January 2000

Mostly J

Keith Smillie

*Some reflections on forty-five years of reading and programming and teaching programming languages with an emphasis on the array language **J**.*

An earlier version of this paper was published as *Some Thoughts on **J** and Other Langauges*.

Part One. Prologue

Introduction

Having spent most of my professional life both writing programs in a variety of programming languages and also teaching programming, I can easily say *why* I have done so. There are several reasons: I have enjoyed writing and documenting programs; I have enjoyed teaching programming; I have liked the people I have met, including my students when I was teaching; and I have obtained satisfaction from knowing that occasionally others have found my work interesting and even useful. I continue to work after retirement for the same reasons. In brief, I have always enjoyed my work and I continue to enjoy it after retirement.

However, the question of *what* I have done is much more difficult to answer. The programming of computers is to many persons a mysterious activity to be understood only by the specialist. The mystery only deepens when a description of the problem being programmed has some mathematical content or involves the use of even the most elementary of mathematical notation or arithmetical operations. This situation is indeed unfortunate because much of this work is in principle rather simple and within the grasp of any literate person.

Much of the blame for this lack of understanding must rest with us, the current practitioners of the computational and mathematical sciences, who often write in a language which is comprehensible only to fellow practitioners, and at times not even to them. It is really a shame that we have not tried more diligently to write for a larger audience so that we may provide some understanding of what we do and why we have found our work so satisfying. Also I believe we have an obligation to explain to people how we spend our professional lives. We owe this to our family and friends, whom we on occasion may neglect when we become preoccupied with our work. We also have an obligation to others who support us directly and indirectly and who possibly often wonder what they are getting in return.

This retrospective and idiosyncratic paper is my attempt to describe in simple language some of the things I have been doing with computers for so many years. As most of the professional problems with which I have been concerned have been mathematical or statistical in nature, I have had to introduce a few technical terms and some notation now and then in the examples. I hope, though, that the notation does not intrude unduly on the ideas being presented. However, since I believe that the language we use influences how we think, I have included some sections near the end of the paper which give some further remarks on the notation which is used in the examples. They may be read almost independently of most of the preceding sections, or they may be omitted altogether.

This retrospective and idiosyncratic paper is my attempt to describe in simple language some of the things I have been doing with computers for so many years.

We shall begin in Part Two with a look at some of the programming languages I have used and make the distinction between what I term “conventional” and “array” languages. We then introduce the three array languages APL, Nial and **J** which I have worked with in turn for so many years, and give a brief discussion of **J** illustrated with a few simple examples of its use. Parts Three, Four and Five give a discussion of a variety of problems I have found instructive, useful and entertaining and show how **J** may be used in their solution. In Part Six we return to a discussion of **J** and try to gather together and unify those parts of the language we have used previously. Then in Part Seven we return to the discussion of languages and compare the teaching of a programming language with that of a natural language especially Japanese. Finally, in Part Eight I discuss some of the technical books I have enjoyed reading, and conclude with a few general remarks and acknowledgements.

What I attempting here - a discussion of a scientific or technological subject for the general reader - is not only not easy but also unfortunately not entirely fashionable in academic circles where the emphasis is on the acquisition of new knowledge and its publication in professional journals read only by other specialists. An excellent statement of the opinion on popular writing of many specialists is

given by the Canadian-born economist John Kenneth Galbraith who summed up with his usual acerbic wit the view of many academics as follows:

> There are a significant number of learned men and women who hold that any successful effort to make ideas lively, intelligible and interesting is a manifestation of deficient scholarship. This is the fortress behind which the minimally coherent regularly find refuge.

In attempting to explain what we do to the general public we find ourselves in distinguished company. One of my favourite examples is the noted English astronomer Sir James Jeans who from the age of about fifty devoted himself exclusively to popular writing. One of his best-known books, *The Growth of Physical Science*, was published posthumously in 1947, the year after his death. Another example is Sir Thomas Huxley's *On a Piece of Chalk*, a lecture on geological evolution delivered to the working men of Norwich at a meeting of the British Association for the Advancement of Science and republished in 1968 on the centenary of its first publication. A third example is *Science and Humanism*, given first as a series of public lectures by the physicist and Nobel laureate Erwin Schrödinger. In introducing his subject he makes the statement "If you cannot - in the long run - tell everyone what you have been doing, your doing has been worthless."

