 7, and 10, the number of coins set aside is one more than the number on each pan. When $3 \mid C$ is 2, for C of 5, 8, and 11, the number of coins set aside is one less than the number on each pan. When $3 \mid C$ is 0, as in the right column, for 6, 9, and 12, the number of coins set aside is the same as the number on each pan. At each weighing, the possible results are: the pans are level, or the left pan is lower, or the right pan is lower, represented by 0, 1, and 2, respectively. The result of all three weighings is thus a list V of three items, chosen from 0 1 2.

For example, if there are four coins, and the counterfeit is coin 0, and is heavier than a good coin, the result of the first weighing is 0, since coin 0 is set aside; for the same reason, the result of the second weighing is also 0; in the third weighing coin 0 is in the left pan, so the result is 1. The overall result V is thus 0 0 1, and this corresponds to column 0 of the 4-coin solution. Additional complications come about if the counterfeit is lighter, for which table (B) is appropriate:

0 0 2 1	2 2 0 1 1	0 2 1 0 2 1	
0 1 2 0	2 1 1 2 0	0 2 1 2 1 0	
2 0 0 1	1 2 1 2 0	2 1 0 0 2 1	
0 2 1 0 0 2 1	1 0 2 2 2 0 1 1	0 2 1 0 2 1 0 2 1	
2 1 0 0 1 2 0	1 2 0 2 1 1 2 0	0 2 1 2 1 0 2 1 0	
1 0 2 2 0 0 1	1 0 2 1 2 1 2 0	2 1 0 0 2 1 2 1 0	(B)
0 2 1 0 2 1 0 0 2 1	0 2 1 1 0 2 2 2 0 1 1	0 2 1 0 2 1 0 2 1 0 2 1	
2 1 0 2 1 0 0 1 2 0	2 1 0 1 2 0 2 1 1 2 0	0 2 1 2 1 0 2 1 0 2 1 0	
2 1 0 1 0 2 2 0 0 1	1 0 2 1 0 2 1 2 1 2 0	2 1 0 0 2 1 2 1 0 1 0 2	

The entries in (B) are not used for allocating the coins, but rather to determine the false coin when it is *lighter* than the good coins. For example, if there are four coins, and the false coin is coin 0, and is lighter than the others, the result would be 0 0 2, corresponding to column 0 of the 4-coin table in (B). The tables in (B) are the 3s complement of those in (A).

The table below gives some of the vital statistics of the problem:

W	N	L	G
3	9	4	12
4	27	13	39
5	81	40	120
6	243	121	363
7	729	364	1092
8	2187	1093	3279

Column W gives the number of weighings required, column N gives the number of different cases that W weighings can solve, column L gives the least number of coins for W , and column G gives the greatest number of coins for W . For example, if W is four, 27 cases are solvable, with 13 coins the smallest case, and 39 coins the largest. I don't show the case for W of 2, since three is the only

First we produce the powers of 3 less than 3^W-1 :

```
3 ^ i. <: W [ W =: 3
1 3
```

Next we use the hook (+i.) with each of these:

```
(+ i.) &.> 3 ^ i. <: W
+-+-----+
|1|3 4 5|
+-+-----+
```

Last, we raze this:

```
; (+ i.) &.> 3 ^ i. <: W
1 3 4 5
```

We convert these to ternary, getting four distinct representations:

```
] za=: (W # 3) #: ; (+ i.) &.> 3 ^ i. <: W
0 0 1
0 1 0
0 1 1
0 1 2
```

Each row of **za** is used to create two additional rows by adding 1 and 2, mod 3, to it. Adding 1, mod 3, to 0 1 2 gives 1 2 0; adding 2 to 0 1 2 gives 2 0 1. Consequently each three-row subtable has distinct rows. Notice that each column is also balanced, having one each of 1 and 2.

```
] zb=: 3|0 1 2+/"1 za
0 0 1
1 1 2
2 2 0

0 1 0
1 2 1
2 0 2

0 1 1
1 2 2
2 0 0

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

This is turned into a single table by applying append insert (,/) to it. Since the individual columns of the subtables were balanced

the whole column is also balanced. In this case each weighing places four coins in each pan.

```

      ,/ zb
0 0 1
1 1 2
2 2 0
0 1 0
1 2 1
2 0 2
0 1 1
1 2 2
2 0 0
0 1 2
1 2 0
2 0 1

```

I'll transpose this to allow you to more easily to compare with the lower right-hand corner of table (A):

```

      ] zc=: |: , / 3 | 0 1 2 +/"1 zb
0 1 2 0 1 2 0 1 2 0 1 2
1 2 0 1 2 0 1 2 0 1 2 0
2 0 1 2 0 1 2 0 1 2 0 1

0 1 2 1 2 0 1 2 0 1 2 0
1 2 0 2 0 1 2 0 1 2 0 1
2 0 1 0 1 2 0 1 2 0 1 2

1 2 0 0 1 2 1 2 0 2 0 1
2 0 1 1 2 0 2 0 1 0 1 2
0 1 2 2 0 1 0 1 2 1 2 0

```

The entire maximal solution process can be encapsulated in monad `SX`, which takes the number of weighings as argument.

```

      SX=: 13 : ' ,/ 3 | 0 1 2 +/"1 (y # 3) #: ; ➡
      (+ i.) &.> 3 ^ i. <: y'
      |: SX 3
0 1 2 0 1 2 0 1 2 0 1 2
0 1 2 1 2 0 1 2 0 1 2 0
1 2 0 0 1 2 1 2 0 2 0 1

```

Constructing a general solution

Mauldon's method of obtaining a general solution involves a fairly complicated way of choosing one of two tables to be appended, in the cases where the number of coins has a 3-residue of 1 or 2, or no appended table for the case where the 3-residue is 0. It simpli-

fies things considerably if a third, empty table, is provided for this last case. If we call the appended tables A0, A1 and A2, for residues of 0, 1 and 2, respectively, and form them into a list of boxes AS. Mauldon doesn't give the principles used in constructing A1 and A2, he merely presents them without apology.

```

A0=: 0 3 $ 0
A1=: 4 3 $ 0 0 1 0 2 0 1 1 0 2 0 2
A2=: 8 3 $ 2 2 2 0 1 0 1 0 1 1 1 2 1 2 1 0 ➡
2 2 2 1 1 2 0 0
] AS=: A0;A1;A2
+---+-----+-----+
|    |0 0 1|2 2 2|
|    |0 2 0|0 1 0|
|    |1 1 0|1 0 1|
|    |2 0 2|1 1 2|
|    |    |1 2 1|
|    |    |0 2 2|
|    |    |2 1 1|
|    |    |2 0 0|
+---+-----+-----+

```

Each of these is balanced, and the last five rows of A2 are also balanced. If K is the number of coins in C, the proper table to append can be given by:

```

> AS {~ 3 | K [ K=: 5
2 2 2
0 1 0
1 0 1
1 1 2
1 2 1
0 2 2
2 1 1
2 0 0

```

This isn't all that is needed. The three columns in the tables are suited to the least number of weighings that may be required. If more weighings than three are needed, the first column is replicated $W-2$ times. For example, if W is 5 there are five weighings and the first column is replicated thrice:

```

(((W-2)#0), 1 2){"1 A2 [ W=: 5
2 2 2 2 2
0 0 0 1 0
1 1 1 0 1
1 1 1 1 2

```

```

1 1 1 2 1
0 0 0 2 2
2 2 2 1 1
2 2 2 0 0

```

A dyad *SA* to produce a table to append having the necessary number of columns can thus be given by:

```
SA=: 13 : '(((y-2)#0),1 2){"1>AS{~3|x'
```

where *x* is the number of coins and *y* is the number of weighings.

The two dyads *SX* and *SA* can be combined to give a general solution function *SG*:

```
SG=: 13 : '(-x){.x(SX,SA)y'
```

where *x* and *y* are as in *SA*. The phrase *(-x){* forms the solution by taking the last *x* rows of the table formed by appending the maximum and the appended tables.

A solution for the case of 8 coins and 3 weighings can be obtained by:

```

8 SG 3
2 2 2
0 1 0
1 0 1
1 1 2
1 2 1
0 2 2
2 1 1
2 0 0

```

I found it convenient to have a monad which takes a list of coins as argument. The monad is:

```

SC=: (SG WK)@#
C=: 0 0 0 0 0 1 0 0
] S=: SC C
2 2 2
0 1 0
1 0 1
1 1 2
1 2 1
0 2 2
2 1 1
2 0 0

```


Finding a false coin

We can find a solution by writing:

```
S+//."1&. |:C
```

The dual transpose ($\&. |:$) causes the arguments to be transposed before being used. This has no effect on the coin list C , but is effective on S , interchanging columns and rows. The rank one ("1") allows the rows of transposed S to be used individually with C . The sum by key ($+//.$) adds the items of C according to the keys in the row of transposed S . The result of sum key of the first row with C is:

```
  2 0 1 1 1 0 2 2 +//. 0 0 0 0 0 1 0 0
0 1 0
```

The result comes from summing all the items of the right argument corresponding to 2s in the left argument (0), then those corresponding to 0s (1) then those corresponding to 1s (0).

The same thing occurs when I use all the rows of transposed S with C :

```
  S +/ /. "1 &. |: C
0 1 1
1 0 0
0 0 0
```

This is difficult to interpret because the first column 0 1 0 is in the order 2 0 1, the order in which they occur in the first column of S ; the second column 1 0 0 is in the order 2 1 0; the third column is in the order 2 0 1.

In order to avoid this difficulty, I prefix the solution rows (transposed columns) with 0 1 2 and the coin list with 0 0 0. The prefixed zeros on the coin list can't alter the result, but the 0 1 2 prefixed to the columns ensures that the result comes in a way that is easy to interpret.

```
  ] zr=: }. S (0 1 2&,@[ +//. 0 0 0&,@])"1&. |: C
0 0 0
0 1 1
```

I drop the first row of the result, which corresponds to the coins set aside, because in the physical experiment these are not seen. The table zr is interpreted thus: the rows correspond to left pan

and right pan, and the columns correspond to weighings. In the first weighing the left and right pans were level. In the second and third weighing the right pan was lower.

Now, if I take the difference of the left and right pan rows I get:

```
-/}.S (0 1 2&,@[ +//. 0 0 0&,@])"1&. |: C
0 _1 _1
```

And I need the 3s-complement of this:

```
3|-/}.S (0 1 2&,@[ +//. 0 0 0&,@])"1&. |: C
0 2 2
```

This tells me that in the first weighing the pans were level, and in the last two the right pan was lower.

The dyad **WR** encapsulates the preceding steps:

```
WR=: 13 : '3|-/ }. x (0 1 2&,@[ +//. 0 0 0 &,@])"1&. |: y'
] WL=: S WR C
0 2 2
```

The next step is to find the index of **WL** in either **S** or its 3s complement. The 3s complement is formed by taking the 3-residue of **-S**. This is laminated (**, :**) to **S**, forming **SC**:

```
] SC=: (, :3&|@-)S
2 2 2
0 1 0
1 0 1
1 1 2
1 2 1
0 2 2
2 1 1
2 0 0

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

By using rank 2 1 with the index of function, we can obtain the indices in both planes of **SC**:

```
SC i."2 1 WL
5 8
```

Since there are 8 items in each of the tables, the result 5 8 means that the weighing list was found in item 5 of the first table, and not at all in the second. This sequence is encapsulated in dyad **WI**:

```
WI=: 13 : '((,:3&|@-)x)i."2 1 y'
S WI WL
5 8
```

This has all the information needed for the answer. The index is clearly the smaller of the two values. The heavier or lighter indication is given by whether the index is in the first or second table: if in the first, it is heavier; if in the second it is lighter. The final result can thus be given by **RW**:

```
RW=: 13 : '(<./y),({&_1 1)</y)'
RW 5 8
5 1
```

This says that coin 5 is false, and it is heavier than a good coin.

To complete the problem, it would be necessary to provide for the case where there isn't a false coin. What happens then?

```
S WI 0 0 0
8 8
RW 8 8
8 _1
```

So the result when there is no false coin is an impossible index and a lighter coin indication.

Solution to Schell's Problem

Assume the eight coins are labeled A B C D E F G H. Then the steps below show how to solve the problem with two weighings. The first column gives the step number, and the next two columns give the allocation of coins to pans. The three columns at the right indicate the result of the weighing. After step 1, only one of either step 2 or step 3 or step 4 is executed. Each of these steps has 3 possible outcomes. For example, if the result of step 1 is "right pan high", go to step 3, which tells us to place coin D in the left pan and coin E in the right pan. If the left pan is now high, this means

that D is the false coin, and so forth.

step	left pan	right pan	left pan high	right pan high	pans level
1	A B C	D E F	go to step 2	go to step 3	go to step 4
2	A	B	A	B	C
3	D	E	D	E	F
4	G	H	G	H	none

30 Second Order Josephus

First published in Vector, 18, 4, (April 2002), 132-138.

Revised by Boyko Bantchev, (April 2009).

Every once in a blue moon this column is relatively easy to write. This one almost writes itself. Just a short while ago I received an intriguing message. It forms the bulk of this column. I've just had to change a word here and there and adjust typography as needed. Here's how the message opens:

From bantchev@math.bas.bg Thu Jan 17 18:07:53 2002
Date: Fri, 30 Nov 2001 18:32:13 -0800 (PST)
From: Boyko Bantchev
To: forum@jsoftware.com
Subject: Second-order Josephus

In a recent posting Eugene McDonnell defined the verb S that gives the survivor number for the Josephus problem (also in *Vector* 9/2 (1992)):

$S = .\ 1\&|. \&.\#:$

Now suppose that, for n persons, $S(n)$ is *not* the survivor number, but the one to be *eliminated*; i.e., every second person in a circle is marked until only one remains—and that one is eliminated.

This leads to a “second-order survival problem”; having eliminated $S(n)$, start again from the beginning with the remaining $n-1$ people, eliminate the one whose ordinal number *in the new sequence* is $S(n-1)$, then do the same with $S(n-2)$ and so forth until only one is left. What is the number $S_2(n)$ of the second-order survivor?

I must confess that my first reaction was somewhat guarded. I wasn't at all sure that this problem would lead to as much in the way of theory as the original Josephus did. For example, the book *Concrete Mathematics*, by Graham, Knuth, and Patashnik, devotes a full nine pages to Josephus, and gives half a dozen Josephus problems, in section 1.3. However, my mind was open, so I plowed on. Bantchev's message continues:

The verb

$E = : ((<:@[\{ . \}], \} .) \sim S @ \#$

eliminates the $S\#y$ -th member from any list y , so, if:

$n = : 9$

and

```

] y=: 1+i.n
1 2 3 4 5 6 7 8 9
E y
1 2 4 5 6 7 8 9    NB. since 3=S 9
E^:2 y
2 4 5 6 7 8 9      NB. since 1=S 8
E^:3 y
2 4 5 6 7 8        NB. since 7=S 7
E^:4 y
2 4 5 6 8           NB. since 5=S 6
E^:5 y
2 4 6 8             NB. since 3=S 5
E^:6 y
4 6 8               NB. since 1=S 4
E^:7 y
4 6                 NB. since 3=S 3
E^:8 y
6                   NB. since 1=S 2

```

Therefore, 6 survives ($6=S_2(9)$).

In general, we can set

$S_2 =: E^:(<:@\#)@(>:@i.)$

but, in fact, it can be shown that, for any $n > 1$, $S_2(n)$ is

$>: k+2^{<:m}$ when $k < 2^{<:m}$, and 2^m otherwise, where
 $m =: <. 2^{<.n}$ (i.e. $(n > 2^m) * . n < 2^{>:m}$) and $k =: n - 2^m$. So,
 S_2 can be defined without resorting to E or S :

$S_2 =: (>:@(1\&,@(\{.+.\})@}\&.\#))"0$

and we can check the definition:

```

S2 2+i.30
2 2 3 4 4 4 5 6 7 8 8 8 8 8 9 10 11 12 13 14 15 16
16 16 16 16 16 16 16 16

```

To my regret S_2 is, though clearly inspired by the definition of S , three times longer than S is, and not at all that elegant. But I did get some fun while writing it, hence my posting it to you.

/Boyko

I studied this message for a while and was confused. Mr.

Bantchev gives a formula for S_2 which seemed completely different from the immediately preceding formulas. I couldn't reconcile:

when $k < 2^{<:m}$, $S_2(n)$ is $>: k + 2^{<:m}$
 otherwise $2^{<.n}$
 $m =: <. 2^{<.n}$
 $k =: n - 2^{<.n}$

with:

```
S2=: (>: @ (1 & , @ ( {. +. }. ) @ }. &. #: ) ) " 0
```

The leap was too great. I decided to go step by step until I had a better grasp of what was going on. I wrote this function, which follows Boyko's analysis faithfully:

```
Boyko=: monad define
n =. y
m =. <. 2 ^ n
k =. n - 2 ^ m
if.
  k < 2 ^ <: m
do.
  >: k + 2 ^ <: m
else.
  2 ^ m
end.
)
```

The argument to Boyko is an integer > 1 . It yields the 2nd-order Josephus survivor number of that integer:

```
Boyko"0 [ 2+i.30
2 2 3 4 4 4 5 6 7 8 8 8 8 8 9 10 11 12 13 14 15 16
16 16 16 16 16 16 16 16 16 (B)
```

This agrees with the result of S2 in his message. The next step was to take his S2 apart, piece by piece. I'll repeat S2 here, so you can follow the steps:

```
S2=: (>: @ (1 & , @ ( {. +. }. ) @ }. &. #: ) ) " 0
```

Taking 19 as argument, convert it to binary:

```
#:19
1 0 0 1 1
```

Behead this:

```
}.#:19
0 0 1 1
```

"Or" the head with the behead:

```
({.+.}.).#:19
0 1 1
```

Prefix a 1:

```
1,({.+.}.).#:19
1 0 1 1
```

Find its base-2 value:

```
#.1,({.+.}).#:19
11
```

Add 1:

```
>:#.1,({.+.}).#:19
12
```

This validates, but doesn't clarify, how S2 was put together. Since we know that Mr. Bantchev was trying to arrive at a solution that used the binary representations of numbers, I suspected that the answer might be found by looking at the binary values of argument and result side by side:

```
q=:2+i.20
(.,q);(#:q);(#:w);(.,w=: Boyko"0 q)
+--+-----+-----+--+
| 2|0 0 0 1 0|0 0 1 0| 2|
| 3|0 0 0 1 1|0 0 1 0| 2|
| 4|0 0 1 0 0|0 0 1 1| 3|
| 5|0 0 1 0 1|0 1 0 0| 4|
| 6|0 0 1 1 0|0 1 0 0| 4|
| 7|0 0 1 1 1|0 1 0 0| 4|
| 8|0 1 0 0 0|0 1 0 1| 5|
| 9|0 1 0 0 1|0 1 1 0| 6|
|10|0 1 0 1 0|0 1 1 1| 7|
|11|0 1 0 1 1|1 0 0 0| 8|
|12|0 1 1 0 0|1 0 0 0| 8|
|13|0 1 1 0 1|1 0 0 0| 8|
|14|0 1 1 1 0|1 0 0 0| 8|
|15|0 1 1 1 1|1 0 0 0| 8|
|16|1 0 0 0 0|1 0 0 1| 9|
|17|1 0 0 0 1|1 0 1 0|10|
|18|1 0 0 1 0|1 0 1 1|11|
|19|1 0 0 1 1|1 1 0 0|12|
|20|1 0 1 0 0|1 1 0 1|13|
|21|1 0 1 0 1|1 1 1 0|14|
+--+-----+-----+--+
```

It takes a bit of study, but it should be possible eventually to arrive at the S2 solution. Notice that the leading bit plays no role. The significant bit is the second. When this is 1, the result will be a power of 2, because the “or” of the second bit with the trailing bits will produce all 1s, prefixing a 1 keeps them all ones, converting this to integer will produce a result one less than a power of two, and adding one to this will yield a power of two. When the

second bit is 0, the trailing bits are unaltered. When 1 is prefixed, the result will be a power of two only when the trailing bits are all 1, as in the case of 11. The binary form of 11 is 1 0 1 1; beheading gives 0 1 1; “or”ing 0 with 1 1 gives 1 1; prefixing 1 gives 1 1 1; converting to integer gives 7; adding 1 gives 8, a power of two.