The importance of having a healthy balance between broad teaching and specialized research has not been completely ignored at the University of Alberta in spite of the present emphasis here and indeed throughout much of academia on research and technological training. One example must suffice. In 1964 at the opening of Red Deer Junior College Walter Johns, then President of the University, made the following remarks:

> The universities today are havens for free inquiry - and so they should remain - but they should also be centres of teaching as well as of learning. There is a way of academic life epitomized in the phrase "publish or perish" for the academic world which requires that each person in the modern college must perforce add his own share of the contributions to the mountains of information that already reach the height of Mount Everest and are growing larger every minute. Is there to be no place for the scholar or the scientist who might wish to study this vast pile of ore to find the precious metal in it? Is there to be no place for the person who considers it is his task to pass on to the students in his classroom the results of this enormous activity for their use and comfort?

Some suggestions

The reader wishing only an introduction to programming languages, the difference between conventional programming languages and array languages such as **J** and a brief account of **J** with a few very simple examples might read Part Two which follows immediately and then skip to Part Seven for some comparison of the teaching of programming languages and natural languages. The intervening sections, which as noted in the previous section, develop the use of **J** in a variety of applications, might well be omitted, at least on a first reading. This brief introduction might be concluded with the first section of Part Eight where I discuss a few of the books I have enjoyed reading.

All of Part Three is devoted to the coupon collector's problem which provides a model for collecting prizes which are contained, one prize per package, in some product. This problem is intended to illustrate how a programming language such as **J** may be used in an arithmetical and statistical investigation of a problem. Part Four gives a discussion of numbers - small numbers, big numbers, really gigantic numbers, prime numbers, etc. - as objects worthy of study for their intrinsic value and interest rather than for their usefulness in expressing, say, prices or bank balances. In Part Five I have gathered together a motley collection of examples which I have found interesting and challenging over many years: the summing of observations arranged in orderly arrays which is fundamental to a broad class of statistical analyses; some aspects of cloth weaving intended to show how a problem which may not appear to be at all mathematical may benefit from a numerical approach; a number of examples related to games of chance; and a somewhat recreational example giving what many persons may find to be a surprising, and economical, alternative to the purchase of Christmas

cards.

Many sections of this paper conclude with supplementary material separated from the preceding material in the section by a horizontal line. This material, which consists often of additional remarks about **J**, may be - or possibly should be - omitted on a first reading.

For readers who have access to **J** on a computer and who may wish to have a further look at the examples, all of the **J** programs and data are available by anonymous ftp at

ftp.cs.ualberta.ca .

The script file for all of the **J** examples given here except for the weaving example is in the file

pub/smillie/mostlyj.ijs ;

the script file for the weaving example is in

pub/smillie/loom.ijs .

Part Two. From Machine Language to J

Programming langauges

One of the definitions of "program" in the *American Heritage Dictionary* is "To provide (a computer) with a set of instructions for solving a problem." The word comes from the Greek and means simply "to write before". In view of this definition, I have found it helpful to consider that the use of any software which gives one control over a computer to constitute programming, whether it involves conventional languages, array languages, spreadsheets, software packages designed for specific applications, or even word processors. In this section I wish to make a few remarks about some of the programming languages I have used.

The first computer I worked with was the National Cash Register 102A which, as were all computers in the 1950s, programmed in machine-language. A program consisted of a sequence of instructions written in numerical form (in octal or base-8 notation, not decimal), each specifying the operation and the addresses (locations) in memory of the numbers to be operated on and the address of the result. Data entered into the computer had to be converted to binary, the arithmetic performed in binary, and the final results converted to decimal before being printed. Moreover, the programmer had to explicitly keep track of the size of all numbers throughout a computation to ensure that they, or at least their binary equivalents, always remained less than unity in absolute value. Thus programs for what we would consider today very simple or even trivial calculations could be quite lengthy. For example, one program for calculating the arithmetic mean of a list of numbers required about four dozen instructions.