I’m guessing that Mr. Bantchev followed a process much like the one I’ve described just above: finding the values with his algebraic analysis, converting these to binary, and comparing them with the binary form of the arguments. This is now enshrined in Sloane’s *On-Line Encyclopedia of Integer Sequences* as sequence A066997.

I thought I had found an interesting property of (B). I noted the indices (in 2-origin) of the first appearance of a power of two were at indices 2, 5, 11, 23. I checked in the *Online Encyclopedia of Integer Sequences* and found that it corresponded to sequence A055010. The entry notes that these numbers, written in binary, are of the form $a(n)$ is 1011111-1. Furthermore, it gave the following formula for $a(n)$:

$$a(n) = (3 \cdot 2^n) - 1$$

I sent a message to Henry Bottomley, the author of this entry, and he in his reply alerted me to the existence of sequence A006165, which is:

1 1 2 2 3 4 4 4 5 6 7 8 8 8 8 8 9 10 11 12 13 14
15 16 16 16 16 16 16 ...

This should be familiar, as it is the S2 sequence, with two leading 1s. I kicked myself for not having found this on my own, and began to appreciate that Mr. Bantchev might be on to something not completely trivial.

The entry for series A006165, gives two recursive formulas, one for odd n , and the other for even:

$$\begin{aligned} a((2 \cdot n) + 1) &= a(n + 1) + a(n) & (A) \\ a(2 \cdot n) &= 2 \cdot a(n) & (B) \end{aligned}$$

By a bit of finagling it’s possible to combine these two into one. The ceiling and floor of $(2 \cdot n) + 1$ are $(n + 1)$ and (n) , as in (A) above. The ceiling and floor of $2 \cdot n$ are both n , as in (B) above. This makes it possible, for whatever integer, to get the result by

the same formula, which takes the sum of the ceiling and floor of half the number. The explicit function A below takes as argument the number of consecutive survivor numbers desired, and yields that many. For an argument of 0 it yields an empty list, and for an argument of 1 it yields a list whose item is 1.

```
A=: monad define
  if. y < 2 do.
    y # 1
  else.
    a =. 1 1
    while. y > # a do.
      b =. ( <. , >. ) -: <: # a
      a =. a , + / b { a
    end.
  end.
)
```

A 30 duplicates A006165 faithfully. It differs from Bantchev's series in using offset 1. I found a much faster way to generate the sequence. It forms two lists, first just the 1-origin integers, then the extra powers of two needed:

```
(>:i./2^i.y)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
;(#each+:)2^i.y
2 4 4 8 8 8 8 16 16 16 16 16 16 16 16
```

(shown for $y=4$) then it joins these, and sorts them:

```
1 2 2 3 4 4 4 5 6 7 8 8 8 8 8 9 10 11 12 13 14 ➡
15 16 16 16 16 16 16 16 16
```

Here is the high-speed Josephus 2 function:

```
hsJ2=: 13 : ' /:~(>:i./2^i.y),;(#each+:)2^i.y'
```

Its y is the number of powers of 2 to use in generating the lists and the result is a list as long as twice the sum of those powers of 2:

```
2^i.4
1 2 4 8
+/2^i.4
15
+:/2^i.4
30

hsJ2 4
1 2 2 3 4 4 4 5 6 7 8 8 8 8 8 9 10 11 12 13 14 15 ➡
16 16 16 16 16 16 16 16
#hsJ2 4
30
```

31 J be Nimble, J be Quick: Nim Addition

First published in Vector, 19, 1, (June 2002), 108-118.

Revised by Fraser Jackson, (April 2009).

Nim Addition

Nim is a simple game that someone knowledgeable playing against someone naïve can almost always win. It was featured in the 1960s film *Last Year in Marienbad*, where two men in a bar play with a number of piles of matchsticks. The players, in turn, take any number of matchsticks from any one of the piles. The object is to be the player who takes the last match or matches. Its name came perhaps from *nimm*, the third person singular imperative of the German verb *nehmen*, meaning *to take*. The trick in Nim is knowing that a position is either safe or unsafe, depending on whether the Nim sum of the number of matches in each pile is or is not zero. The Nim sum can be obtained by converting the number of matchsticks in each pile to binary, and inserting not-equals, or exclusive-or over this, then converting back to integer. For example, if there are three piles, with three, five, and seven matches in the piles, the Nim sum is obtained in three steps. First, the binary forms of the numbers are taken:

```
#: piles=: 3 5 7
0 1 1
1 0 1
1 1 1
```

The not-equal function yields the parity of its summands:

```
~: / #: piles
0 0 1
```

This is converted to decimal:

```
#. ~: / #: piles
1
```

The function NS encapsulates this:

```
NS=: ~: / &. #: ➡
NB. not-equal insert dual antibase
NS piles
1
```

A safe move can be made if and only if the Nim sum of the piles is not zero, meaning unsafe. If a position is safe, any move will change it to unsafe. Furthermore, if the Nim sum of the piles is

nonzero, it can always be made safe by subtracting from one of the piles. There will always be at least one such pile. For example, given the piles 3 5 7, with Nim sum 0 0 1, we can subtract one from any one of the piles. Thus, three different safe moves can be made, resulting in one of 2 5 7 or 3 4 7 or 3 5 6.

```
NS/"1 [ 2 5 7, 3 4 7,: 3 5 6
0 0 0
```

The choice of which pile to subtract from, when more than one is a candidate, is arbitrary.

Now, suppose we have a Nim sum of a list of piles that is a bit more complicated (the function `h` displays the binary form of the piles and its binary sum):

```
h=: ,. @ (#: ; [: ~:/ #:)
h 10 11 4
+-----+
|1 0 1 0|
|1 0 1 1|
|0 1 0 0|
+-----+
|0 1 0 1|
+-----+
```

The only solution for this is to subtract 3 from the last pile, which yields 10 11 1:

```
h 10 11 1
+-----+
|1 0 1 0|
|1 0 1 1|
|0 0 0 1|
+-----+
|0 0 0 0|
+-----+
```

There's a certain amount of art in playing a winning game of Nim.

Nim multiplication

John H. Conway and Richard K. Guy have written *The Book of Numbers*.^[1] I was encouraged to read this by Ken Iverson's Lab which uses J to explore many of the parts of this book. Its last chapter is "Infinite and Transcendental Numbers", and in it, to my surprise, is a discussion of the game of Nim. Conway & Guy coined the word *nimbers* for the ordinary decimal integers, in the

Nim context. I think they are confusing the numbers involved with the functions used with them. Conway & Guy, in addition to discussing Nim addition, also treat Nim multiplication, which they state is valuable in studying the digital transmission of information, in particular “the integral lexicographical code of minimal distance 3”. They give a multiplication table for the first sixteen nonnegative integers. They also write:

And here’s all you need to know about the multiplication of nimbers: If the ‘larger’ of two different nimbers is 1 or 2 or 4 or 16 or 256 or 65536 or 4294967296 or ..., you multiply them just as you multiply the corresponding ordinary numbers. The product of one of these *special* nimbers with itself is obtained by taking $1\frac{1}{2}$ times its ordinary value.

I found it impossible to use this rule for nimbers greater than 4. I turned to Google for help, and found that Sloane’s *On Line Encyclopedia of Integer Sequences* [2] contained entries on Nim multiplication which included a function which built a Nim multiplication table.* The problem was that the function was written in Maple, and although I am able to read very simple Maple, this one used built-in functions with meanings I couldn’t grasp, even after I found a Maple manual on the Web. After weeks of trying to come to terms with it, appealing for help to several people I thought could help, but didn’t, I wrote appeals to Conway and Guy, and also appealed to the J discussion group on the Web for help. Both pleas were successful; Professor Guy replied to my letter, and Mike Day read my appeal to the J group, was able to decipher the Maple, and turned it into J. Ken Iverson made some revisions to it and I added my own changes. Here is its latest manifestation:

```
mt=: monad define
  iN=.i.N=.y
  MT=.(>.|:.)iN 1}0$~,~N
  for_a. 2}.iN do.for_b. a}.iN do.
    c=.a,b,|:(#:i.@*)a,b
    r=.<"1[0 1|:(2 1,0 3,:2 3){c
    t=..(~/&.#:)"1 r{"1 2 MT
    j=.(0:i.~)e.~[:i.2:+>./)t
    MT=.j((;|. )a,b)}MT
  end.end.
)
```

* viz. entry A051776. We show the Maple program on page 265. (Ed.)

The argument `N=.y` is the size of the square desired. Nim multiplication is commutative so the derivation of one nondiagonal value allows its symmetrical twin to be created at the same time. I use the term *nonneg* in order to shorten the phrase *nonnegative integer*. The list `iN` of the first `N` nonnegs serves to initialize row 1 and column 1, and is also used to determine the values of the loop counters `a` and `b`. An `NxN` matrix of zeros is created (`0$~,~N`) and row 1 is amended with `iN`; column 1 is set by forming the maximum of this matrix and its transpose (`>.|:`). For `N=5` the result is:

```
] iN=: i.N=. y [ y =: 5
0 1 2 3 4
] MT=: (>.|:)iN 1}0$~,~N
0 0 0 0 0
0 1 2 3 4
0 2 0 0 0
0 3 0 0 0
0 4 0 0 0
```

The first value to be created is that in row 2, column 2. After that comes row 2 column 3 and row 3 column 2, and so on until row 2 and column 2 is completed. Then comes 3 3 and 3 4 (and 4 3), and finally 4 4. The process for 2 2 is as follows:

```
a=: b=: 2
|:(#:i.@*/)a,b
0 0 1 1
0 1 0 1
```

This matrix gets two new rows, a row of all `a`'s on a row of all `b`'s. The resulting four rows correspond to those labeled `a`, `b`, `i` and `j` in the `Maple` function. Combinations of these rows can be assembled so that critical values preceding the one currently being made can be used according to a rule which I can't explain, since I don't understand it.

```
] c=: a,b,|:(#:i.@*/)a,b
2 2 2 2
2 2 2 2
0 0 1 1
0 1 0 1
```

The actual selection uses items at 2 1 (`i,b`), 0 3 (`a,j`), and 2 3 (`i,j`).

```

      (2 1,0 3,:2 3){c
0 0 1 1
2 2 2 2

2 2 2 2
0 1 0 1

0 0 1 1
0 1 0 1

```

These are transposed by placing the first two axes at the end
(0 1 |:)

```

      0 1|:(2 1,0 3,:2 3){c
0 2
2 0
0 0

0 2
2 1
0 1

1 2
2 0
1 0

1 2
2 1
1 1

```

In order for these to be used as indices to MT, their rows are boxed:

```

      ] r=: <"1[0 1|:(2 1,0 3,:2 3){c
+---+---+---+
|0 2|2 0|0 0|
+---+---+---+
|0 2|2 1|0 1|
+---+---+---+
|1 2|2 0|1 0|
+---+---+---+
|1 2|2 1|1 1|
+---+---+---+

```

The table of indices selects the needed values: (r{"1 2 MT), then the Nim sums of the rows are determined: ((~:/&.#:)"1) and duplicate sums are removed: (~.)

```

      r{"1 2 MT
0 0 0
0 2 0
2 0 0
2 2 1
    ] t=: ~.(~:/&.#:)"1 r{"1 2 MT
0 2 1

```

The mysterious part comes now. The value *j* to be stored at (*a*,*b*) is the *least* nonneg not in *t*. Why this produces the Nim multiplication of *a* and *b* is beyond me to explain.

The candidates for *j* are all in the first 2+>./*t* nonnegs:

```

      i.2+>./t
0 1 2 3

```

The ones already present in *t* are identified:

```

      t e.~ 0 1 2 3
1 1 1 0

```

and *j* is the index of the first zero in this list:

```

      0 i.~1 1 1 0
3

```

Here's the whole:

```

    ] j=: (0:i.~]e.~[:i.2:+>./)t
3

```

and here is the finished 5×5 table:

```

      mt 5
0 0 0 0 0
0 1 2 3 4
0 2 3 1 8
0 3 1 2 12
0 4 8 12 6

```

We now know how to make a Nim multiplication table, but still don't know how to Nim-multiply two arbitrary numbers.

Here is Mike Day's program *mtMD* which accurately translates the Maple program I had asked for help with. Without Mike's contribution I couldn't have got any further:

```

nimsum =: ~:/&.#:@,"0/~      NB. EemcD
mtMD=: verb define
iN =. i. >: N =. y
NB. =====
NB. lines 1 to 6

```



```

MT =. 0 $~ 2 # N + 1      NB. initialise MT
                           NB. with 0 top & left
MT =. iN 1 } MT           NB. and indices in row 1
NB. MT =. iN 1 }"_1 MT    NB. originally also in col 1
NB. - We can defer symmetrising and just work on
NB. diag and upper triangle
NB. =====
NB. lines 7 - 11 - should be able to cut out some
NB. loops by eg recursion or scan
for_a. 2 }. iN do.
  for_b. iN }.~ a do.
    t1 =. i. 0
    for_i. i. a do.
      for_j. i. b do.
NB. =====
NB. lines 12-24 are preamble to line 25
  NB. references to stored AT where available
  NB. or nimsum where not avail. obscures the
  NB. process -
  NB. This is ok on a fast m/c and/or for small N

  NB. line 25 (26 is a comment) ...
  NB. sort refs since using diag and upper
  NB. triangle only
  refs =. sort each (i,b);(a,j);(i,j)
  t1 =. t1 , nimsum / refs { MT
NB. =====
  end.      NB. line 27
  end.      NB. line 28
NB. =====
  NB. line 29 - seems to require the nub
  t2 =. sort ~. t1
NB. =====
  NB. lines 31 - 36 - locate first element of t2
  NB. not equal to its index
  j =. 1 i.~ t2 ~: i. # t2
NB. =====
  NB. line 37 only
  MT =. j (<a,b) } MT    NB. don't need line 38
NB. =====
  end.      NB. line 39
end.      NB. line 40
NB. =====
NB. extra line to symmetrise
MT + (iN >/ iN) * |: MT
)

```

NB. a line-numbered Maple source listing for ➡
comparison

```

maple_source := 0 : 0
0 MT:=array(0..N,0..N); for a from 0 to N
1 do
2 MT[a,0]:=0;
3 MT[0,a]:=0;
4 MT[a,1]:=a;
5 MT[1,a]:=a;
6 od;
7 for a from 2 to N do
8 for b from a to N do
9 t1:={};
10 for i from 0 to a-1 do
11 for j from 0 to b-1 do
12 u1:=MT[i,b];
13 u2:=MT[a,j];
14 if u1<=NA and u2<=NA then
15 u12:=AT[u1,u2];
16 else
17 u12:=nimsum(u1,u2);
18 fi;
19 u3:=MT[i,j];
20 if u12<=NA and u3<=NA then
21 u4:=AT[u12,u3];
22 else
23 u4:=nimsum(u12,u3);
24 fi;
25 t1:={ op(t1), u4};
26 #t1:={ op(t1), AT[ AT[ MT[i,b], MT[a,j] ], ➡
MT[i,j] ] };
27 od;
28 od;
29 t2:=sort(convert(t1,list));
30 j:=nops(t2);
31 for i from 1 to nops(t2) do
32 if t2[i] <> i-1 then
33 j:=i-1;
34 break;
35 fi;
36 od;
37 MT[a,b]:=j;
38 MT[b,a]:=j;
39 od;
40 od;
)

```

I found that the number of times t the inner j loop of Day's program was used, for differing sizes s of arguments, to be:

```
s    t
2    4
3   19
4   55
5  125
6  245
7  434
8  714
```

The fifth difference of t is zero, so a polynomial of degree 4 can be found:

```
diff=: 2: --/\ ]
t=: 4 19 55 125 245 434 714
diff t
15 36 70 120 189 280
diff^:2 t
21 34 50 69 91
diff^:3 t
13 16 19 22
diff^:4 t
3 3 3
diff^:5 t
0 0
```

The polynomial is formed like this:

```
x=: i.#t
t %. x^/i.5x
4 97r12 43r8 17r12 1r8
```

These can be made the numerators for a rational polynomial:

```
] c=: 24 2 3 2 3*4 97 43 17 1
96 194 129 34 3
polyn=: c&p.%24&p.
polyn i.7
4 19 55 125 245 434 714
```

I'll make a slight detour here, to explore the result of `polyn t` further. I found that the fourth degree polynomial for the figurate numbers of order 5 is relevant. These numbers are those in the fifth diagonal of the Pascal triangle. In fact, I found that a multiple of these added to the figurate numbers of order 4 gives us our numbers:

```
] p4=: 3!3+i.7
```

```

1 4 10 20 35 56 84
  ] p5=: 4!4+i.7
1 5 15 35 70 126 210
  p4+3*p5
4 19 55 125 245 434 714

```

The detour is over. Now I'll use our polynomial to find how often the inner loop is entered for a size 209 table:

```

polyn 209x
251673415

```

A quarter of a thousand million iterations seems excessive.

This makes clear how ridiculous and expensive it is to have to make a 209×209 table in order to get the Nim product of 167 and 208! The letter I got from Professor Guy helps here. I had asked him how to Nim multiply 8x8, and his letter showed how, and also how to multiply 5 by 11.

Here it is. He uses the plus and times signs within circles, and I've substituted + and *. I've also replaced his linear ordering of equal statements with Iverson's convention of placing them one below the other.

Dear Eugene McDonnell,

Nim-multiplication is tricky, but you can probably catch on by remembering to deal with the exponents in the same way that you deal with numbers in nim-addition, namely split them into powers of 2. Nim multiplication of powers of 2 is defined, in the first instance, only for the 'Fermat powers of 2'

```

(2^2^0) = 2
(2^2^1) = 4
(2^2^2) = 16
(2^2^3) = 256
(2^2^4) = 65536

```

...

each, after the first, being the square of the previous one, but if instead of 'square' you mean 'nim-multiply by itself', then the answer is defined to be

```

(x*x)
(3%2)*x

```

(just if x is a Fermat power of 2).

To deal with other powers of 2, you work at one level higher up, thinking of (2^13), for example, as (2^(8+4+1)) and use the associative, commutative and distributive laws, e.g.,

$$\begin{aligned}
 &8 * 8 \\
 &(2^3) * (2^3) \\
 &(2^{(2+1)}) * ((2^{(2+1)})) \\
 &(2^2) * (2^1) * (2^2) * (2^1) \\
 &((2^2) * (2^2)) * ((2^1) * (2^1)) \\
 &(4 * 4) * (2 * 2) \\
 &6 * 3 \\
 &(4+2) * (2+1) \\
 &(4*2) + (4*1) + (2*2) + (2*1) \\
 &8+4+3+2 \\
 &13
 \end{aligned}$$

To deal with numbers which are not powers of 2 leads to a corresponding extra level of complication. E.g.,

$$\begin{aligned}
 &5 * 11 \\
 &(4+1) * (8+2+1) \\
 &(4*8) + (4*2) + (4*1) + (1*8) + (1*2) + (1*1) \\
 &(4*4*2) + 8+4+8+2+1 \\
 &(6*2) + 7 \\
 &((4+2)*2) + 7 \\
 &(4*2) + (2*2) + 7 \\
 &8+3+7 \\
 &12
 \end{aligned}$$

8 is the first power of 2 that is not a Fermat power, and the first place where you run into any difficulty.

Best wishes,

Yours sincerely,

Richard K. Guy,
 Faculty Professor of Mathematics
 University of Calgary.

```
NP=: 4 : 'NS x,y'"0 NB. Nim-Plus
NB. NP is a dyadic NS for use with utility: table
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
1	1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14			
2	2	3	0	1	6	7	4	5	10	11	8	9	14	15	12	13			
3	3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12			
4	4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11			
5	5	4	7	6	1	0	3	2	13	12	15	14	9	8	11	10			
6	6	7	4	5	2	3	0	1	14	15	12	13	10	11	8	9			
7	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8			
8	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7			
9	9	8	11	10	13	12	15	14	1	0	3	2	5	4	7	6			
10	10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5			
11	11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4			
12	12	13	14	15	8	9	10	11	4	5	6	7	0	1	2	3			
13	13	12	15	14	9	8	11	10	5	4	7	6	1	0	3	2			
14	14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1			
15	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

Here is the Nim multiplication table:*

```
label=: 4 : '(x;" : ,.i.#y),.({.;})" : (i.{$y),y'
'**' label mt 16
```

+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	0	2	3	1	8	10	11	9	12	14	15	13	4	6	7	5
	3	0	3	1	2	12	15	13	14	4	7	5	6	8	11	9	10
	4	0	4	8	12	6	2	14	10	11	15	3	7	13	9	5	1
	5	0	5	10	15	2	7	8	13	3	6	9	12	1	4	11	14
	6	0	6	11	13	14	8	5	3	7	1	12	10	9	15	2	4
	7	0	7	9	14	10	13	3	4	15	8	6	1	5	2	12	11
	8	0	8	12	4	11	3	7	15	13	5	1	9	6	14	10	2
	9	0	9	14	7	15	6	1	8	5	12	11	2	10	3	4	13
	10	0	10	15	5	3	9	12	6	1	11	14	4	2	8	13	7
	11	0	11	13	6	7	12	10	1	9	2	4	15	14	5	3	8
	12	0	12	4	8	13	1	9	5	6	10	2	14	11	7	15	3
	13	0	13	6	11	9	4	15	2	14	3	8	5	7	10	1	12
	14	0	14	7	9	5	11	2	12	10	4	13	3	15	1	8	6
	15	0	15	5	10	1	14	4	11	2	13	7	8	3	12	6	9
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

This is identical with Table 10.2 in Conway and Guy[1].

* Thanks to Roger Hui for the definition of: `label`.

To produce the addition table we were able to use the standard J utility: `table` with our own special Nim “plus-sign”: `NP`, which needs to be called for every entry in the table. But for the multiplication table we have an efficiently computed table: `mt 16`, hence the need for a different verb to append row and column numbers. Note that `label` requires as its left argument the string to appear in the top left corner of the formatted output. Alternatively, use the `border` utility: `system/packages/misc/border.ijs` (*Ed.*)

Reference

- [1] Conway, J. H., Guy, R., *The Book of Numbers*. Springer (Feb 1998), ISBN 978 0 3879 7993 9.
- [2] Sloane, N. J. A., *On-Line Encyclopedia of Integer Sequences*.
<http://www.research.att.com/~njas/sequences/>

32 Beware Scholes

First published in Vector, 19, 3, (January 2003), 137-142.

Revised by Chris Burke and Oleg Kobchenko, (April 2009).

Beware Scholes!

This article is about a J version of the Black-Scholes formulas, the brainchild of Myron Scholes and the late Fischer Black. The document: <http://bradley.bradley.edu/~arr/bsm/pg04.html> gives a lot of information on the formula and its creators (which won the surviving creator the Nobel Prize in economics in 1997), and if you want to find out more about Black and Scholes or the theory behind their formula, I recommend it.

A *call* is an option to *buy* a stipulated amount of stock at a specified time and price, and a *put* is an option to *sell* ditto. A person might acquire a call option who expects the price of the asset to rise. The Black-Scholes formulas enable the seller of the option to determine quite accurately what price to charge for such options.

Here are the formulas in conventional mathematical notation:

$$C = S N(d_1) - X e^{(-rT)} N(d_2)$$

$$P = X e^{(-rT)} N(-d_2) - S N(-d_1)$$

$$d_1 = \frac{\ln(S/K) + (r + v^2/2) T}{v \sqrt{T}}$$

$$d_2 = d_1 - (v \sqrt{T})$$

C = Theoretical Call Premium

P = Theoretical Put Premium

r = Risk-Free Interest Rate

T = Time in years until strike date

N = Cumulative Standard Normal Distribution

\ln = Natural Logarithm

S = Current Stock Price

X = Option Strike Price

v = volatility, or Standard Deviation of Asset Price.

Many different programming languages have been used to write programs for these formulas. The document:

<http://home.online.no/~espehaug/SayBlackScholes.html>

contains a couple of dozen of these programs, written in these languages:

C#	O'Caml
C++	Pascal
Fortran	Perl
Haskell	PHP
HP48	Python
Icon	Real Basic
IDL	Rebol
JAVA	Scheme
JavaScript	S-Plus
K	Squeak
Maple	Transact SQL
Mathematica	VBA
Matlab	

The programs are in one of two forms, both adhering closely to the original mathematical formulas shown above. Some have separate programs for calls and puts; some exploit the family resemblance of calls and puts and so write just one general program that requires an additional parameter to indicate whether a solution for a call or a put is desired. Here is a typical general program, this one written in C++:

```
Double BlackScholes(char CallPutFlag, double S, ➡
    X, T, r, v)
{
    double d1, d2;

    d1=(log(S/X)+(r+v*v/2)*T)/(v*sqrt(T));
    d2=d1-v*sqrt(T);

    if (CallPutFlag == 'c')
        return S * CND(d1)-X * exp(-r*T)*CND(d2)
    elseif (CallPutFlag == 'p')
        return X * exp(-r * T) * CND(-d2) - S * CND(-d1);
}
```

This program includes as its first argument the letter 'c' for a call option, and 'p' for a put option, and then discriminates between the two by an `if/elseif` control structure. Otherwise, it follows the Black-Scholes formulas closely. The entry includes a long separate

program for the required cumulative normal distribution function, as do many of the other entries.

At present the document has no contribution written in J (or APL, for that matter). The rest of this paper describes the evolution of J programs for the Black-Scholes formulas. Five different people made transformations of the formulas that ended in a J version radically different from all the others.

My attention was first called to this subject by a message from Hu Zhe to the J Forum that uses separate functions for call and put.

```

load '~system\packages\stats\statdist.ijs'
cnd=: 3 : 'normalprob 0, 1, __, y'

d1=: 3 : 0
'S X T r v'=. y
((^ .S%X)+(r+-: *:v)*T)%(v*%:T)
)

d2=: 3 : 0
'S X T r v'=. y
((^ .S%X)+(r--: *:v)*T)%(v*%:T)
)

BlackScholesCall=: 3 : 0
'S X T r v'=. y
(S*cnd d1 y) - (X*(^-r*T)*cnd d2 y)
)

BlackScholesPut=: 3 : 0
'S X T r v'=. y
(X*(^-r*T)*cnd -d2 y) - (S*cnd -d1 y)
)

```

These are reasonably concise and straightforward. They show what was to be expected: that J, as well as any other programming language, can translate the mathematical notation directly into computer programs. Notice that he loads a J library function for the cumulative normal distribution.

Shortly after this appeared, Oleg Kobchenko sent the following version, a single function for both call and put, that incorporates `d1` and `d2`:

```

    BlackScholes=: 4 : 0
    'S X T r v'=. y
    d1=. ((ln S%X)+(r+-:v)*T)%(v * sqrt T)
    d2=. d1 - v * sqrt T
    (S, X * exp-r*T) (-/ . * cnd"0)&(-^:x) (d1, d2)
  )

```

where `cnd` is as already defined, and verbs `ln`, `sqrt` and `exp` are as defined at the end of this article.*

The lines forming `d1` and `d2` are like those in the C++ program. The last line is an instance of array thinking. It exploits the similarity of the call and put functions. The put option definition can be rewritten. Here are the call and put options, with put in its new form.

```

c=. (S*cnd( d1))-((X*exp-r*T)*cnd( d2))
p=. -(S*cnd(-d1))-((X*exp-r*T)*cnd(-d2))

```

This shows that `p` differs from `c` solely in the use of negation of `d1` and `d2`, and in negating the overall result. Kobchenko exploits this by rearranging things so that a left argument of 0 or 1 discriminates call and put, respectively,

More abstractly, the last line of his function can be written as:

```

a ((b c)&d) e
(d a)(b c)(d e)
(d a) b (c(d e))
(d      a      ) b      (c      ( d      e      ))
(-^:x)(S,X*exp-r*T)(-/ . *) (cnd"0 (-^:x)(d1,d2))

```

This shows the conditional negation of the left and right hand sides, the application of `cnd` to the right hand side, and the difference of the product, so that for a call we would have:

```

c=. ( S,X*exp-r*T) -/ . * cnd"0( d1,d2)

```

and for a put we would have:

```

p=. (-S,X*exp-r*T) -/ . * cnd"0(-d1,d2)

```

* The last line originally read:

```

(S, X * exp-r*T) (-/ . * cnd)&(-^:x) (d1, d2)

```

For this and subsequent examples to work properly with J602, `cnd` must be replaced with `cnd"0` as shown, which forces `cnd d1` and `cnd d2` to be computed separately. This fix is *not* needed with the alternative implementation of `cnd`, due to Ewart Shaw, introduced later.

At the same time that Kobchenko was working on his array approach, I had been working on the other main part of the program, the formation of `d1` and `d2`. I wrote down the definition of `d2`:

```
d2=. d1 - v*%:T
```

Then I replaced `d1` by its definition, and with a bit of algebra arrived at:

```
d2=. ((^ .S%X)+(r--: *:v)*T)%(v * %:T)
```

and if you compare this with the definition for `d1`, you will find that the only difference is that `(r--: *:v)` is changed to `(r+: *:v)`. This being the case, it was simple to replace the two lines defining `d1` and `d2` by a single line that forms a two-item list `d` that uses the fork `(+ , -)`:

```
d=. ((^ .S%X)+(r(+,-)-: *:v)*T)%v*%:T
```

This permitted the definition of `BlackScholes` to become:

```
BlackScholes=: dyad define
'S X T r v'=. y
d=. ((^ .S%X)+(r(+,-)-: *:v)*T)%v*%:T
(S,X*^-r*T)(-/ .*cnd"0)&(^:x)d
)
```

The only thing about this that I found not to my liking was the need to specify a left argument to indicate call or put. Happily for me, just about this time Arthur Whitney posted a message to the K forum that showed that `v` can be used to discriminate the two cases, by using it positively for call, and negatively for put. Thus it became possible to do without the left argument, and write:

```
BS=: monad define
'S X T r v'=. y
d=. ((^ .S%X)+T*r(+,-)-: *:v)%v*%:T
-/ (S,X*^-r*T) * cnd"0 d
)
```

Notice that I have separated the parts of `(-/ . *)`, giving, I believe, a program easier to explain and understand.

Here are examples of call and put. The result for put is negative, and this differs from the usual put result, which is positive. The negative result can be useful to distinguish a call result from a put result. If a positive put result is necessary, a magnitude sign (`|`) can be placed in front of the last line of `BS`.

```

yc=: 60 65 0.25 0.08 0.3
BS yc
2.13338
yp=: 60 65 0.25 0.08 _0.3
BS yp
_5.84629

```

We haven't ended quite yet. Perhaps you remember the article by Ewart Shaw in *Vector* [1], in which he defined the error function `erf` using J's hypergeometric conjunction:

```

erf=: (*&(%:4p_1)%^@:*)*[:1 H. 1.5*: ➡
NB. A&S 7.1.21 (right)

```

and then defined the cumulative distribution function of the normal distribution `cnd` by:

```

cnd=: [: -: 1 :+[: erf%&(%:2) ➡
NB. A&S 26.2.29 (solved for P)

```

All of the functions written in other languages must do something special to define `cnd`, either using a library function, or writing the definition using approximation A&S 26.2.16 [2].

I'm going to contribute `BS`, `erf`, and `cnd` to the Black-Scholes website, but in the following training-wheels versions so that the innocent reader may come close to understanding them without having to learn any J.

```

BS=: monad define
'S X T r v'=. y
d=. ((ln S dv X) + T * r (+,-) hlf sqr v) ➡
dv (v * sqrt T)
diff (S , X * exp - r * T) * cnd d
)

erf=: monad define
NB. A&S 7.1.21 (rightmost)
((2 * y) dv (sqrt pi)) * (exp - y ^ 2) ➡
* (1 H. 1.5) y ^2
)

```

* As noted earlier, this implementation of `cnd` does not need the construct: `cnd"0` though it will still work with it.

```

cnd=: monad define
NB. A&S 26.2.29 (solved for P)
(1 + erf y * sqrt 0.5) dv 2
)

```

where:

```

diff =: -/
dv    =: %
exp   =: ^
hlf   =: -:
ln     =: ^.
pi     =: 1p1
sqr   =: *:
sqrt  =: %:

```

References

- [1] Shaw, E., Hypergeometric Functions and CDFs in *J. Vector* 18, 4, (April 2002), 139-143.
 - [2] Abramowitz, M., Stegun, I. A., (eds.), *Handbook of mathematical functions*. US National Bureau of Standards, Applied Mathematics Series - 55, (1964—1972). Also: Dover Publications (June, 1965), ISBN 978-0486612720.
- See also:
<http://www.math.sfu.ca/~cbm/aands/>
<http://dlmf.nist.gov/>

33 Pick A Card, Any Card

*First published in Vector, 19, 4, (April 2003), 101-107.**

Revised by Gilles Kirouac, (April, 2009).

Introduction

The crowd sits and waits, eagerly anticipating the showman's grand entrance. Eventually he arrives, bringing his glamorous assistant and a pack of cards with him. He selects a volunteer from the audience and asks them to pick five cards out of the pack and give them to the assistant, without of course seeing them himself. The assistant then shows him four of the cards, and, after a suitable dramatic pause, the showman identifies the fifth. The crowd applauds, and the magician and his assistant leave after a few repeats to show it wasn't a fluke.

Those of you who were at last month's Finnish conference will already have seen this spectacle, know that the showman in question is actually Adrian, and moreover will know the twist, which is that the assistant is not a 5' 6" blonde but a 4" by 2" grey box which comes with a screen and a stylus.

I'm sure that some of you asked him afterwards how he did it, and I suspect that instead of the usual "Magic!" he said "Wait for the next Vector"; this article shows you how the trick works and how the digital Esmerelda is written. (Of course, the assistant is far more important than the magician.)

The Trick

Anyone who reads *New Scientist* can skim-read this section, as the trick follows the same principle as that described in one of its recent articles. However, it is obviously vital to understand the trick before trying to understand the implementation.

The interesting nature of this trick stems from the fact that using a simple analysis, it would seem to be impossible. 4 cards can only encode $4!$, or 24, permutations, while there are 48 options for the fifth card. We can narrow it down by using one of the cards to pin down the suit of the fifth (there must be at least 2 cards which share a suit in

* The original attribution read: "by Gene McDonnell and Richard Smith".

the five), but then we only have 3 cards, giving 3! or 6 permutations, to account for 12 possibilities.

The secret, of course, lies in the ordering of the cards. Because the assistant gets to choose which of the five cards is hidden, she can choose such that the hidden card is within 6 of the visible card in that suit. For example, if two of the cards were the 10 and 2 of spades, hiding the 10 would not work ($10 - 2 = 8$, which is more than 6), but hiding the 3 gives a difference of 5 (J-Q-K-A-2). There is always a way to arrange two cards in a suit so this is true.

Now we can use the 6 combinations of the other three cards, combined with a suit card, to find the missing card. We take the relative sizes of the cards, and use their order to generate a number: 1 for small-medium-large, 2 for small-large-medium all the way up to 6 for large-medium-small. Then add this number to the number on the exposed card of the suit to find the missing card.

There is one small complication – what if we have, say, two 3's? We define the suits to have an order, so that the 3 of spades is 'higher' than hearts, diamonds or clubs.

The Implementations

Both of us have produced an implementation of `Esme`; Gene's is written in J and Richard's is in Dyalog APL. (Richard's is the one you may have seen in Finland, running under Pocket APL.) Both are very simple and easy to follow.

Gene's Version

```
suits=: 7&u: '♣♦♥♠' *
values=: 'A23456789TJQK'

|:{values;suits
```

* For those unable to type Unicode, enter either of the following:

```
] suits =: 4 u: 9827 9830 9829 9824
♣♦♥♠
] suits =: 4 u: 16b2663 16b2666 16b2665 16b2660
....
```

The latter (the same numbers in hexcodes) is more convenient for looking up given symbols at the website: <http://www.unicode.org/charts/>

A♣	2♣	3♣	4♣	5♣	6♣	7♣	8♣	9♣	10♣	J♣	Q♣	K♣
A♦	2♦	3♦	4♦	5♦	6♦	7♦	8♦	9♦	10♦	J♦	Q♦	K♦
A♥	2♥	3♥	4♥	5♥	6♥	7♥	8♥	9♥	10♥	J♥	Q♥	K♥
A♠	2♠	3♠	4♠	5♠	6♠	7♠	8♠	9♠	10♠	J♠	Q♠	K♠

```
lvs=: , |: {values;suits
vsi=: , @: ( , &' '>@: {&lvs)
```

Here is a sketch showing the results of the steps in performing the magical trick. (Note that because of the use of the randomising primitive (?) the actual numbers will vary.)

The argument is list of 5 distinct integers from i. 52

```
z=: 45 24 49 20 40
vsi z
7♠ Q♦ J♠ 8♦ 2♠
```

Sort the integers and append them below a 2 × 5 table of suits and values.

```
] sv=: ([ , ~ [: |: 4 13 " _ #: ]) @ /: ~) z
1 1 3 3 3
7 11 1 6 10
20 24 40 45 49
```

Get a list of sublists, each sublist giving the indices of cards having the same suit.

```
] ta=: ({. < /. 0 1 2 3 4 " _) sv
+---+-----+
|0 1|2 3 4|
+---+-----+
```

Select the sublists having more than one card.

```
] tb=: ([ # ~ 1: < [: # & > ]) ta
+---+-----+
|0 1|2 3 4|
+---+-----+
```

Select one of these sublists at random.

```
] tc=: (> @ ({ ~ ? @ #)) tb
2 3 4
```

Select two of its indices at random, preserving order.

```
] td=: ([ { ~ /: ~ @ (2 & ?) @ #) tc
2 4
```

Move the two columns with the selected indices to the front.

```
] pc=: td(([:([ , ] -. ~ 0 1 2 3 4 " _)[) ➡
{ " 1 [[: ]. ]sv
1 10 7 11 6
40 49 20 24 45
```

Find the commuted difference of the first two columns of the value row. This is positive, in the range 1-12. Take top row only of result, and append difference.

```
] cd=: ((([: - ~ / 2: {. {.} , ~ {:) @ }. )pc
40 49 20 24 45 9
```

If the tail is less than 7, delete the second item, otherwise the first.

```
] dc=: ((([: < [: < [: < {: < 7:) { ]) cd
49 20 24 45 9
```

If the tail is less than 7, leave it unaltered; otherwise replace it by 13 - tail.

```
] ce=: (([ ` (13 & | @: -) @. (7 & <:)) ➡
{: dc)4 } dc
49 20 24 45 4
```

Determine which special permutation to apply, using the tail value as determinant; drop the tail; apply the permutation.