To simplify the programming task, most computers were soon provided with programs which allowed the programmer to use a simpler language similar to machine-language which would be interpreted one command at a time as sequences of machine-language instructions. An example was Simple Code for the Stantec Zebra, a computer installed at the Suffield Experimental Station in Alberta and which I used for the Canada Department of Agriculture in Lethbridge, Alberta. Programming in Simple Code was still a most demanding task but it was very much easier than programming the Stantec Zebra in machine-language with its repertoire of, theoretically at least, several million different instructions.

A breakthrough in programming came in the late 1950s with the development by the International Business Machines Corporation of FORTRAN, for Formula Translating System, which allowed the programmer to write programs in an algebraic-like language. The program would be first translated in its entirety to machine-language and then executed. Since its release in 1957 FORTRAN has been almost continuously developed - the latest version is FORTRAN90 - and has had a profound effect on the the development of programming languages and their teaching.

I can still remember attending a lecture on FORTRAN in the late 1950s and realizing that at last I would be freed from the drudgery of machine-language programming. I hope, though, I was not sufficiently naive to think that FORTRAN would remove all of the drudgery from programming! The appearance of FORTRAN made the use of computers feasible for people who did not wish to become full-time programmers at the expense of their chosen professions. Courses in FORTRAN were soon established at universities and colleges for students in science and engineering. In the early

1970s at the University of Alberta FORTRAN was replaced briefly with ALGOLW and then with Pascal as the first language for computing science students. Presently Pascal is being replaced with Java in introductory courses. Although there are important differences between these languages, they are sufficiently similar that students experience the same problems when learning to use them.

BASIC, Beginner's All-Purpose Symbolic Instruction Code, was developed at Dartmouth College, a small liberal arts college in New Hampshire, as a simple alternative to FORTRAN for undergraduate students, most of whom were in the social sciences and the humanities. The first BASIC program was run on May 1, 1964 at four o'clock in the morning. It was equivalent to the evaluation of the arithmetic expression $(7 + 8) \div 3$. The language was an immediate success and has become probably the most popular and widely used language in the world. It received only limited use at the University of Alberta until the mid-1980s when it became the first language for students in Arts and Education. Again, in spite of important differences with the languages already mentioned, students encountered the same problems in learning it as with the other languages.

The first BASIC program was run on May 1, 1964 at four o'clock in the morning. It was equivalent to the evaluation of the arithmetic expression (7 + 8) ÷ 3.

Most of my early work was concerned with writing, documenting and maintaining programs for statistical calculations. For the Canada Department of Agriculture I worked with both the Stantec Zebra Simple Code in Suffield and Lethbridge and FORTRAN in Ottawa. Until the Department of Agriculture obtained its own computer we used either the computer at the University of Ottawa located in a very old "temporary" building or at the local IBM office where the computer was located in the front window of what was then a modern office building on Laurier Avenue. I continued this work of program development and assistance with statistical calculations for a few years at the University of Alberta until the work was taken over by the Department of Computing Services. It is interesting to realize that almost all of these calculations would be done now, and probably more comprehensively and more conveniently, with either spreadsheet or statistical packages.

I believe that FORTRAN, ALGOLW, Pascal and BASIC - and indeed many other languages - are sufficiently similar to be grouped in a class which I call conventional languages. Although I have used and taught all of these languages, and have derived great pleasure in doing so, my main professional interest has been with what I term the array languages APL, Nial and **J**. In the next section we will give a simple analogy to illustrate what I think is the basic difference between these two classes of language.

I have mentioned only a half-dozen or so of the many hundreds of programming languages and dialects that have appeared. What is now their early history has been chronicled by Jean Sammett in *Programming Languages: History and Fundamentals* (Prentice-Hall, Inc., 1969). One of the figures in this book depicted programming languages as a huge "Tower of Babel" with the languages given on a spiral staircase leading to the (imaginary) language "BABEL" at the top. How much larger would a present-day Tower of Babel be for programming languages!