```
] rs=: (((5 3 & p. - -. @ (2 & |)) ➡
{: ce) A. i. 4) { } : ce
24 45 49 20
vsi rs
Q♦ 7♠ J♠ 8♦
```

The magician sees that the third card is a spade, and that the other cards are in the order 1 2 0, which is the fourth permutation of order 3. The fourth card beyond J♠ is 2♠. QEF.

Richard's Version

This implementation is very much in the same mould, with `⎕IO` set to 0 so as to match the J. Here is the code for the core algorithm, together with the (totally minimal) user-interface to make it workable in the field on Pocket APL:

```

    ▽ Go;inp;deck
[1]  A Run ESME simulator
[2]  'Tell me the 5 cards ...'
[3]  'Suits are CDHS and cards'
[4]  'range from A,2 to 9,TJQKA'
[5]  'e.g. 5s 8d 3c 4h as'
[6]  ' '
[7]  Next:[]←>'
[8]  inp←1↓,[] ♦ →(pinp)↓Done
[9]  deck←uCards2Nums inp
[10] →(5≠pdeck)↑Badboy
[11] →(^/deck<52)↑Badboy
[12] 'Say these 4 ...' ♦ ' '
[13] Nums2Cards Esme deck
[14] ' ' ♦ '(in this order!)' ♦ ' '
[15] →Next
[16] Badboy:'Try to fool me eh!'
[17] 'We need 5 distinct cards here ...' ♦ ' '
[18] →Next
[19] Done:'Easy, for a PocketAPL'
    ▽

    ▽ nv←Cards2Nums str;lkp;vtv
[1]  A Look up names and return card index
[2]  lkp←,⊞names°. ,suits
[3]  vtv←1↓''(+\1,stre', /;')<',',toupper str
[4]  nv←lkpivtv
    ▽

    ▽ r←Esme cards;sv;ta;tb;tc;td;pc;cd;diff;dc;[]IO;[]ML
[1]  A Do the sorting. See MagicCD.doc
[2]  []IO←0
[3]  []ML←3 A for partition enclose
[4]  sv←{3 5p{([ω÷13),(13|ω),ω}ω[⌈ω]}cards
[5]  ta←(1+sv[0;])÷15
[6]  tb←({1≤1<pω}''ta)/ta
[7]  tc←0⇒tb[?ptb]
[8]  td←tc[{ω[⌈ω]}2?ptc]
[9]  pc←sv[1 2;td,((15)~td)]
[10] cd←pc[1 0;] ♦ diff←,--/1 2↑pc
[11] dc←((6≥diff),(6<diff),1 1 1)/cd
[12] :If 6<diff ♦ diff←13-diff ♦ :End
[13] dc←dc[0;0,1+⌈dc[1;1 2 3]] A Reorder numerically
[14] r←dc[(diff-1)⇒Δperms]
    ▽

    ▽ vtv←Nums2Cards nv;lkp
[1]  A Report names
[2]  lkp←(,⊞names°. ,suits),<'??'
[3]  vtv←tolower⌽lkp[nv]
    ▽

```

```

    tolower←{
upper←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lower←'abcdefghijklmnopqrstuvwxyz'
(lower,⊡AV)[(upper,⊡AV)ιω]
}

```

```

    toupper←{
upper←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lower←'abcdefghijklmnopqrstuvwxyz'
(upper,⊡AV)[(lower,⊡AV)ιω]
}

```

```

names suits
A23456789TJQK CDHS
Δperms
1 2 0 3 1 3 0 2 2 1 0 3 2 3 0 1 3 1 0 2 3 2 0 1
Go
Tell me the 5 cards ...
Suits are CDHS and cards
range from A,2 to 9,TJQKA
e.g. 5s 8d 3c 4h as

>5c 9d ad th 2c
Say these 4 ...

9d ad 2c th

(in this order!)

>
Easy, for a PocketAPL

```

There is still a small amount of brain-work left for the magician—in this case the thought process would be “It’s a club. *Nine, Ace, Ten is Medium, Small, Large* (Ace is low) which is 3 on our scale of 1-6 (SML→LMS) so 2+3 is the Five of Clubs.” As for the ‘inverse’ function, it could be an exercise for the reader, but it would take longer to enter the data than it takes to do the logic in your head. Anyway, people would suspect a WiFi network!

34 Greed

First published in Vector, 20, 1, (July 2003), 117-121.

Revised by Ric Sherlock, (April 2009).

My first experience of the British monetary system was in early 1953, in London. As was the case with many other visiting Americans, I felt daunted by the pound-shilling-pence currency. I had no idea what value the coins had. At a kiosk I picked up a newspaper, and then, expecting that a newspaper would be sold for just a few pennies, tendered, from the handful of coins I had, a smallish one. The vendor rapidly poured into my hand so many coins in change that I was completely unnerved. I took on faith that this was not a mistake, but I realized that I had better study the coins much more than I yet had. To this day I don't know what the coin was that produced such a flood of change. To solve the problem, I got in the habit, when I bought something, of just holding out a handful of coins, expecting that sturdy British honesty would ensure that vendors would take only the coins that would satisfy the transaction. This seemed to work quite well.

This paper discusses the problem of making change. When change is made in a store, we are used to seeing the clerk reach into the till and then take coins from its separate compartments. Unless there is a shortage of one or more coins, this is done by taking as many of the largest coin as are needed, followed by as many of the next largest, and so on, down to the penny. This almost instinctive method is called the *greedy algorithm* by computer scientists. The design of many, if not all, currency systems is such that the greedy algorithm also minimizes the number of coins that make up the amount of change.

I'll represent a set of coins by a list of the values in descending order. For example, the set of coins in the US* is 25 10 5 1; the coins are called, in order, quarter, dime, nickel and cent, or penny. The list for European coinage is 200 100 50 20 10 5 2 1. In order to be able to give just 1 pence in change it is necessary that every coinage set have as its least coin one that is worth just one penny. The change itself can

* There is a 50-cent piece in the US, but it is rarely used; the half-dollar is as rare in circulation as the two-dollar bill—which is very rare.

be represented by a corresponding list with the values showing the number of coins of each denomination used. For example, 99 pence change in the US would be represented by 3 2 0 4; in Europe by 0 0 1 2 0 1 2 0.*

Although the greedy algorithm will give the fewest coins possible for any amount of change, it is not necessarily the case that existing coinage systems give the fewest number of coins possible. Jeffrey Shallit [1] recently suggested (with tongue firmly planted in cheek) that in order to make it possible in the United States to make change using the fewest coins, and still using just four values, the coinage should be replaced by one having coins worth either 25 18 5 1 or 29 18 5 1 cents. He showed that with the existing coins, assuming all values from 0 to 99 to occur equally often, which he admits may be far from the truth, the average number of coins needed to give change is 4.7; with either of his suggested sets only 3.89 coins would be needed. He recommended the 25 18 5 1 set, since only the 10-cent piece would need to be changed. To show the benefit of the proposal, note that to give 36¢ change, the minimum number of coins with the current system is three: 1 1 0 1; with the alternative, only two are needed: 0 2 0 0.

Shallit points out that there is a problem with his suggested change, and it has to do with the failure of the greedy algorithm. For example, to give 36¢ change using the set 25 18 5 1, the greedy algorithm gives the four-coin solution 1 0 2 1, but the optimal solution, as shown above, needs only two coins: 0 2 0 0. If a coinage system was such that the greedy algorithm was not always optimal, it would require such expertise in all those who make change to do so optimally that, if for no other reason, it would simply be too impractical. In the rest of this paper I'll concentrate on a set of J functions for the greedy algorithm.

* What Eugene says about the denominations of "European" coinage applies equally well to the Eurozone "euro" (€) or the British "Sterling" (£) currency. Americans should note however that these are different currencies with a floating exchange rate. The pound Sterling (£) is nowadays divided into 100 pence (*sing.* penny), which are never called "cents". Conversely the euro (€) is divided into 100 cents, or euro-cents, which are never called "pence".

A Greedy J Algorithm

In looking for a solution to a programming problem, I frequently try to picture the steps required. For the greedy algorithm, the picture that eventually stabilized was of two linked lists: one, a list A, beginning with a list of the number of coins needed for each coin considered so far, initially empty, followed by the total amount of change needed; the second list, C, was of the coins not yet considered, initially the complete coinage set in descending order. The two lists were linked to form the list AC. Assuming these two lists were available, the processing required to obtain the next result was to replace the last item of A by the quotient and remainder of this last item divided by the leading coin of C, and C was then modified by removing its leading item. For example, assuming that a function CS was available, that performed one step of the change process, the steps in obtaining 99 pence change in the Euro system would be:*

```

EU=: 200 100 50 20 10 5 2 1
A=: (i. 0) , 99
C=: EU
] AC=: A ; C
+---+-----+
|99|200 100 50 20 10 5 2 1|
+---+-----+
] AC=: CS AC
+---+-----+
|0 99|100 50 20 10 5 2 1|
+---+-----+
] AC=: CS AC
+---+-----+
|0 0 99|50 20 10 5 2 1|
+---+-----+
] AC=: CS AC
+---+-----+
|0 0 1 49|20 10 5 2 1|
+---+-----+
] AC=: CS AC
+---+-----+
|0 0 1 2 9|10 5 2 1|
+---+-----+

```

* To get this example to work you will need to enter the collected definitions on page 292.

```

] AC=: CS AC
+-----+-----+
|0 0 1 2 0 9|5 2 1|
+-----+-----+
] AC=: CS AC
+-----+-----+
|0 0 1 2 0 1 4|2 1|
+-----+-----+
] AC=: CS AC
+-----+-----+
|0 0 1 2 0 1 2 0|1|
+-----+-----+
] AC=: CS AC
+-----+-----+
|0 0 1 2 0 1 2 0 0||
+-----+-----+

```

The process stops when there are no more coins to be used. Since the last divisor is 1, only the quotient is germane, and after razing AC, the spurious remainder is discarded, forming the result R. A vector product shows that the number of coins provided does indeed give the needed amount of change:

```

] R=: } : ; AC
0 0 1 2 0 1 2 0
EU +/ . * R
99

```

We'll start by building the nuts and bolts that go into making CS. This involves developing a new A and a new C, and linking them.

CS=: NA ; NC

The new A is formed by curtailing A and appending the result of the quotient-remainder function QR.

NA=: CA , QR

CA is straightforward:

CA=: } : @ > @ { .

QR uses the quotient-remainder primitive of J: the quotient and remainder of X divided by Y is given by (0,Y)#:X. For example, the quotient and remainder of 99 divided by 50 is 1 49. The two-part divisor is formed by appending the head of C, which is the largest remaining coin, to 0. The head of C is trivial:

HC=: { . @ > @ } .

The divisor is:

DR=: 0 , HC

The dividend is the current tail of A:

TA=: {: @ > @ {.

And QR is now easily formed:

QR=: DR #: TA

The new A is the curtailed current A appended with QR.

NA=: CA , QR

The new C is just the behead of the old C:

NC=: }. @ > @ {:

CS can now be used to obtain the successive results, as shown above. We need to find the number of times CS should be used, which is the number of coins in the coinage system, given by NS:

NS=: # @ > @ }.

A function which executes the change step function CS the correct number of times is ES:

ES=: CS ^: NS

After CS has been executed the proper number of times, the result is still two linked lists, the first list having a spurious item at the end, and the second list empty. To get the final result, the result of ES is razed and curtailed. The outermost function which encapsulates all that has preceded is MC:

MC=: }: @ ; @ ES

You'll notice, I'm sure, that this is a great many defined functions. Perhaps it says something about my attention span. I prefer to think that by making every function as simple as possible, with as few steps as are meaningful, it is easier for me to test for errors as I go along. Since J has the *fix* adverb (f.), it is easy to obtain a single longish function, which on my computer executes eleven times faster, although when displayed it probably appears quite daunting. It's not necessary to look at the result of fix, any more than one would look at the result of compiling a program written in a compiler environment.

For the diehard masochist, I offer:

```
MCf=: MC f.
] q=: 5 !: 5 < 'MCf' NB. in character form
}:@;@(((}:@>@{. , (0 , {. @>@}.) #: {:@>@{.} ; ➡
}. @>@{.:)^:(#@>@}.)
```

Notice that of the 45 tokens, ten are parentheses. The line above may make more sense if I use words for most of the tokens:

```
curtail@raze@(((curtail@A , (0 , head@C) #: ➡
tail@A) ; behead@C) ^: (tally@C))
```

For convenience, here are the functions that have been defined, in top-down order:

```
MC=: } : @ ; @ ES NB. make change: curtail raze IS
ES=: CS ^: NS NB. iterate change step NI times
NS=: # @ > @ }. NB. how many iterations: count C
CS=: NA ; NC NB. change step: new A link new C
NC=: }. @ > @ { : NB. behead C
NA=: CA , QR NB. new head: curtail A, append QR
CA=: } : @ > @ { . NB. curtail A
QR=: DR #: TA NB. divisor antibase tail A
TA=: { : @ > @ { . NB. tail A
DR=: 0 , HC NB. divisor: 0, head C
HC=: { . @ > @ } . NB. head C
```

Reference

- [1] Shallit, Jeffrey, What this country needs is an 18¢ piece. *Math. Intelligencer* 25, 2, (2003), 20-23. Also available at Shallit's website: <http://www.math.uwaterloo.ca/~shallit/papers.html>. This gives pointers to the paper in two forms: PostScript and PDF.

35 The Magical Matrix

First published in Vector, 20, 2, (October 2003), 122-126.

Revised by Roger Hui, (April 2009).

Christ! What are patterns for?

Amy Lowell, "Patterns"

Books on combinatorial subjects seem to believe that results are obtained seriatim, that the problem is to find the next combination or permutation or partition. Such is the case in the book *Combinatorial Algorithms*. [1] It is also the way the latest chapters in Knuth's *Art of Computer Programming* treat these topics [2]. Roger Hui, on the other hand, takes a more organic view, and his programs all grow an entire table from a seed. This article discusses `perm`, his algorithm for obtaining permutation tables, using a *magical matrix*. I call it that because, in studying his algorithm I stumbled hard against the critical part of his algorithm that used it, and puzzled over it for a long, long time before I could see how it worked. When I did finally understand it, all I could say was that it was magic.

Here are the tables of all the permutations from one to four:

[illegible]

The underlined portion of the 4-table is one plus the 3-table; that of the 3-table is one plus the 2-table; and even, quite trivially, that of the 2-table is 1 plus the 1-table. Hui's algorithm is recursive for arguments 2 or greater. For arguments 0 and 1 it simply returns `,.y$0`, which gives an empty table with shape `1 0` when `y` is 0, and a table of shape `1 1`, having the single value 0, when it is 1. The table of order 1 is the seed used to grow all the larger tables. If we want the table of order 4 this means that we have to go back to the seed to get the table of order 2, then the table of order 3, before we work on the one we want, of order 4.

Assuming we have the table of order 3, we can get what we need by adding 1 to it, then prefixing 0 to each row:^{*}

```
] p3=: 0,.>:perm 3  ➡
NB. prefix 0 to rows after adding 1
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
```

After studying the code that produced this, I was sure it was going to be worked on in such a way that three more analogous tables would be made and strung together with it to give the desired result. But as I looked more at Hui's function I couldn't believe what I saw: it appeared that it was going to be used as an *index*! What in the world was happening? The thing being indexed was table `mm`:

```
] mm=: \:"1=i.4
0 1 2 3
1 0 2 3
2 0 1 3
3 0 1 2
```

I was sure that `mm` should be the index and `p3` the thing indexed. I tried various ways of doing this, ending with:

^{*} The example which follows needs `perm` from page 297.

```
$qw=: mm{"2 1 p3
6 4 4
; /qw
```

```
+-----+-----+-----+-----+-----+-----+
|0 1 2 3|0 1 3 2|0 2 1 3|0 2 3 1|0 3 1 2|0 3 2 1|
|1 0 2 3|1 0 3 2|2 0 1 3|2 0 3 1|3 0 1 2|3 0 2 1|
|2 0 1 3|3 0 1 2|1 0 2 3|3 0 2 1|1 0 3 2|2 0 3 1|
|3 0 1 2|2 0 1 3|3 0 2 1|1 0 2 3|2 0 3 1|1 0 3 2|
+-----+-----+-----+-----+-----+-----+
```

This can't be right, because, for example, the row 1 0 2 3 appears three times. In order to show you how I was at last able to understand this strange indexing, I'll show the proper table of order 4 with its four sections side by side:

```
;/_6]\perm 4
+-----+-----+-----+-----+
|0 1 2 3|1 0 2 3|2 0 1 3|3 0 1 2|
|0 1 3 2|1 0 3 2|2 0 3 1|3 0 2 1|
|0 2 1 3|1 2 0 3|2 1 0 3|3 1 0 2|
|0 2 3 1|1 2 3 0|2 1 3 0|3 1 2 0|
|0 3 1 2|1 3 0 2|2 3 0 1|3 2 0 1|
|0 3 2 1|1 3 2 0|2 3 1 0|3 2 1 0|
+-----+-----+-----+-----+
```

Studying this I at last saw the pattern that Hui was using. The actors are typecast in the first section, with 0, 1, 2 and 3 playing themselves. In the second, actors 0 and 1 change roles; in the third, the three actors 0, 1 and 2 play the roles of 1, 2 and 0; in the last all four actors are in disguise: 0 acts as 1, 1 acts as 2, 2 acts as 3, and 3 acts as 0. If you look at the first rows of each of the four sections, you'll see that they are exactly the rows of *mm*, the *magical matrix*!

Now we have to study how the magical matrix was produced. The first part gets the cast of actors:

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

Classifying this gives an identity matrix:

```
=i.4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

And last, each row is replaced by its downgrade:

```
] mm=: \:"1=i.4
0 1 2 3
1 0 2 3
2 0 1 3
3 0 1 2
```

Now, if we index each row of `mm` with all of `p3`, we get an array of four 6 by 4 tables:

```
p3{"2 1 mm
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1

1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 0 2
1 3 2 0

2 0 1 3
2 0 3 1
2 1 0 3
2 1 3 0
2 3 0 1
2 3 1 0

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

Last, join the tables:

```
,/p3{"2 1 mm
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
1 0 2 3
```



```

1 0 3 2
1 2 0 3
1 2 3 0
1 3 0 2
1 3 2 0
2 0 1 3
2 0 3 1
2 1 0 3
2 1 3 0
2 3 0 1
2 3 1 0
3 0 1 2
3 0 2 1
3 1 0 2
3 1 2 0
3 2 0 1
3 2 1 0

```

And this gives the desired result. The same process works for tables of all sizes greater than one.

Here is `perm` in all its flabbergasting entirety:

```

perm=: 3 : 'if.1>:y do.,:y$0 ➡
else.,/(0,.perm&.<:y){"2 1\:"1=i.y end.'

```

Sometimes I feel that Hui has an advantage over the rest of us, even more than is given him by his native intelligence. Since he wrote it all, and keeps improving it, he must have an instinctive knowledge of the performance of each of its parts, and thus can (usually) write functions that are faster than those written by the rest of us. I suspect that indexing is one of the fastest ways to do selecting, and thus `perm` is likely to be the fastest way to build permutation tables.

References

- [1] Nijenhuis, A., Wilf, H. S., *Combinatorial Algorithms*. Academic Press, New York, (1978).
- [2] Knuth, D., (home page):
<http://www-cs-faculty.stanford.edu/~knuth/news.html>
Pre-Fascicle 2a: Generating all n-tuples (version of 29 Aug 2003)
Pre-Fascicle 2b: Generating all permutations (version of 29 Aug 2003)
Pre-Fascicle 2c: Generating all combinations (version of 29 Aug 2003)

36 Giddyap

First published in *Vector*, 20, 3, (January 2004), 117-122.

Revised by Ric Sherlock, (April 2009).

Giddyap

The *OED* doesn't have a giddyap entry; the *Concise Oxford Dictionary* has a *giddap* entry; Webster 3 has an entry for *giddap*, *giddyap*, *giddyup*. I think it must be a children's word; I don't think I've ever heard it used by an adult.¹ When I was 70 or so years younger I know that when I pretended I was riding a horse – which was surprisingly often – I swung my imaginary whip on my imaginary horse as I pranced about, shouting *giddyap* with every stroke of the whip. I find it to be a suitable title because the article deals with a problem concerning horseraces, and also treats of the speeding up of programs that solved the proposed problem. The answer to the problem turned out to be an old friend of mine.

I don't recall now where it was that I found the problem, but when I ran across it, it sounded as if it might a suitable challenge for the J Forum. In any event, on September 25 I sent this message to the J Forum:

N horses enter a race. Given the possibility of ties, how many different finishes to the horse race exist? Write a program that shows all the possibilities.

By way of example: here is the solution by brute force for N=3. There are 13 solutions. horses are named a, b and c. The expression $\{\{b,c\},a\}$ denotes a finish in which b and c tie for first and a comes in next.

$\{a, b, c\}$, $\{a, c, b\}$, $\{b, a, c\}$, $\{b, c, a\}$, $\{c, b, a\}$, $\{c, a, b\}$,
 $\{a,\{b,c\}\}$, $\{\{b,c\},a\}$, $\{b,\{a,c\}\}$, $\{\{a,c\},b\}$, $\{c,\{a,b\}\}$, $\{\{a,b\},c\}$, $\{a,b,c\}$

1 It was shortly after writing this that I attended a Choral Christmas Spectacular at San Francisco's Davies Symphony Hall with my granddaughter and wife, and heard 3000 adults singing the second stanza of a song called "Sleigh Ride", which goes:

*Giddyap, giddyap, giddyap, let's go. Let's look at the show.
We're riding in a wonderland of snow.
Giddyap, giddyap, giddyap, it's grand just holding your hand.
We're gliding along with a song of a winter fairyland.*

Notice the fourth variant in spelling of the key word.

Methods for finding how many different finishes

There were over two dozen responses over the next two weeks. The first response misunderstood the problem, and assumed that a race with 3 horses was the only one to consider. Since I had already given the solution of this one, it was clear that more had to be done than to submit the number 13. The answers to the first question, the number of solutions, were various. The brute force way is good only for the first few number of horses—the answer for eight horses is already 545,835. The nicest early entry used a table of Stirling subset numbers and factorials to give the number of different finishes effectively:

```

] fc=: !i.9
1 1 2 6 24 120 720 5040 40320
  require'math/misc/numbers'*
] s8=: subsets 8
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0
0 1 3 1 0 0 0 0 0
0 1 7 6 1 0 0 0 0
0 1 15 25 10 1 0 0 0
0 1 31 90 65 15 1 0 0
0 1 63 301 350 140 21 1 0
0 1 127 966 1701 1050 266 28 1
  s8 +/ . * fc
1 1 3 13 75 541 4683 47293 545835

```

Here's another way that uses curtailed binomial lists and the list of terms so far found:

```

1 +/ . * 1
1
1 2 +/ . * 1 1
3
1 3 3 +/ . * 1 1 3
13
1 4 6 4 +/ . * 1 1 3 13
75
1 5 10 10 5 +/ . * 1 1 3 13 75
541

```

Which can be done either iteratively or recursively.

* This line loads `subsets` into the base locale. If it doesn't work, use the J *Package Manager* to download the addon: `math/misc`.

Still another way to get the number of different finishes uses the Weighted Taylor coefficient adverb `t`: defined in *J Help > Voc* as:

The result of `u t: k` is $(!k) * u t. k$. In other words, the coefficients produced by `t`: are the Taylor coefficients *weighted* by the factorial. As a consequence, the coefficients produced by it when applied to functions of the exponential family show simple patterns. For this reason it is sometimes called the *exponential generating function*.

The exponential generating function for the our numbers is $(1/(2-e^n))$ so we can write a function `fn` using it, modified by the Weighted Taylor adverb:

```
fn=: (%@(2 - ^)) t:
fn 8
545835
fn i. 9
1 1 3 13 75 541 4683 47293 545835
```

All of these methods are discussed in Sloane's *On-Line Encyclopedia of Integer Sequences* [3], sequence 670.

Methods for representing all the possible Finishes

The method I used for representing all 13 of the finishes for a three-horse race was informal. Various methods were used in the J solutions. This is one of the J solutions for a three-horse race:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|a b c|a c b|b a c|b c a|c a b|c b a|a b c|b c a|a b c|b c a|a b c|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Results using boxed arrays, like this, were relatively slow. Faster results were obtained using a list of post positions to show the finish, with the finish order of the horse in post position k given as the value of item k of the result.

Here is how the finishes of a three-horse race are displayed using this method:

0 0 0	abc
0 1 1	a bc
1 0 0	bc a
0 0 1	ab c
0 1 0	ac b
1 0 1	b ca
1 1 0	c ab
0 1 2	a b c
0 2 1	a c b
1 0 2	b a c
1 2 0	c a b
2 0 1	b c a
2 1 0	c b a

I've appended to the right of each finish the order of finish of three horses **a**, **b** and **c**, having post positions 0, 1 and 2, respectively. Horses tied in a finish are shown by abutting letters—for example, **a bc** shows horse **a** in first place and horses **b** and **c** tied for second place.

The function to produce the finishes in this order is `fin3`, by Roger Hui:

```
rankings=: , "1 0~@i. , /:"1@=@i.@>:
ext      =: [: ,/ _1&, . {"2 1 rankings@#@~. @{.
fin3     =: ([: ; >./"1 <@ext/. ])@$:@<: ➡
` (i.@(1&,) ) @. (1&>:)
```

I found that it was difficult to understand how these functions obtained the proper result, so if you are in the same boat, I'll try to explain them. The first part of `fin3`, `([: ; >./"1 <@ext/.])`, is where most of the action occurs. Its argument is predecessor of the table to be produced. For example, to produce the table of order 3, one needs the table of order 2, which I'll call `q2`.

```
] q2=: 0 0 , 0 1 ,: 1 0
0 0
0 1
1 0
```

The left argument to `<@ext/.` is a list of the row-maxima of `q2`, or `0 1 1`. The key adverb `/.` modifying `<@exp` uses this list to partition `q2` into as many parts as there are keys, in this case two. Thus `<@ext` is applied separately to each part of `q2`, which I'll call `q2a` and `q2b`:

```

    ] q2a=: ,: 0 0
0 0
    ] q2b=: 0 1 ,: 1 0
0 1
1 0

```

The application to q2a

```

    q2aa=: (_1&,. ) q2a
    q2aa
_1 0 0
    q2aa{"2 1 rankings 1
0 0 0

0 1 1

1 0 0
    ,/q2aa{"2 1 rankings 1
0 0 0
0 1 1
1 0 0
    rankings # ~. {. q2a
0 0
1 0
0 1
    q2aa{"2 1 rankings # ~. {. q2a
0 0 0

0 1 1

1 0 0
    rankings 1
0 0
1 0
0 1
    q2ab=: rankings 1
    q2aa {"2 1 q2ab
0 0 0

0 1 1

1 0 0
    $q2aa
1 3
    q2ab
0 0
1 0
0 1

```

```
q2aa
_1 0 0
_1 0 0 { 0 0
0 0 0
_1 0 0 { 0 1
1 0 0
_1 0 0 { 1 0
0 1 1
```

Here is Hui’s explanation:

To generate the finishes for n , `fin3` first partitions the finishes for $n-1$ by the maximum ranks. Then for each partition with maximum rank m and each finish v therein, `(_1, v)` is indexed into the matrix `, "1 0~i .1+m` (tying the new competitor with each possible rank) and into the matrix `/: "1=i .2+m` (slotting the new competitor into each possible position (“no ties”)).

Therefore, if `c` is a vector of the number of finishes with maximum ranks `i.#c`, the corresponding counts for `1+#c` are: `(c*1+i.#c)` are the number of finishes for ranks `i.#c` coming from ties, and `c*2+i.#c` are the number of finishes for ranks `1+i.#c` coming from non-ties. For example, for $n=2$:

max rank	finishes	each finish indexed into	
		ties	nonties
0	0 0	0 0	1 0
			0 1
1	0 1	0 1 0	1 2 0
	1 0	0 1 1	0 2 1
			0 1 2

There is 1 finish for max rank 0 and 2 finishes for max rank 1.

The new counts for $n=3$ are:

max ranks	0	1	2
ties	1*1	2*2	
nonties		2*1	3*2
total	1	6	6

And for $n=4$:

max ranks	0	1	2	3
ties	1*1	2*6	3*6	
nonties		2*1	3*6	4*6
total	1	14	36	24

The following functions encode the algorithm:


```

ntie1=: 0: ,~ ] * 1&+@i.@#
ntie0=: 0: ,~ ] * 2&+@i.@#
nfin2=: (ntie1 + ntie0)@$:@<: ` ((,1x)"0) @. ➡
(1&>:)

```

```

      nfin2 1
1
      nfin2 2
1 2
      nfin2 3
1 6 6
      nfin2 4
1 14 36 24

```

[end of Hui's explanation].

In a race where there are no ties the order of finish is a permutation. Notice that the bottom six results give the permutation table of order 3. In the rankings function you see `/:1@=@i`. This is the “magical matrix” described in my last column (*Vector* 20.2, October 2003), and it is used in precisely the same way: to produce a table of permutations.

The results of the function `nfin2` above give the number of finishes of `n` horses having `k` as the maximum rank. If we form a triangle from these results:

```

1
1 2
1 6 6
1 14 36 24

```

and ravel it, `1 1 2 1 6 6 1 14 36 24` we get a list that is sequence 19538 in Sloane's *Online Encyclopedia*[3]. It is described as “the number of ways n labelled objects can be distributed into k nonempty parcels”. I wanted to obtain a different triangle, one showing the number of finishes having each leading digit. An easy way to do this is to look at the first column of the table of order n , as found by `f in3`. Thus if one transposes table `f in3 n`, takes its head item, applies tally modified by the key adverb and reflexive to tally one would get the number of instances of each leading digit, in order.

```

# /. ~ {. |: fin3 3
6 5 2

```

This says that the digits 0, 1 and 2 occur 6, 5 and 2 times as leading digits, respectively, in the table of order 3. You can verify this by inspecting the table above. Here are the results for tables of order 1 through 7:

1						
2	1					
6	5	2				
26	25	18	6			
150	149	134	84	24		
1082	1081	1050	870	480	120	
9366	9365	9302	8700	6600	3240	720

I’ve submitted this triangle to Sloane’s *Online Encyclopedia*.

I’ve come across the sequence given by the row sums of the triangle above, namely 1 3 13 75 541 4683 47294, in several different contexts. In 1977 I found that it enumerates the number of different left arguments for APL’s dyad transpose [1]. In 2000 it enumerated the number of distinct basic lists, I called *Blists*, which mathematicians call *preferential arrangements* [2]. And now, here they come galloping again to enumerate horserace finishes!

References

[1] McDonnell, E. E., How Shall I Transpose Thee? Let Me Count The Ways. *APL Quote Quad*, 8, 1, (1977-09).

[2] McDonnell, E. E., Blists in OLEIS. *Vector* 17, 1, (2000-07), 110-120.

[3] Sloane, N. J. A., *On-Line Encyclopedia of Integer Sequences*.
<http://www.research.att.com/~njas/sequences/>

37 Jacob's Ladder

First published in *Vector*, 20, 4, (April 2004), 84-97.

Revised by Ric Sherlock, (April 2009).

And he dreamed, and behold a ladder set up on the earth, and the top of it reached to heaven: and behold the angels of God ascending and descending on it.

Genesis 28:12

Dedicated to my grandson Jacob (15 months old).

The Name of the Game

Lewis Carroll invented a game he called *Doublets* in 1879. He used *doublet* to describe two words of the same length, which were to be connected by a *chain* of other words, called *links*. Two words are linked if they use the same letters in every position but one, like *rota* and *iota*. As an example, he gave as a doublet the words *head* and *tail*, and for the links the words *heal*, *teal*, *tell* and *tall*, so that the entire chain was *head*, *heal*, *teal*, *tell*, *tall* and *tail*. The word *Doublet* hasn't stuck, however, and the game is now usually called *Word Ladders*.

The Word Ladder game can be played mentally, and many people enjoy playing in their head, sometimes making it a game for two or more people, to see who can find a ladder quickest. This article, however, treats computer solutions to the problem, which now asks that the chain be as short as possible. There may be more than one shortest solution. For example,

```
+-----+-----+
|head|head|
|heal|heal|
|teal|heil|
|taal|hail|
|tail|tail|
+-----+-----+
```

are two solutions shorter by one link than Carroll's. Carroll would probably point out that *taal* is usually capitalized, in phrases like *the Taal*; it is a name for a language, like *English* or *Italian*, and is another name for *Afrikaans*, one of the official languages of South Africa; and *heil* is a German interjection used infamously by the Nazis in phrases like *Heil Hitler*. These words let me point out that *all* the words in my

word tables are in lower case, even names like *Hugo* and *Clive*; and that they include numerous words from foreign languages that have gained currency in English, like Russian *dvor* and Spanish *amigo*. Different word lists will give different results.

Some doublets give rise to eight or more solutions – here are the solutions for the doublet *white* and *black*.

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|white|white|white|white|white|white|white|white|white|
|whine|whine|whine|whine|whine|whine|whine|whine|whine|
|chine|chine|chine|chine|chine|chine|chine|chine|chine|
|chink|chink|chink|chink|chink|cline|cline|cline|cline|
|chick|clink|clink|clink|clink|clink|clink|clink|clink|
|click|blink|clank|clank|click|blink|clank|clank|click|
|clack|blank|blank|clack|clack|blank|blank|clack|clack|
|black|black|black|black|black|black|black|black|black|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Later on I'll bring to your attention the ladders for the doublet *dvor* and *lade*: there are *eighty* solutions to it, each nine words long!

My word tables have a history. Many years ago, in our IPSA office in Palo Alto, Joey Tuttle acquired a tape from Houghton Mifflin, the publishers of *The American Heritage Dictionary*, that contained an alphabetical listing of all the words in their dictionary. Joey extracted a number of files, each file giving all of the words having the same length, and mounted these on the I. P. Sharp computer in Toronto. I found the list useful in a number of ways, and one of the things I did was to write an APL program to form Word Ladders, of which more later. I have these word files now on my personal computer.

Structure of the Game

Looked at from the point of view of the game, a table of words is seen as an undirected graph, where the nodes are the words, and the edges are the links for each word. Linkness is symmetric: if *iota* is a link of *rota*, then vice-versa. This being the case, the graph is also symmetric, and the counterdiagonal is all zero – a word is not a link of itself. A word may have many links. For example, *bare* has 26, using my table. They are:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|babe|bake|bale|bane|base|bate|barb|bard|bari|bark|barm|barn|bars|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|care|dare|fare|hare|mare|pare|rare|tare|vare|ware|yare|bore|byre|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

A word may have no links, as well. Some four-letter hermits are *agog*, *ecru*, *idol*, *ugly*, *xmas*, *yeti* and *zarf*.

I've adopted some naming conventions. A word list – actually a character table – is called T_n , where n is the length of the words in the table. Thus my four-letter word table is T_4 . A link list, one that gives the links for each word, is called E_n . Thus, the link list for four-letter words is E_4 . Later on, in the discussion of J functions, I'll introduce some more conventions.

The Link List

There's a story to do with the creation of link lists from a word table. I wrote one for myself that took a T_n as argument, and produced the corresponding E_n . I showed this to Roger Hui, who noted that it had quadratic time. He thought it would be possible to make one that had linear time. At the time of this message we were still calling link lists *neighbour lists*.

From: rhui000@shaw.ca
 Subject: Re: Word Ladders
 Date: February 14, 2004 8:10:30 AM PST
 To: eemcd@mac.com
 Reply-To: RHui@Jsoftware.Com

There is indeed a much faster, linear, method for generating the neighbors list. The idea is this: for each word, blank out successive letters, and use these blanked out words to match for neighbors. e.g. if the words are *abba* and *abbe*,

```
_bba  _bbe
a_ba  a_be
ab_a  ab_e
abb_  abb_
```

So if you have a (m,n) matrix of words, in the matching you'd be dealing with a $((n*m),n)$ matrix and doing linear operations on it, rather than the (m,m,n) outer product.

```
m|x=: <@I.@(<:@#@[ = +/@|:@:="1)"1 2~ NB. eemcd
m|x1=: [: <@I."1 <:@{:@$ = +/@@:="1 / ~ NB. eemcd
```

```

m1x2=: 3 : 0                                NB. hui
'm n'=. $y                                  ➡
NB. # of words, # letters
i=. n (* + i.@[*"1 -. @])=i.n             ➡
NB. indices for blanking successive positions
t=. ,/ i{"_ 1 y,.'_'                      NB. blanked words
p=. ,/@:([ ,. -.~)"0 1)~                  ➡
NB. verb for pairing
j=. ; t <@p/. n#i.#y                       ➡
NB. group word indices per blanked words
h=. ({."1 </. {"1) j                       ➡
NB. unordered neighbors
k=. ~. {"1 j                               ➡
NB. word indices corresp. to h
(m{.h-.&.>k) /: k,(i.m)-.k                NB. reorder
)
```

`m1x` and `m1x1` give identical results. `m1x2` gives neighbor indices that are unordered, but is otherwise identical. The improved efficiency in `m1x2` is more pronounced as the number of words gets large.

```

alp=: a.{~97+i.26
x=: ~. alp {~ 4000 4 ?@$ #alp             ➡
NB. table of 4000 'words'
$x                                          ➡
NB. 3980 after duplicates removed
3980 41
```

```

(m1x -: /:~&.>@m1x2) x
1
(m1x1 -: /:~&.>@m1x2) x
1
```

```

ts=: 6!:2 , 7!:2@]                         ➡
NB. to find time and space used
ts 'm1x x'
12.0508 608832
ts 'm1x1 x'
2.27842 8.38869e7
ts 'm1x2 x'
0.0560371 1.40915e62
```

¹ Results may vary, due to the use of ?.

² Timings (2009) are with a modern iMac. (Ed.)

As you can see, for a table of 3980 words, Hui's program is about 160 times faster. That's a lot. It does use twice the space, but it's worth it. I've been happily using $m1 \times 2$ since to form my link lists. It's always a joy to get instant results rather than "start the process, go out and mow the lawn, then come back and maybe it will be done."

Two Schools of Thought

I'll discuss two different ways of building ladders, given a table T_n , and a links list E_n . The first way is the standard approach to finding paths in a graph: search forward from one of the words, the *starting* word, and work one's way through until the other word, the *goal* word is reached, or it is found that there is no path. This is the approach used in Edsger Dijkstra's well-known algorithm. The function that uses this forward search I call FL , for "forward ladders".

The second way, one I used a quarter-century ago, started searching from both ends of the problem. This can be done because of the problem's symmetry: if I find the shortest path from *white* to *black*, I've also found the shortest path from *black* to *white*. My intuition told me that a forward search algorithm would take significantly more space, and possibly more time, than a two-way search. Somewhat fortuitously, the two-way search can be identified with the biblical ladder in Jacob's dream, with angels going up and down. Because of this, and in honour of my grandson Jacob, I'll call the two-way search the *Jacob's Ladder* way, and the associated function JL , for "Jacob's Ladder".

My reasoning was this: suppose we use FL , starting with a one-by-one matrix, and that there are three links in every item of E_n . After one step, we have at most a three-by-two matrix; after two a nine-by-three ... and after eight we have at most a 6561-by-9 matrix. Suppose that we have now found the goal. We've looked at 6561 nodes.

If, on the other hand, we use JL , after four iterations forward and backward, we have at most two 81-by-5 matrices, meaning that we have, at most, 162 nodes. In each case we've taken eight steps. The backward steps begin with a node that is necessarily one of the 6561. The next backward step continue with 3 nodes, one of which is in that of the 2187 of the forward search. Likewise, the next backward step finds 9 nodes, one of which is in the 729 of the forward search. Next

27, one included in the 243. By the next step, finding 81 nodes, the forward search has also found 81, and one of the backward 81 nodes must be among the 81 in the forward search. Thus the forward search has worked with 40 times more nodes than the forward and backward search. I had to conclude that it would take less time and less space for the two-way search.

The Case of the Mysterious Test

I wrote FL and JL, tested them, and found that my reasoning was correct, and communicated the results to Roger Hui. His response was dumbfounding. Earlier I had sent Roger a copy of the first fifty words in T4 (see page 322). He generated a two-column table called `pairs` giving all 1225 combinations of two out of fifty, and used this as the right argument in a timing test. He made the left argument `E` using `m1x2` on the first fifty words. This is what he saw:*

```
$pairs=: 2 comb 50
1225 2
  ts 'E FL"1 pairs'
0.94485 1.1991e6
  ts 'E JL"1 pairs'
0.90665 1.1991e6
```

I tried this test on my machine and got the same kind of result: the forward search had essentially the same speed and the same use of space as Jacob's Ladder. This was completely contrary to all the earlier measurements I had made, but I couldn't deny what my own senses told me. For several days I was at a standstill – I had no idea what was wrong with my earlier measurements – or, less likely, whether there was anything wrong with Roger's measurements that I didn't understand.

The Case Solved

There was to be a happy ending however. Roger had earlier told me that he knew a way to do pathfinding that was yards better than mine, and not only that, but would find the shortest paths for all possible pairs. I asked him if he had experimented with this yet. Then this message came:

* To get these and subsequent examples to work you will need to enter the definitions of FL on page 315, JL on page 317 and `comb` on page 239.

A few minutes of experimentation with the 4-letter words reviews the fallacy of my approach. The transitive closure of that neighbors list took so long that I interrupted it after about 10 minutes. Further investigation reveals that most words are reachable from most other words. Starting from every way possible generates too much information (and too much redundant information).

If I had to choose, I'd choose the your original FL approach. JL is not sufficiently faster enough to warrant the extra complications.

This was interesting, but still left me baffled by the anomalous timing we both had seen. But then, the same day, came a new message:

Further ... The up-and-down approach is enough faster after all...