Counting apples

An excellent example of the difference between conventional languages and array languages has been given by Frederick Brooks of the University of North Carolina. In his example, which we have modified somewhat, we are to give a list of instructions for counting the number of good apples in a box of apples. We shall give the instructions first in the style they would be given in a conventional language and then in the style of an array language.

The first instruction in the conventional programming style would be to get a blank piece of paper for keeping a tally of the good apples as we come across them in the box. The next instruction would be to pick an apple from the box, and examine it for goodness according to some criterion.

If it is a good apple, then put a mark on the tally paper. Then pick a second apple, examine it for goodness, and increase the tally if it is a good apple. Then pick another apple, ... Continue this procedure of examining the apples one by one until all of the apples in the box have been examined and the occurrence of the good apples recorded. The number of good apples will be given by the number showing on the tally sheet.

On the other hand, the instructions given in the style of an array language would consist of a simple statement such as "Mark all of the good apples in the box and then add up the marks." It would be assumed that the person to whom the instructions were given would be able to work out the details of marking the good apples one at a time and then determining how many apples had been marked.

An array language is an all-the-apples-at-once language in which most of the details are taken care of by the language itself and are not visible to the programmer.

A conventional programming language, then, is an apple-at-a-time language in which all of the details must be carefully specified in the program and tested to ensure that the program does what it is designed to do. An array language is an all-the-apples-at-once language in which most of the details are taken care of by the language itself and are not visible to the programmer. Of course, a program written in any language must be carefully tested but the testing is usually much simpler with array languages.

Now we are ready to consider very briefly the origins of the three array languages with which I have been concerned for so many years. In the next section we shall give a very quick sketch of the origins of these languages and in the section following that one we shall begin our look at **J**.

APL, Nial and J

APL, for A Programming Language, was originally conceived by Kenneth Iverson while a graduate student at Harvard in the early 1950s. It was intended as an alternative to conventional mathematical notation for the description of algorithms arising in problems of sorting, searching and optimization. After leaving Harvard, Iverson joined IBM where APL was first implemented on a computer in 1966. Since then there have been several major implementations, the current one being designated APL2.

The principles underlying the design of APL have been simplicity, brevity and generality. While the conventions of mathematical notation have been respected, these principles have always been given precedence. The data objects in APL are one-dimensional lists, two-dimensional tables, and in general rectangular arrays of arbitrary dimension. In addition to the usual elementary arithmetical operations of addition, subtraction, multiplication, division and raising to a power, there is a large number of additional operations which are defined for arrays as well as for individual numbers. For example, if we have a list of unit prices and a list of quantities of each item purchased, then a single multiplication will give a list of the total amount spent for each item, a single operation on this list will give the total amount spent, and if there is a sales tax one more multiplication will give the total cost including tax.

The principles underlying the design of APL have been simplicity, brevity and generality. While the conventions of mathematical notation have been respected, these principles have always been given precedence.

Nial, Nested Interactive Array Language, combines concepts from APL and other languages within the framework of a mathematical model called array theory. It was developed over many years by Trenchard More of the IBM Cambridge Scientific Center. The name comes from the Old Norse Icelandic name *Njal*. The data objects in Nial are arrays whose items may be arrays, whose items in turn may be arrays, and so on. The basic items of arrays may be any of several types such as integers, decimal numbers, literal characters, etc., and different types may occur in the same array. As with

APL, there is a large number of primitive array operations available and additional operators and programs may be defined.

Upon retirement about ten years ago Kenneth Iverson began work on a "modern dialect" of APL that would provide the simplicity and generality of APL while at the same time be readily and inexpensively available on a variety of computers and capable of being printed on standard printers. The language was given the name **J** by Roger Hui - who along with Ken, his son Eric, and Chris Burke have been its principal developers - because, he said, "the letter 'J' is easy to type". (We might note that the name **J** precedes the naming of Microsoft's Java development tool as Visual J++.) As **J** will be used in the examples in this paper, we give no further remarks on the nature of the language in this section.