```
ts 'i=: E FL 704 1407 '
11.9041 3.04531e6
ts 'i=: E FL2 704 1407 '
2.27514 1.66336e6
ts 'i=: E JL 704 1407 '
0.881336 515264
```

The test he now used, timing for the pair 704 1407, was one that had eighty(!) nine-node solutions. The first test above used my original FL; the second used FL2, his rewriting of FL; the third used my original JL. This was very welcome news, but what about those anomalous timings? How explain them? I looked at all aspects of the data and I believe I can now explain it.

The anomalous results occurred because the first 50 words had no link list longer than 5, and most were 3 or less; the average number of links per node was 1.8; 18 out of the 50 link lists were empty, a very large percentage; thus the timings reflected an atypical set, one in which the up-and-down program couldn't show itself better than the forward search. The best demonstration is to compare the results of a test using E which has only 50 items, with tests using E4, which has 2962 items. Here are the tests using E, showing FL and JL essentially equal in time and space:

```
ts 'E FL"1 pairs '
0.94485 1.1991e6
ts 'E JL"1 pairs '
0.90665 1.1991e6
```

The only change in the next tests is using the E4 of 2962 nodes. This averages 8.1 links per node, as many as 31; 91 are empty, only 3.2%.

The distribution of links in the first 50 items of E4 is also quite different from that of E. There are 155 links, an average of 3.1 per item, and one has as many as thirteen. Only seven are empty. Here are the tests using E4:

```
ts 'E4 FL"1 pairs'
674.971 6.14182e6
ts 'E4 JL"1 pairs'
24.1225 2.28742e6
```

To me, this is conclusive. With this one change, JL is 28 times faster than FL, instead of being equal. It uses less space, less than half as much as FL.

```
675%24      NB. time ratio
28.125
6.142%2.287 NB. space ratio
2.69
```

Just for the purpose of this paper I've made some more tests, using both E4 and E5, with right argument 100 random pairs of nodes, with no node repeated. The results are what I have come to expect to get.

Tests of 100 random pairs from T4 and T5, no node repeated, 2004 03 05

```
$E4
2962
pr4=: 100 2$200?#E4      NB. 200 distinct numbers
                          NB. from i. 2962

f4=: ts 'E4 FL"1 pr4'
j4=: ts 'E4 JL"1 pr4'
f4,:j4
58.3915 4.19264e6      NB. FL numbers
4.08996 2.68454e6      NB. JL numbers
f4%j4      NB. ratio of test numbers
14.2768 1.56177      NB. JL 14 times faster,
                          NB. 1.5 times less space

$E5
5604
pr5=: 100 2$200?#E5      NB. 200 distinct numbers
                          NB. from i. 5604

f5=: ts 'E5 FL"1 pr5'
j5=: ts 'E5 JL"1 pr5'
f5,:j5
69.0011 2.84186e6      NB. FL numbers
4.43224 843456      NB. JL numbers
f5%j5      NB. ratio of test numbers
15.568 3.3693      NB. JL 16 times faster.
                          NB. 3.4 times less space
```

Function Syntax and Use

The versions of FL and JL I wrote were gone over and tightened up by Roger Hui. Their syntax is:

```
R=: En FL a,b
R=: En JL a,b
```

Where *a* and *b* are indices of words in some *Tn*. The result is an integer table, where each row is distinct, and the successive values in a row are pairwise links, with first item *a* and last item *b*.

```
] R=: E4 JL 2182 861
2182 628 617 616 589 810 859 861
2182 1505 1483 1455 812 810 859 861
2182 2619 2608 2573 812 810 859 861
```

To obtain the desired word ladder, use this result as an index to *T4*:

```
<"2 T4 {~ R
```

For example,

```
<"2 T4 {~ E4 JL 2182 861
+-----+-----+-----+
|rips|rips|rips|
|dips|lips|tips|
|dies|lies|ties|
|died|lees|tees|
|deed|fees|fees|
|feed|feed|feed|
|fled|fled|fled|
|flew|flew|flew|
+-----+-----+-----+
```

Forward March

The program FL is easier to explain than the more intricate JL.

```
FL=: 4 : 0
's e'=. y
u=.d=. ,s
c=. ,.s
while. -. e e. d do.
  d=. ; v=. (d{x) -. &.> < u
  if. '' -: d do. _1 return. end.
  u=. u , ~. d
  c=. ((#&>v)#c) ,. d
end.
c #~ e=d
)
```

The two word indices are **s** and **e**. In **FL**, these signify start and end, but in **JL** they don't have that mnemonic significance, since they start separate chains. The variable **x** is the links list, some **En**.

Variable **d** is an integer list, initially **, s**. It is used to select boxed lists of potential new links. The first time through it gets all the links of **s**. Next time it gets all the links of those links, and so forth. Variable **u** contains all of the links already seen. No link appears in **u** more than once. Initially it is **, s**. It is used to ensure that no later use is made of a link that has already been used. This is because once a link is used in any step, there is no point to using it again in a new step—any chain with a later appearance of an earlier link must be longer than one with an earlier appearance.

Variable **c** is the table of chains, initially with **s** as its only value. Within the **while** loop it will be extended. Each of its rows represents a potential shortest path.

The **while** loop continues until **d** contains **e** as an item; when it does, it means that one or more shortest paths have been found.

Variable **d** is used to select boxed lists of links from **x**. Before further use, each box has removed from it all links that have already been used. These cleaned-up boxes are assigned to **v**, the raze of which becomes the new link selector list **d**.

If **d** is empty, there are no new links, and since **e** hasn't been found, we'll have to admit that there are no paths between **s** and **e**. When this happens, the scalar **_1** is returned.

Variable **u** is updated with all the new links, by appending **d** to it. Because **u** should never contain two appearance of the same link, duplicates are removed from **d** before appending it.

Table **c** is updated by adding a new column, with the items of **d**.

When the **while** loop ends, a selecting mask is formed by the equality of scalar **e** and list **d**, and this mask is used to remove from **c** all those rows not ending in **e**. This gives the desired result.

The Angels of God Ascending and Descending

Since JL goes forwards and backwards, there are separate variables for the forward and backward sequences. Instead of *c* we have *sc* and *ec* – two chains; instead of *u* and *d* we have *su* and *eu*, *sd* and *ed*.

The *while.* is different—it says, effectively, “while forever”, since the loop continues as long as the value of the *while.* phrase is 1. The exiting from the loop will take place by way of *if.* statements.

```
JL=: 4 : 0
sc=. ,. su=.sd=. ,{.y
ec=. ,. eu=.ed=. ,{:y
while. 1 do.
  if. +./ sd e. ed do. break. end.
  if. ' ' -: sd=. ; v=. (sd{x) -.&.> <su ➡
do. _1 return. end.
  su=. ~.su,sd [ sc=. sd ,.~ (#&>v)#sc
  if. +./ sd e. ed do. break. end.
  if. ' ' -: ed=. ; v=. (ed{x) -.&.> <eu ➡
do. _1 return. end.
  eu=. ~.eu,ed [ ec=. ed ,. (#&>v)#ec
end.
sc join ec
)
```

The variables ending in *u*, *d* and *c* have the same functions as the *u*, *d* and *c* variables in FL. The first and the last three statements in the *while.* loop have almost identical structure. With two path tables being built in the same loop, the test for termination is by finding that the same item or items appear in *sd* and *ed*. The test for “no path found” is essentially the same as in FL, but there are separate ones for *sd* and *ed*. An important difference is that *sc* is built from left to right, but *ec* is built from right to left. This makes joining the two easier.

Linking the Chains

When the *while.* of JL terminates, the fitting together of the two path tables requires some agility. It is complicated enough so that a special *join* function has been made. It was written by Roger Hui as a re-write of a *joinends* function provided to me by R. E. Boss when I sent a message to the J forum explaining the problem and asking for a solution.

```
join=: 4 : 0
x=. x#~ ({:"1 x) e. {"1 y
y=. y#~ ({:"1 y) e. {"1 x
(({:"1 i){}:"1 x) ,. ({:"1 i){y [ ➡
i=. (0,#y)#:I.,({:"1 x)=/{"1 y
)
```

The variables *x* and *y* are the forward and backward chains, respectively. We have reached this point because one or more items of the last column of *x* and the first column of *y* match. We want to keep only those rows containing these matching items. The first two lines remove from *x* and *y* all the rows that don't have matching values in them. I'll invent an *x* and a *y* and go slowly through the steps that lead to the desired result.

Here are the two:

```
] x=: 7 3$ 200 300 400 0 1 130 2 3 120 ➡
4 5 130 6 7 120 8 9 130 500 600 700 ➡
200 300 400
0 1 130
2 3 120
4 5 130
6 7 120
8 9 130
500 600 700

] y =: 8 4$ 500 600 700 800 130 2 1 0 ➡
120 5 4 3 120 8 7 6 120 11 10 9 ➡
120 14 13 12 130 17 16 15 900 1000 1100 1200
500 600 700 800
130 2 1 0
120 5 4 3
120 8 7 6
120 11 10 9
120 14 13 12
130 17 16 15
900 1000 1100 1200
```

Only five rows of *x* and six of *y* match. First we remove the rows of *x* that don't end in one of the matching values:

```
] x=. x#~ ({:"1 x) e. {"1 y
0 1 130
2 3 120
4 5 130
6 7 120
8 9 130
```

And similarly, remove the rows of y that don't begin with one of the matching values.

```
] y=. y#~ ({."1 y) e. {"1 x
130 2 1 0
120 5 4 3
120 8 7 6
120 11 10 9
120 14 13 12
130 17 16 15
```

Now we have to maneuver to get the rows of x ending in 120 in line with those of y beginning with 120, and similarly for 130. First we compare for equality the tail of each row of x with the head of each row of y :

```
({"1 x)=/{"1 y
1 0 0 0 0 1
0 1 1 1 1 0
1 0 0 0 0 1
0 1 1 1 1 0
1 0 0 0 0 1
```

This is ravelled and the indices of 1s found:

```
I.,({"1 x)=/{"1 y
0 5 7 8 9 10 12 17 19 20 21 22 24 29
```

We convert this into their base $\#y$ representation.

```
] i=. (0,#y)#:I.,({"1 x)=/{"1 y
0 0
0 5
1 1
1 2
1 3
1 4
2 0
2 5
3 1
3 2
3 3
3 4
4 0
4 5
```

It's another Magical Matrix [1]. The first column is used to select rows from *x* and the second to select rows from *y* .

Use the second column to select rows from *y* in the right quantity and in the right order:

```
({"1 i){y
130 2 1 0
130 17 16 15
120 5 4 3
120 8 7 6
120 11 10 9
120 14 13 12
130 2 1 0
130 17 16 15
120 5 4 3
120 8 7 6
120 11 10 9
120 14 13 12
130 2 1 0
130 17 16 15
```

and use the first column to select rows of *x*, at the same time removing the last column; it merely repeats the first column of *y*.

```
(({"1 i){}:"1 x)
0 1
0 1
2 3
2 3
2 3
2 3
4 5
4 5
6 7
6 7
6 7
6 7
8 9
8 9
```


Lastly, stitch these together, and we have the desired result:

```
(({"1 i"){: "1 x) ,. ({"1 i){y [ ➡
i=. (0,#y)#:I.,({"1 x)=/{"1 y
0 1 130 2 1 0
0 1 130 17 16 15
2 3 120 5 4 3
2 3 120 8 7 6
2 3 120 11 10 9
2 3 120 14 13 12
4 5 130 2 1 0
4 5 130 17 16 15
6 7 120 5 4 3
6 7 120 8 7 6
6 7 120 11 10 9
6 7 120 14 13 12
8 9 130 2 1 0
8 9 130 17 16 15
```

And we've matched the three 130s in the last column of *x* with the two 130s in the first column of *y*, giving six rows; and matched the two 120s in the last column of *x* with the four 120s in the first column of *y*, giving eight rows, which makes fourteen rows altogether. This contrived example shows a solution where there are fourteen different ladders with six links each, giving the links we can use to select the words that make Word Ladders.

Acknowledgements

R. E. Boss provided a workable joining function on the same day that I sent a message to the J forum asking for one. Norman Thomson gave me many ideas about finding paths in graphs. Roger Hui took my ugly ducklings and turned them into beautiful swans, and, by jumping to a conclusion, made it possible for me to understand better the structure of the problem.

Reference

- [1] McDonnell, E. E., The Magical Matrix. *Vector* 20, 2, (2003-08), 122-126.

Appendix

The list of 50 words from table T4 that Eugene sent to Roger Hui:

```
T4=: _4]\ (' ',LF)-.~ ]0 : 0
'tis abba abbe abed abel abet abib able ably abut
ache acid acme acne acre acts acyl adak adam adar
adds aden adit aery afar afro agar aged ages agio
agni agog agra ague ahab ahem ahoy aide aids aims
aine aire airs airt airy ajar ajax akee akin alai
)
```

38 The Google Test

First published in Vector, 21, 1, (Autumn 2004), 116-122.

Revised by Brian Schott, (April 2009).

Highway 101 in the San Francisco bay area is a busy commuter highway, with employees commuting to work at the headquarters of such companies as Adobe, Apple, Applied Materials, Cisco, eBay, Genentech, Google, Hewlett Packard, Informatics, Intuit, Oracle, Silicon Graphics, Sun Microsystems, Yahoo, and hundreds more high-tech companies. These commuters recently drove past a large poster paid for by Google, reading:

{first 10-digit prime found in consecutive digits of e }.com

Google apparently trusted that some among those passing the poster would understand it, and of these some might be intrigued enough by it to see if they could find that prime, and perhaps some of them might use it to go to the resulting web address. Those who did go the whole route would then find themselves with this message:

Congratulations. You've made it to level 2. Go to **www.linux.org** and enter *Bobsyouruncle* as the login and the answer to this equation as the password.

$$F(1) = 7182818284$$

$$F(2) = 8182845094$$

$$F(3) = 8747135266$$

$$F(4) = 7427466391$$

$$F(5) = \underline{\hspace{2cm}}$$

Those who find the value of $F(5)$, and go to the site shown, would get this message from Google Labs:

Congratulations.

Nice work. Well done. Mazel tov. You've made it to Google Labs and we're glad you're here.

One thing we learned while building Google is that it's easier to find what you're looking for if it comes looking for you. What we're looking for are the best engineers in the world. And here you are.

As you can imagine, we get many, many resumes every day, so we developed this little process to increase the signal to noise ratio. We

apologize for taking so much of your time just to ask you to consider working with us. We hope you'll feel it was worthwhile when you look at some of the interesting projects we're developing right now. You'll find links to more information about our efforts below, but before you get immersed in machine learning and genetic algorithms, please send your resume to us at problem-solver@google.com.

We're tackling a lot of engineering challenges that may not actually be solvable. If they are, they'll change a lot of things. If they're not, well, it will be fun to try anyway. We could use your big, magnificent brain to help us find out.

You now have all you need to know to dazzle Google with your magnificent brain. I haven't spoiled it for you, so you can legitimately do it on your own. I will, however, give you a similar puzzle, in two parts, and will solve it for you. It uses the digits of π .

Problem 1: Finding 10-digit primes

The first problem is to find among the digits of π a ten-digit sequence that, when evaluated in base ten, is a prime number, and is the eighth such. Your first problem, then, is to obtain the first few hundred or so digits of π . We're in luck! The great Indian mathematician Ramanujan used the theory of complex multiplication of elliptic curves to give a number of beautiful formulas for π 's digits, and a variation of this technique was discovered by the ingenious Chudnovsky brothers, from New York City by way of Kiev. A J function for their method is:

```
bigpi=: 3 : 0
a=. 545140134x
b=. 640320x ^ 3
c=. 13591409x
d=. 6541681608x
n=. i. >: x: y
s=. (! 6 *n) * c + a * n
e=. (! 3 * n) * ((! n) ^ 3) * b^n
m=. {: e
f=. d * - / (s * m) % e
k=. (a * m) * <. @ %: b * 10x ^ 28 * y
k <. @ % f
)
```

Given an integer argument n it finds $1+14*n$ places of π . To solve the problem you should use a hundred or so digits, so 15 would give about the right number.

```
q=: bigpi 15
```

To work with the individual digits it is convenient to work with the character form of q .

```
# w=: ": q
211
```

Here are the first 210 digits:

```
7 30 $ w
314159265358979323846264338327
950288419716939937510582097494
459230781640628620899862803482
534211706798214808651328230664
709384460955058223172535940812
848111745028410270193852110555
964462294895493038196442881097
```

We need these ten at a time:

```
$ t=: 10 [ \ w
202 10
5 {. t
3141592653
1415926535
4159265358
1592653589
5926535897
```

To determine which of these are prime we have to convert each row into a number.

```
$ p=: ". t
202
5 {. x: p
3141592653 1415926535 4159265358 1592653589 ➡
5926535897
```

Some of these may have had leading zeros, so that they are effectively 9-digits long. We remove them—some of these may be prime, but they don't qualify as ten-digit primes.

```
$ pa=: (p > 999999999) # p
183
```

A convenient way to determine whether a number is prime is to count how many prime factors it has; if it has just one, the number is prime. We can think of using J's *prime factors* primitive q : to obtain the factors of the numbers in p , but will find that this is not always possible; many of the numbers are outside its domain.

```

    q: 2004
2 2 3 167
    # q: 2004
4
    q: 2003
2003
    # q: 2003
1

```

So 2003 is prime, and 2004 isn't. Here is an is-prime function `ip`:

```

ip=: 13 : '1 = # q: y'
ip 2003
1
ip 2004
0
ip 2000000000
0
ip 2100000000
0
ip 2200000000
|domain error: ip1
| ip 2.2e9
2200000000 = * / 11 5 5 5 5 5 5 5 5 2 2 2 2 2 2 2 2
1

```

At this writing, the `q:` function will yield a domain error for many integers with ten or more digits, even when it is obvious the number isn't prime.

To circumvent this, we can use J's adverse conjunction, defined as "The result of `u :: v` is that of `u`, provided that `u` completes without error; otherwise the result is the result of `v`." We can write a special is prime function `sip` to return `_1` as result if otherwise a domain error would be reported:

```

sip=: ip :: _1:
sip 2200000000
_12

```

We use `sip` on `pa` to let us know which are known composites (0), which are known primes (1), and which are not determined yet (`_1`).

¹ In J602a this no longer produces an error.

² In J602a this now returns the value 0.

```

pb=: sip " 0 pa
~. pb
_1 0 1
# /. ~ pb NB. how many of each
159 23 1

```

We can remove the known composites, leaving us with just the known primes and the suspects.

```

pc=: (pb ~: 0) # pa
$ pc
160

```

Now comes the hard part: determining which of these are prime without the use of (q:).*

A number is composite if it has two or more prime factors. Twenty-five is composite since it has the two factors 5 and 5. The larger number 9999399973 is also composite, with the two factors 99991 and 100003.

A prime number z has only one prime factor, namely z . A composite number w must have one or more prime factors less than its square root r . It can only have one prime factor s larger than its square root r . It may be that all of its prime factors are less than r . In any case, to find whether a number is prime or not it suffices to see whether any of the primes less than its square root divides it, that is, gives a residue of zero with n . Thus, if we have a list pf of all of the primes less than r we will be able to determine whether a number n is composite by seeing if it has a residue of zero for any of pf . If it doesn't then we know that n is a prime.

We know that the numbers we are interested in will have values from 1000000000 to 9999999999, inclusive. The square root of the largest number we may find is thus less than 100000, so it suffices that pf contain just the primes less than 100000.

* Still a good exercise, even if the J error which made it necessary has been fixed.

We can find these easily by using the function inverse to `p:`, that is,
`p: ^: _1.`

```
p: ^: _1 [ 100000
9592
p: 9592
100003
p: <: 9592
99991
pf=: p: i. 9592
{: pf
99991
```

We need a function that finds the residues of a number with respect to each prime in `pf` and gives 0 if the number is composite and 1 if it is not, that is, if it is prime.

```
nc=: 13 : '-. 0 e. pf | y'"0
```

This reads “not 0 in `pf` residues of `y`”. We apply it to `pc`:

```
pd=: nc pc
```

How many 10-digit primes have we found?

```
+/pd
9
```

More than enough to solve the puzzle. Which are they?

```
] pe=: I. pd
2 33 36 43 73 108 128 135 149
```

And what are they?*

```
pg=: x: pc { ~ pe
,. x: pg
5926535897
4197169399
1693993751
7510582097
4825342117
5822317253
2841027019
8521105559
8954930381
```

* Because of improvements in J602a all that is needed now is `,. x: pc`. Likewise `pe` takes the value 0 1 2 3 4 5 6 7 8, not as shown.

Just for fun, locate these in the digits of π :

```

      7 30 $ w
314159265358979323846264338327
950288419716939937510582097494
459230781640628620899862803482
534211706798214808651328230664
709384460955058223172535940812
848111745028410270193852110555
964462294895493038196442881097

```

Three of them overlap. We want the eighth, 8521105559.

Problem 2: Finding the sixth in a series

You are given five 10-digit numbers from the digits of π , and must find the sixth. Here are the numbers:

```

4338327950
2795028841
6939937510
3993751058
2110555964

```

Here they are, embedded in π :

```

      7 30 $ w
314159265358979323846264338327
950288419716939937510582097494
459230781640628620899862803482
534211706798214808651328230664
709384460955058223172535940812
848111745028410270193852110555
964462294895493038196442881097

```

The first and second overlap, as do the third and fourth.

I'll give two hints, the second vacuous:

Hint 1: Primarily, the sixth number has three doublets and overlaps the fifth.

Hint 2: Alternately, something for nothing.

Answer on page 354.

39 Metlov's Triumph

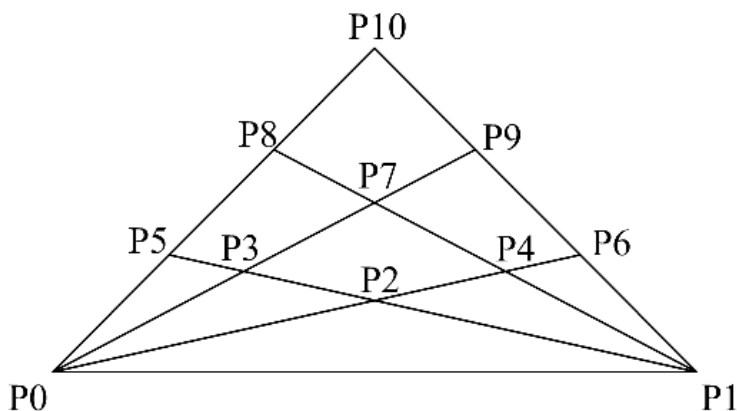
First published in Vector, 21, 4, (Autumn 2005), 25-30.

Revised by Boyko Bantchev, (April 2009).

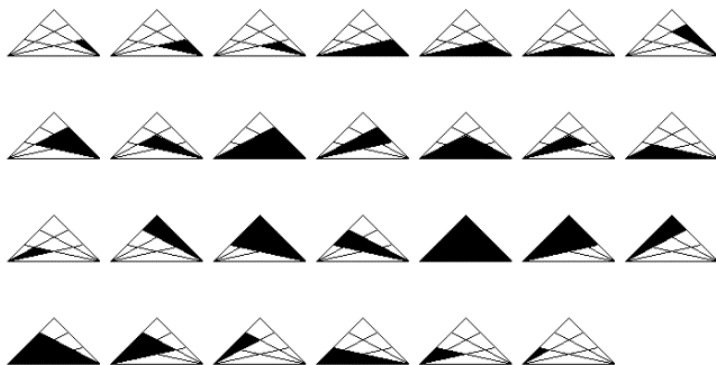
A puzzle was recently announced by Frank Buss on the Internet that led to some interesting discoveries. The puzzle is to be found by Googling "Frank Buss Triangle Problem" and then clicking on "Triangles Challenge" or browsing directly to

<http://www.frank-buss.de/challenge/> if you prefer. It says:

The challenge is to write a program, which counts all triangles with area > 0 in this figure:



But count only the triangles, which are bounded by lines, like (P0, P7, P8), not all possible connections between the points, like (P7, P8, P9). If anything is unclear, the solution is 27 and looks like this:



Graphic output is not needed, but you can do what you want. If a GUI or something else is included, it would be nice to write: how long you needed for the pure algorithm and for the rest.

This is not a quantitative, but more a qualitative challenge. Neither the number of lines nor the time (which I can't verify anyway) is important, but I'm interested in good solutions, which show the advantages of the chosen language.

Every program should be documented enough to understand how it works and it should not simply print 27, but somewhere it should read from a file or integrate the points and geometry, so that it is easy to change it for similar problems, for example if another line is added, but it need not to be so general as to count the number of squares.

There were 31 entries. The languages they used, the number of entries in that language, and the average number of lines in the programs are tabulated below:

Language	Number of entries	Average number of lines
C++	3	115
Java	4	105
Python	1	94
Haskell	1	93
Ruby	1	75
Scheme	1	66
Awk	1	59
Lisp	17	56
Kogut	1	29
J	1	1

Most of these had a generous amount of documentation along with the actual program. I don't know most of the languages used, but I could come to some conclusions about them. It seems

to me that most of the authors were more programmers than mathematicians. Almost all of them tackled the problem as one of establishing the proper way to represent the points, lines, and intersections in the triangle. Most of them gave solutions which were wired in, and their programs could not easily be extended to variations of the problem.

Since Haskell is supposed to be a functional programming language, I thought it might give an interesting and useful result, but I was disappointed. It hard-coded the geometry of the problem, so that it, like many others, couldn't be extended.

The J solution was submitted by Dr. K. L. Metlov. Here it is:

```
* -: @ * +
```

Metlov is a physicist, with many publications in his field, and he obviously studied the triangle puzzle as a mathematical one. In his notes, one sees that he experiments with variations of the problem, and in a relatively short time had concluded that a simple expression could be formed that would apply to a triangle with any number of lines.

This is a fork, and a dyad, and it is better understood by emphasizing its forkiness.

```
  *   +
  \   /
  |
  *
  |
  -:
```

The arguments are multiplied and added, this product and sum are multiplied, and the product is halved.

I give Metlov's Documentation over the next three pages:

"When both sides of the triangle are divided into an equal number of steps (let's call this number - n), the number of triangles is n^3 (n to the third power). For the example Frank Buss gives $n=3$ and the answer is $3^3 = 27$.

When sides are subdivided into a different number of subdivisions, say, n and m , the number of triangles is equal to

$$\frac{1}{2}m \times n(m + n)$$

which is integer for any integer m and n .

In J language (see <http://www.jsoftware.com/> for description and download) the first formula is coded and invoked as

```
nt=: ^&3
nt 3
27
```

The second formula is coded and invoked as

```
nt=: * -: @ * +
3 nt 3
27
2 nt 5
35
```

The first variant of the program is three characters, the second is 6 characters.

It took me 15-20 minutes of drawing rectangles to derive the formula. J is an array-oriented language, descendent of APL. Therefore, the above programs (without change) can indeed be used to process millions of rectangles very fast. In order to achieve this the arguments must be arrays (of equal length in the second case). For example:

```
(3 2) nt (3 5)
27 35
```

How the formula was developed:

Here is the link to the page of notes I made when thinking about the problem.

http://www.livejournal.com/users/dr_klm/51584.html?thread=435072#t435072

and [overleaf] is a copy of the page.

The direct link is here:

http://galaxy.fzu.cz/~metlov/Triangles_Deriv.gif

Metlov's Triumph

$$N = 2n + 1$$

$$n = m \cdot n + (n-1) \cdot m$$

$$N = \left(2 + \frac{1}{2} + \frac{1}{2} \right) + \frac{1}{2} + \frac{1}{2} = 3 + 1 = 4$$

$$2 \cdot \left(1 + 2 + \frac{1}{2} + \frac{1}{2} \right) + \frac{1}{2} + \frac{1}{2} = 8$$

$$2 \cdot \left[\frac{3+3+3 \cdot \frac{1}{2}}{m \cdot (n-2) + \frac{m}{2}} + \frac{3+3 \cdot \frac{1}{2}}{m \cdot (n-1) + \frac{m}{2}} + \frac{3 \cdot \frac{1}{2}}{m \cdot (n-3) + \frac{m}{2}} \right] =$$

$$= 15 + 9 + 3 = 27 (!)$$

$$= \sum_{i=1}^n \left[m(n-i) + \frac{m}{2} \right] =$$

$$= \frac{m(n-1)}{2} \cdot n + \frac{mn}{2} =$$

$$= \frac{mn^2}{2}$$

$$N_1 = 2 \cdot \frac{n \cdot n^2}{2} = n^3$$

$$N_2 = \frac{mn^2}{2} + \frac{nm^2}{2} = \frac{mn(m+n)}{2}$$

I do not know if that will be enough to communicate the basic idea used for deriving the formula. On the other hand I do not have time to explain it in full detail.

The interesting part occupies the lower left quarter of the page. Triangles are counted separately for two lower corners of the big triangle (left and right) and then the result is multiplied by 2 (if $n=m$), or added up with exchanging $n \leftrightarrow m$ (if $n \neq m$). To count triangles for one corner I sum up the triangles, occupying all single sub-sectors, the triangles, occupying all pairs of two consecutive sub-sectors, ...three sub-sectors... etc. In this sum, the triangles, which include both left and right corners of the big triangle, are counted with weight $\frac{1}{2}$ (to note that they will be counted again, in the sum for the other corner).

I ran this procedure for $n=3$, $m=3$ approximately in the middle of the page. Then, by induction, wrote a general formula with the sum. The sum is nothing else but an arithmetic progression, which is immediately summed up. Then, with very basic algebra, the final formulas are obtained."

Comment from Frank Buss:

This is a nice solution and the language looks interesting. It is the same concept as the Scheme solution, which uses a formula instead of counting the triangles, but this formula is much easier than the one used in the Scheme solution.

40 Belgian Numbers

First published in Vector, 22, 1, (November 2005), 96-101.

Revised by Boyko Bantchev, (April 2009).

Belgian numbers were recently introduced by Eric Angelini. For N to be a Belgian number, it is necessary that it appear in the sum scan of lots of replications of N 's digits. For example, try the digits of 16:

```
+/\8 $ 1 6
1 7 8 14 15 21 22 28
```

So 16 isn't Belgian. What about 17?

```
+/\8 $ 1 7
1 8 9 16 17 24 25 32
```

So 17 is Belgian.

My first concern on hearing of them was to find out how one could tell whether a number was Belgian or not. Assume that N is 176; how many copies of 1 7 6 would be needed to reach 176 or thereabouts? This (C) can be found by taking the ceiling of N divided by the sum of its digits S . Like this:

```
N=: 176
] D=: 10&#. ^: _1 N
1 7 6
] S=: +/D NB. Sum of digits of N
14
] C=: >. N % S NB. Number of copies needed
13
```

Then multiplying this by its number of digits, in this case, 3, and sum scanning:

```
] R=: +/\(C * #D) $ D
1 8 14 15 22 28 29 36 42 43 50 56 57 64 70 71 ➡
78 84 85 92 98 99 106 112 113 120 126 127 134 ➡
140 141 148 154 155 162 168 169 176 182
```

and, sure enough, we find that 176 is a Belgian number. This was, for the moment, an easy way to determine which numbers were Belgian—if they were not too big. I thought it might be possible to use it for numbers up to about ten million—but thereafter it gets ridiculous.

For example, suppose that one wants to see whether 1234567898765 is Belgian:

```
N=: 1234567898765x
] D=: 10&#. ^: _1 N
1 2 3 4 5 6 7 8 9 8 7 6 5
#D
13
] S=: +/D NB. Sum of digits of N
71
] C=: >. N % S NB. Number of copies needed
17388280265
(C * #D) NB. as in expression: +/\(C * #D) $ D
226047643445
```

Ouch. A fifth or so of a trillion (US style) digits. I concluded that this was impractical.

There had to be a better way. Eventually I thought I saw that in the case of 176 there was a threeness lurking. I decided to form them into a table of three-item rows, together with the first difference of the rows, and the 14-residues of the whole table:

qn=: 13 3 \$ + / \ 39 \$ 1 7 6

qn ; (2--/\qn);14|qn

+-----+-----+-----+		
1 8 14	14 14 14	1 8 0
15 22 28	14 14 14	1 8 0
29 36 42	14 14 14	1 8 0
43 50 56	14 14 14	1 8 0
57 64 70	14 14 14	1 8 0
71 78 84	14 14 14	1 8 0
85 92 98	14 14 14	1 8 0
99 106 112	14 14 14	1 8 0
113 120 126	14 14 14	1 8 0
127 134 140	14 14 14	1 8 0
141 148 154	14 14 14	1 8 0
155 162 168	14 14 14	1 8 0
169 176 182		1 8 0
+-----+-----+-----+		

The first row of the first table shows that the sum of the integers is 14; the second table shows that each row differs from the previous row by 14, the third table shows that the 14-residue of each row is 1 8 0.

The point made by the second table is that each row after the first is formed by adding 14 to the preceding row. This means that

I could create qn exactly by adding multiples of 14 to the starting row:

$$(14 * i. 13) + / 1 \ 8 \ 14$$

1	8	14
15	22	28
29	36	42
43	50	56
57	64	70
71	78	84
85	92	98
99	106	112
113	120	126
127	134	140
141	148	154
155	162	168
169	176	182

The point made by the third table is that any concern about residues after the first row is irrelevant.

With this information at hand, we're almost there. The number 176 is 8 beyond 168, which is $14 * 12$. The $14 * 12$ is irrelevant; the 8 is significant. It means that 8, which is $14 | 176$, is of great importance. It is one of the numbers in the first row of the table, and after being added to by multiple 14s, must arrive at 176, and so, is Belgian! This is a perfectly general observation, and allows any integer to be tested for Belgianity. Here is an outline of the steps to take to see if N is Belgian:

- Let D be the digits of N .
- Let S be the sum-scan of the digits of N .
- Let T be the sum of D (which is also the last item of S).
- Let R be the T -residue of N .
- Then N is Belgian if R is in S .

For the case of $N = 176$, we have:

- D is 1 7 6
- S is 1 8 14
- T is 14
- R is the T -residue of N , or 8.

We ask: is R in S ? The answer is Yes!

The general case is that N is Belgian if:

$$(M \mid N) \text{ e. } S$$

is 1.

Now it's clear that 176 is Belgian if any of these residues is the same as $14 \mid 176$:

$$14 \mid 176$$

8

It is certain that if one adds the proper multiple of 14 to 8, the result must be 176. It isn't necessary to find what this multiple is. It is sufficient that there is an 8 in the sum scan of 1 7 6.

Now we can write an expression to let us determine whether 176 is Belgian:

$$(14 \mid 176) \text{ e. } + / \setminus 1 \ 7 \ 6$$

1

Another way of doing this is: since 8 is the 14 residue of 176, it is sufficient to see whether there is a zero in the 14 residue of

$$176 - + / \setminus 1 \ 7 \ 6$$

$$14 \mid 176 - 1 \ 8 \ 14$$

7 0 8

So we may write:

$$0 \text{ e. } 14 \mid 176 - + / \setminus 1 \ 7 \ 6$$

1

to determine that 176 is Belgian.

We can now work with large numbers with some confidence. First, we'll need a way of getting the integer components of N. The function `dig` works very well:

$$\text{dig}=: 1j1 \& \# \&. ": @ x:$$

We'll use this on 1234567898765:

$$\text{dig } 1234567898765$$

1 2 3 4 5 6 7 8 9 8 7 6 5

$$+ / \setminus \text{dig } 1234567898765$$

1 3 6 10 15 21 28 36 45 53 60 66 71

$$+/1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 8 \ 7 \ 6 \ 5$$

71

$$71 \mid 1234567898765 - + / \setminus \text{dig } 1234567898765$$

20 18 15 11 6 0 64 56 47 39 32 26 21

There is a zero in this list, indicating that N is Belgian. We can verify this.