There appear to be no data readily available on either the use of array languages relative to other programming languages or the relative use of APL, Nial and **J**. APL has been used extensively in a wide range of scientific and commercial applications and has had a group of devoted advocates in many universities. It continues to be used extensively and is supported by IBM and other major companies. Nial had a small but enthusiastic group of supporters, mostly academics, in the 1980s, but appears to be little used now. Interest in **J** has grown very much in the last few years, and should continue to grow as the language develops.

As an interesting footnote to this section we shall make a few remarks about the on-line use of APL in the classroom. In the late 1960s we decided to experiment with the use of a live computer terminal in one of our introductory programming courses. With very little effort we were able to get a connection in one of the classrooms with the IBM 360/67 in the Computing Centre and by means of a television camera display APL calculations as they were being done on two or three television screens. This arrangement continued to be used until it was replaced by the newly announced IBM 5100 minicomputer in 1975.

The IBM 5100 was a small computer by the standards of the day, measuring 17.5 inches by 24 inches by 8 inches and weighing 48 pounds. It had an attached keyboard, a very small screen, a tape unit for permanent storage, a direct connection to a televison monitor, and supported both APL and BASIC. The system fitted conveniently on the top shelf of an audio-visual trolley with the printer on the bottom shelf. This system was used effectively for several years both in the classroom for teaching and in faculty offices for course preparation and research. The audio-visual trolley remained in my office for many years, its only use being to hold one of my copies of the *American Heritage Dictionary*. Now even the trolley is gone.

It might be remarked that all of the arrangements for these very early experiments with computers in the classroom were made informally without recourse to any committee work. I can remember being chastised for having a hole bored through the classroom floor for the IBM 360/67 connection without obtaining proper authorization. My excuse that it was "only a little hole" was not too kindly received. Also I was told I had been "very naughty" for arranging for the purchase of the IBM 5100 without authorization from the appropriate committee.

I can remember being chastised for having a hole bored through the classroom floor for the IBM 360/67 connection without obtaining proper authorization. My excuse that it was "only a little hole" was not too kindly received.

Rainfall in Edinburgh

As a first simple example of a **J** program we shall use the following data on the monthly rainfall in millimetres for Edinburgh, Scotland in 1978 together with the thirty-year averages for the years 1941 to 1970 taken from *Understanding Data* by Peter Sprent (Penguin Books, 1988) :

Month	*1978*	*Ave.*
Jan.	58	54
Feb.	75	41
Mar.	53	36
Apr.	49	38
May	24	58
June	49	47

July	69	75
Aug.	72	86
Sep.	63	63
Oct.	13	57
Nov.	50	64
Dec.	99	54

We wish to count the number of months in which the rainfall was greater than the long-term average for the month.

The following is a BASIC program for this calculation:

```
REM Rainfall in Edinburgh
DATA 12
DATA 58,54,75,41,53,36,49,38
DATA 24,58,49,47,69,75,72,86
DATA 63,63,13,57,50,64,99,54
Tally = 0
FOR Month = 1 to 12
  READ R, A
  IF (R > A) THEN Tally = Tally + 1
NEXT Month
PRINT Tally
STOP
END
```

A person unfamiliar with BASIC should be able to follow the steps in the program and see how they are similar to those in the apple-at-a-time procedure described above.

If we use **J** to perform the calculations, we first define the lists

```
   R=: 58 75 53 49 24 49 69 72 63 13 50 99
```

and

```
   A=: 54 41 36 38 58 47 75 86 63 57 64 54
```

to give the current and average rainfalls, respectively, for the months. (We could have used longer and more descriptive names such as `Rainfall` and `Average` if we had chosen to.) The expression

```
   R > A
```

compares each current monthly rainfall with the monthly average and gives the result

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

which is a list of twelve items corresponding to the twelve months. The `1`s correspond to those months where the rainfall is greater than average and the `0`s to those months where it is not. The number of months with a greater than average rainfall is simply the total number of `1`s in this list which is the sum of the items in the list. In **J** this is given by

```
   +/1 1 1 1 0 1 0 0 0 0 0 1
```

which is `6`. These calculations may be done in the single expression

```
   +/R > A   .
```

Thus the entire **J** program is given by the following three statements:

```
R=: 58 75 53 49 24 49 69 72 63 13 50 99
A=: 54 41 36 38 58 47 75 86 63 57 64 54
+/R > A
```

Its simplicity as compared with the BASIC program is obvious.