```

N=: 1234567898765x
] S=: + / \ dig N
1 3 6 10 15 21 28 36 45 53 60 66 71
] M=: {: S
71
] Q=: <. N % M
17388280264
] P=: Q * M
1234567898744
] R=: S + P
1234567898745 1234567898747 1234567898750 →
1234567898754 1234567898759 1234567898765 →
1234567898772 1234567898780 1234567898789 →
1234567898797 1234567898804 1234567898810 →
1234567898815
N e. R
1

```

So 1234567898765 is Belgian.

Here is a function that will determine whether or not a number is Belgian:

```

BN=: 3 : 0 " 0
N =. x: ^: (y >: 2147483648) y
'S M' =. (] ; {:) + / \ (1j1 & # &. ":) N
(M | N) e. S
)

```

The first line makes the argument an extended integer N if it is larger than the largest positive 32-bit integer, and otherwise leaves it alone. M and S are modulus and digits sum, as discussed above.

See if BN agrees that N is Belgian:

```

BN 1234567898765
1

```

Why Belgian?

Why Belgian, indeed. Well, there are Roman numbers, Arabic numbers, even Catalan numbers. It's about time Belgium was recognized. I could pretend that Eric Angelini (the creator of these numbers) knew that my wife Jeanne was a Fulbright Scholar at

the University of Brussels for the 1952-1953 year – and named them Belgian in her honour. But I have to admit that this is quite far-fetched. The truth undoubtedly is – because Angelini is Belgian.

41 Token Counting: APL versus J

First published in Vector, 22, 3, (August 2006), 49-54.

Revised by Don Guinn, (April 2009).

One measure of the difference between APL\360 and J is the number of tokens needed in a function to do the same work. I'll discuss two examples, comparing well-written, but rather old, APL solutions, to well-written J solutions.

Direct from Atomic

The first APL example is a marvellous function written by Luther Woodrum, that appears on page B.11 of the APL\360 User's Manual of 1968. Luther is best known to me by his design and implementation of the original upgrade and downgrade. His function `PERM`, below, is given a left argument `A`, the length of the permutation to be constructed, and a right argument `B`, the anagram index of the permutation to be constructed. It uses an algorithm first discussed, I believe, by D. H. Lehmer of the University of California in Berkeley. Here it is:*

```
▽ Z←A PERM B;I;Y
[1] I←ρZ←1+(ϕιA)τB-1
[2] →0×ι0=I+I-1
[3] Z[Y]←Z[Y]+Z[I]≤Z[Y←I+ιA-I]
[4] →2
▽
```

Disregarding the header, footer and line numbers, this has 55 tokens. What does it do? Let's suppose we want the 288918th permutation of order 9. Then (because APL begins counting at 1 in this case) we must compute:

```
(9 PERM 288918+1)-1
7 1 3 2 6 4 0 5 8
```

the same result as `9 sra 288918` which we develop below.

* In up-to-date APL, with control structures, this becomes (48 tokens):

```
▽ Z←A PERM B;I;Y
[1] :For I :In 1↓ϕιρZ←1+(ϕιA)τB-1
[2]     Z[Y]←Z[Y]+Z[I]≤Z[Y←I+ιA-I]
[3] :EndFor
▽
```

Here I should tell you that much of the material that follows is taken from my *At Play With J* article in *Vector 12.1* (July 1995).^{*} This is not surprising, since it deals with the subject of Luther's function.

The first step is to make a function that gives the factorial digits of permutations of length A. Luther's function uses origin 1, and that obfuscates things, so I'll use the more suitable 0-origin indigenous to J. To give you some idea of what the factorial digits number system is like, here are the six factorial digits in the system for three:

```
fdb=: >: @ i. @ -
      (fdb 3) #: i. ! 3
0 0 0
0 1 0
1 0 0
1 1 0
2 0 0
2 1 0
```

You can see the regularity in the rows. Notice also that every row ends in zero. This is true for all factorial digits systems.

In PERM, line 1 gives Z the factorial digits value for B.

```
NB. fdb n yields the radix digits of order 9
fdb 9
9 8 7 6 5 4 3 2 1
(fdb 9) #: 288918
7 1 2 1 3 1 0 0 0
```

Lines 2 and 4 control the executions of line 3, so that line 3 is executed only as long as I is positive. Line 3 can be defined as function g:

```
g=: [ , ] + ] <: [
NB. left , right + right >: left
```

Here I'll have to pause, and to point out that what I'm doing with g is taking the clutter out of PERM line 3. What we've done so far reduces line 3 from 26 tokens to 7, yet it does precisely what line 3 does. Perhaps if I show the successive uses of g you'll get the idea:

^{*} which is Chapter 7 in this book.


```

0
0 1
0 1 2
1 0 2 3
3 1 0 2 4
1 4 2 0 3 5
2 1 5 3 0 4 6
1 3 2 6 4 0 5 7
7 1 3 2 6 4 0 5 8
    
```

Successive lines are formed this way: for example, given line

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

the next line is formed by beginning with the corresponding factorial digit, in this case 2, and following with the previous line in which each item greater than or equal to 2 has 1 added to it.

```
2 1 5 3 0 4 6
```

Notice that each line is a permutation.

The function *g* is made of three forks:

```

g
+--+-----+
| [ | , | +--+-----+ | | | | | | |
| | | | ] | +--+-----+ |
| | | | | | ] | > : | [ | |
| | | | | | +--+-----+ |
| | | +--+-----+ |
+--+-----+
    
```

The three forks are:

```

fz=: ] >: [
fy=: ] + fz
fx=: [ , fy
    
```

Here's a *J* function, a functional duplicate of *PERM*:

```

sr=: /:@/:@,/          NB. standard form from reduced
ra=: ([: fdb [) #: ]   NB. reduced form from atomic
sra=: sr@ra f.          NB. standard from atomic
sra
/::@/:@,/@@([: >:@i.@- [) #: ])
;: 5!:5 sra            NB. tokens of sra
+--+-----+
| /: | @ /: | @ | , | / | @ | ( | ( | [ : | > : | @ | i . | @ | - | [ | ) | # : | ] | ) |
+--+-----+
# ;: 5!:5 <'sra'       NB. count of tokens in sra
20
    
```

The function `sr` uses an identity I found March 9, 1970, when I was looking at Luther's PERM once more:

```

N=: 7
P=: 1 3 2 6 4 0 5 7
(/:/:N,P) -: (N,P+N<:P)
1
NB. double upgrade matches addition, thus:
N,P+N<:P
7 1 3 2 6 4 0 5 8
/:/:N,P
7 1 3 2 6 4 0 5 8

```

So double-upgrade can take the place of Luther's line 3, and `sra` squeezes PERM down from 55 to 20 tokens.

Pyramigram

In *APL Quote Quad* 11.1, September, 1980, I asked for solutions to a problem posed by Linda Alvord, of Scotch Plains Fanwood High School in Scotch Plains, New Jersey. Here it is:

Write an APL function `PG` that takes a scalar integer argument from 1 to 26 and produces a rectangular character matrix containing a pattern like this:

```

PG 5
  Q
 W Q
Q E W
R Q W E
Q E R T W

```

In each row r there are r randomly selected and randomly ordered letters, separated by single spaces, arranged to form an equilateral triangle. The $(n-1)$ letters in row $n-1$ are selected from the n letters in row n .

This was one of the most popular problems I'd ever given, and there were a wide variety of solutions, including ones by some fairly gifted programmers, but one stood out from the rest, from Roger Hui. It took me several hundred words to describe what his function did. Roger recently told me that he had written his function without having access to an APL implementation. His function `PG` was not written in the conventional way that a function was defined in APL\360. Instead, it uses the alpha-omega form

introduced by Ken Iverson, in which the left and right arguments are denoted by α and ω . Here is a J function which uses the same algorithm as his from 1980:

```
PG=: ([:i.[:-])(|. "0 1)1j1"_ # "0 1(([:/:(([:-/ \]-
~[:|.[:i.[+:])#([:i.[+:])*[:+:])+[:?~[:+:]){[: ,
( , )}{. (('ABCDEFGHIJKLMN O P Q R S T U V W X Y Z' "_{~}?26"_ )
{.~[::-.[:+:])$~],[+:])$~],[
```

I won't dwell on this version other than to say that (1) it has 103 tokens and (2) it is in tacit form. The details are discussed in the cited issue of *APL Quote Quad*. It is a truth universally acknowledged that a good programming language, worked over and pondered over for a sufficiently long time by the same people who had produced the original, may very well show advantages over the original. Thus I sent an email to the J Forum list, and messages to key people in the APL community, for new solutions to the problem. I made it clear that the degree of improvement in expressiveness, as measured by token count, would be the criterion used. I received new solutions from the J and the APL communities. The shortest token count among the numerous APL solutions was 30, and there were several that used up to 60 tokens. The shortest J solution, by Roger Hui, was 20 tokens long, so I'll discuss that one only. This is it:

```
h=: /: # ? #
pyr=: i. &. - @ # |. " _1 [: 1j1 & # @ h \ h
```

His `h` has 4 tokens, and `pyr` has 16, totalling 20.*

The subfunction `h` is a hook. It randomizes its argument. Its first function is `/:` and its second function is the fork `# ? #`.

```
h 'qwert' NB. results may vary: h uses (?)
reqtw
```

* `pyr` itself contains 16 tokens because each instance of `h` is counted as only 1 token. But `pyr f.` (which expands both instances of `h`) contains 30 tokens, as shown thus:

```
pyf=: pyr f.
# ;: 5!:5 <'pyf'
```


which require a negative value. That's the reason for using *dual minus*. (&. -).

```

      i. _5
4 3 2 1 0
      (i. &. -) 5
_4 _3 _2 _1 0

```

Thus, the whole result is given by 20 tokens:

```

      (i. &. - @ # |. " _1 [: 1j1 & # @ h \ h)'qwert'
      e
      r e
      r e t
      e q r t
      r w q t e

```

From 103 tokens in 1980 to 20 in 2005, and by the same author, is a huge reduction.

Answers to Problems

These are not offered as the best answers, but the best known at the time of going to press (November, 2009). The reader is referred to an ongoing discussion of the problems on the *J Wiki*. See:
<http://www.jsoftware.com/jwiki/Doc/Articles/AnswersToAPWJ>

From page 38:

Solution to Problem 1:

```
magicperm=: C. @ < @ (|. @: >: , }.) @ (i.&.-:)
magicperm 2
0 1
magicperm 4
0 2 3 1
magicperm 6
0 2 4 1 5 3
magicperm 8
0 2 4 1 6 3 7 5
```

Solution to Problem 2:

Test each permutation individually:

```
seq =: 3 : 'y {^:():i.#y) i.#y'
pairs=: [: /:~ _2 /:~\ ,@seq
+/ (2 comb n)&-:@pairs@(A.&(i.n))"0 i.!n=: 2
2
+/ (2 comb n)&-:@pairs@(A.&(i.n))"0 i.!n=: 4
8
+/ (2 comb n)&-:@pairs@(A.&(i.n))"0 i.!n=: 6
48
...
```

From page 78:

Note: the solution shown for 91 is 4 tokens, the vector constant counting as 1 token.

```
0 /: 1996
1 * 1996
2 # 1 996
3 # 1 9 96
4 # 1 9 9 6
5 # #: 19.96
6 {: 199 6
7 <. ^ 1.996
8 >. ^ . 1996
9 1 9: 96
10 >. {: 19 9.6
11 # #: 1996
```

```

12 1 #. 9 9 _6
13 +/ 1 9 9 _6
14 <. %: 199.6
15 #. * 1 9 9 6
16 +/ 19 _9 6
17 p: { : 199 6
18 <: { . 19 96
19 { . 19 96
20 #. 1 9 _9 _6
21 >. >: 19.96
22 +/ 19 9 _6
23 +/ _1 9 9 6
24 1 #. 9 9 6
25 +/ 1 9 9 6
26 -: #. 19 _9 _6
27 <: -: #. 1 9 9 _6
28 <. #. 19 _9.6
29 p: 1 9: 96
30 #. >. 1 9.9 6
31 <: #. 1 9 _9 6
32 #. 1 9 _9 6
33 <. #. 19.9 _6
34 +/ 19 9 6
35 >: +/ 19 9 6
36 <. 1.9 ! 9.6
37 >. 1.9 ! 9.6
38 +: { . 19 96
39 <. +: 19.96
40 >. +: 19.96
41 -: #. 1 _9 96
42 >: -: #. 1 _9 96
43 <: <. %: 1996
44 <. %: 1996
45 >. %: 1996
46 >. #. 19.9 6
47 <. #. 19 9.6
48 -: { : 19 96
49 *: <. ^ 1.996
50 +: + / 1 9 9 6
51 <: #. 19 _9 _6
52 #. 19 _9 _6
53 #. <: 1 9 9 6
54 <. *: ^ 1.996
55 <: #. 1 9 9 _6
56 #. 1 9 9 _6
57 <: #. _19 96
58 #. _19 96
59 >: #. _19 96

```


60 >. {. o. 19 96
 61 p: p: {. 199 6
 62 <. o. 19.96
 63 >. o. 19.96
 64 #. 19 _9 6
 65 >: #. 19 _9 6
 66 >: >: #. 19 _9 6
 67 -: #. 19 96
 68 #. 1 9 9 6
 69 >: #. 1 9 9 6
 70 >: >: #. 1 9 9 6
 71 p: {. 19 96
 72 >: p: {. 19 96
 73 p: >. 19.96
 74 #. _1 _9 96
 75 1 9 #. 9 _6
 76 <: +/_ _19 96
 77 +/_ _19 96
 78 +/_ p: 1 9 9 6
 79 p: >. >: 19.96
 80 <: <: #. 1 _9 96
 81 #. -. 1 9 _96
 82 #. 1 _9 96
 83 #. >: 1 9 9 6
 84 >: #. >: 1 9 9 6
 85 <: +/_ _1 _9 96
 86 +/_ _1 _9 96
 87 1 9 #. 9 6
 88 +/_ 1 _9 96
 89 #. >: 1 _9 96
 90 >: #. >: 1 _9 96
 91 <: +/_ _1 99 _6 NB. 4 tokens!
 92 +/_ _1 99 _6
 93 1 #. 99 _6
 94 +/_ 1 99 _6
 95 <: {: 19 96
 96 {: 19 96
 97 >: {: 19 96
 98 #. <. 1.9 96
 99 <: #. 19 9 6
 100 #. 19 9 6

From page 177:

- A. The locker numbers are primes.
- B. The locker numbers are primes-squared.
- C. The locker numbers are products of two primes or a prime-cubed.
- D. Locker number 24.
- E. The students are 1, 2, 3, 4, 6, 12: all factors of $12 = \text{HCF}(24, 36)$.
- F. The students are 1, 2, 4, 5, 10, 20: all factors of $20 = \text{HCF}(100, 120)$.

From page 329:

The sixth number is 5596446229. *Why?*

We note the six numbers are divisible by 11, which is the only common factor. This lets us guess the defining property to be: (sum of even digits) = (sum of odd digits).

Method: compute **pa** as in the chapter. Then convert **pa** back to digits form, like **t**. We cannot use **t** directly because it includes several terms with leading zeros, which **pa** doesn't include.

```
pd=: (10#10) #: pa NB. pa in digits form
even=: -. odd=: 1 0 1 0 1 0 1 0 1 0
NB. 'alternately, something for nothing'
,. x: 10 #. ((even&# =&(+/) odd&#)"1 # ]) pd
4338327950
2795028841
6939937510
3993751058
2110555964
5596446229
```

Thanks to Ewart Shaw for this solution.

References

Abramowitz, M., Stegun, I. A., (eds.), *Handbook of mathematical functions*. US National Bureau of Standards, Applied Mathematics Series – 55, (1964–1972). Also: Dover Publications (June, 1965), ISBN 978-0486612720.

See also:

<http://www.math.sfu.ca/~cbm/aands/>

<http://dlmf.nist.gov/>

Backus, J., Can programming be liberated from the von Neumann style? A Functional Style and its Algebra of Programs. *Comm. ACM* 21, 8, (1978-08).

Bernecky, R., Hui, R. K. W., Gerunds and Representations. *APL Quote Quad* 21, 4, Stanford, Calif., (1991-08), 39-45.

Brenner, C. H., Brenner, J. L., The popularity of small integers as primitive roots. *Numer. Math.* 4, (1962), 336-342.

Brenner, J. L., Beasley, L. P., Erdős, P., Szalay, M., Williamson, A. G., Generation of alternating groups by pairs of conjugates. *Period. Math. Hungar.* 18, (1987), 259-269.

Conway, J. H., Guy, R., *The Book of Numbers*. Springer (Feb 1998), ISBN 978 0 3879 7993 9.

Erdős, P., Linial, N., Moran, S., Extremal Problems on permutations under cyclic equivalence. *Discrete Math.* 64, (1987), 1-11.

Erdős, P., Shallit, J., New bounds on the length of finite Pierce and Engel series. *Séminaire de Théorie des Nombres de Bordeaux* 3, (1991), 43-53.

FinnAPL Idiom Library.

<http://www.pyr.fi/apl/texts/Idiot.htm>

Golub, G. H., Van Loan, C. F., *Matrix Computations*. Johns Hopkins Studies in Mathematical Sciences, 3rd Edition, (October 1996), Johns Hopkins University Press. ISBN 978-0-80185-414-9.

Hui, R. K. W., *Incunabulum*.

<http://www.jsoftware.com/jwiki/Essays/Incunabulum>

Hui, R. K. W., The Ball Clock Problem. *Vector* 12, 2, (1995), 55-66.

Hui, R. K. W., Iverson, K. E., McDonnell, E. E., Whitney, A. T., “APL\”. *APL90 Conf. Proc.*, 192-200.

Ibarra, O., Moran, S., Hui, R. K. W., A Generalization of the Fast LUP Matrix Decomposition Algorithm and Applications. *Journal of Algorithms* 3, (1982), 45-56.

Illing, Robert, *A Dictionary of Music*. Penguin Books, (1950), 297 (fig. f).

Iverson, K. E., *A Programming Language*. Wiley, (December 1962).
ISBN 0471430145.

Iverson, K. E., Operators and functions. RC 7091, IBM Corp., Yorktown Heights, NY, (1978).

Knuth, D., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Third Edition, Reading, Massachusetts: Addison-Wesley, (1997), ISBN 0-201-89684-2.

Knuth, D., (home page):
<http://www-cs-faculty.stanford.edu/~knuth/news.html>

Knuth, D., *The Stanford GraphBase: A Platform for Combinatorial Computing*, Reading, Massachusetts: Addison-Wesley, (1993).
ISBN 0-201-54275-7.

Lagaria, Miller, Odlyzko, Computing $\pi(x)$: Meissel-Lehmer Method. *Mathematics of Computation* 44, 170, (1985-04), 537-560.

Lanczos, Cornelius, *Applied Analysis*, Prentice-Hall, (1956).

McDonnell, E. E., Blists in OLEIS. *Vector* 17, 1, (2000-07), 110-120.

McDonnell, E. E., Complex Floor. *APL Congress 73*, Copenhagen.

McDonnell, E. E., How Shall I Transpose Thee? Let Me Count The Ways. *APL Quote Quad*, 8, 1, (1977-09).

McDonnell, E. E., Spirals & Time. *APL Quote Quad* 7, 4, (Winter 1977), 20-22.

McDonnell, E. E., *The Four Cube Problem*. APL Press, Palo Alto, (1981).

McDonnell, E. E., The Magical Matrix. *Vector* 20, 2, (2003-08), 122-126.

McDonnell, E. E., Shallit, J., Extending APL to Infinity. *APL80*, Noordwijkerhout, 123-132.

Nijenhuis, A., Wilf, H. S., *Combinatorial Algorithms*. Academic Press, New York, (1978).

Pohlig, S. C., Hellman, M. E., An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Trans. Info. Theory* IT-24 (1978) 106-110.

Reiter, C., *Fractals, Visualization and J*. Lulu.com, Third Edition, (2007), ISBN 978-1-4303-1980-1.
<http://www.lulu.com/content/635966>
<http://www.lulu.com/content/635938> (coil binding.)

Schechter, Bruce, *My Brain Is Open, The Mathematical Journeys Of Paul Erdős*. Simon & Schuster, New York, (1998).

Shallit, J. O., Merrily We Roll Along: some aspects of ?. *APL83 Conf. Proc.*, reprinted in: *APL Quote-Quad* 13, 3, (1983-03), 243-248.

Shallit, J. O., What this country needs is an 18¢ piece. *Math. Intelligencer* 25, 2, (2003), 20-23. Also available at Shallit's website:

<http://www.math.uwaterloo.ca/~shallit/papers.html>.

This gives pointers to the paper in two forms: PostScript and PDF.

Shaw, E., Hypergeometric Functions and CDFs in J. *Vector* 18, 4, (April 2002), 139-143.

Sloane, N. J. A., *On-Line Encyclopedia of Integer Sequences*.

<http://www.research.att.com/~njas/sequences/>

Smillie, K., Primes, Spirals and Coffee Tables. *Vector*, 11, 4, (1995), 104-107.