It is interesting to note that the total rainfall in 1978 is `+/R` which is `674` millimetres and the average rainfall is `+/A` which is `673` millimetres.

A little more J

In this section we shall give a few more examples of **J** by means of a short dialogue with the computer. The **J** expressions entered by the user are indented automatically three spaces, and the responses by the computer begin at the left margin. The comments which follow the expressions and which begin with `NB.` are for the reader and are ignored during evaluation. The calculation for the average, or arithmetic mean, is done by the program `am`, the details of which are given in the next section.

```
   3 + 5              NB. Plus
8
   2 * 3              NB. Times
6
   2 + 3 * 4
14
   3 - 5              NB. Minus
_2
   15 % 6             NB. Divided by
2.5
   % 8                NB. Reciprocal
0.125
   %: 6               NB. Square root
2.44949
   3^5                NB. Power
243
   3 * 3 * 3 * 3 * 3
243
   2.3 + 5 + 3.5 + 6  NB. Sum
16.8
   +/ 2.3 5 3.5 6
16.8
```

```
   am 2.3 5 3.5 6      NB. Average
4.2
   w=: 2.3 5 3.5 6
   +/w
16.8
   am w
4.2
   Price=: 1.39 0.85 2.90 1.49
   Qty=: 1 3 1 2
   Price * Qty
1.39 2.55 2.9 2.98
   +/ Price * Qty      NB. Total
9.82
```

Parts of speech

In **J** the terminology of English grammar is used rather than that of programming languages. Functions such as + for *plus* and > for *larger than* are referred to as verbs. Their arguments, i.e., the numbers to which they apply, are called nouns and pronouns instead of constants and variables. Furthermore, verbs may be modified by adverbs to give related verbs, and they may be combined by conjunctions to give more verbs. In this paper the main use of this terminolgy we shall make is that we shall use the term "verb" rather than the more customary terms "function", "procedure" and "program" used in most programming languages. In the remainder of this section we shall give a few more details for the interested reader about the **J** language.

The verb plus `+` which gives the sum of its arguments - as in `4 + 6` is `10` - may be modified by the adverb `/` to give the verb `+/` which will give the sum of a list of numbers. For example,

```
   +/ 4 6 8 5 2
```

is equal to `25`, the sum of the numbers in the list. Since this sum may be written

```
   4 + 6 + 8 + 5 + 2 ,
```

where the verb `+` is inserted between the numbers in the list, the adverb `/` has the name *insert*. We have seen the verb `+/` in an earlier section where it was used to find the sums `+/R` and `+/A` of the 1978 and average rainfalls, respectively.

The verb `+/` may be given an arbitrary name such as `sum` by the statement

```
   sum=: +/
```

so that it may be referred to conveniently. Thus

```
   sum 4 6 8 5 2
```

is `25` and `sum R` is `674`.

The rainfall calculations consisted of first comparing each monthly rainfall with the average rainfall for that month, which gave a list of `1`s and `0`s, and then summing the items in this list to give the total number of `1`s. The two parts of the calculation, represented by the verbs `>` and `+/`, may be combined to give a verb for performing the entire calculation and this verb may given a name. Thus we may define the verb

```
   NumGT=: +/ @: > ,
```

where `@:` is the conjunction *at* which is used for combining verbs, and the calculations may be performed simply as

```
   R NumGT A
```

which is equal to `6`. The expression

```
   A NumGT R
```

is equal to `5` because here we are finding the number of months in which the average rainfall is greater than the 1978 rainfall for that month. We note that in September the rainfall was equal to the average rainfall.

We remarked earlier that a machine-language program for the arithmetic mean required about four dozen instructions. It is interesting to note how simply the calculations may be performed in **J**. If

```
   w=:  2.3 5 3.5 6
```

is a list of observations, then the arithmetic mean is given by the expression

```
   +/w % #w
```

which is equal to `4.2`. The verb `%` is *divided by*, and `#` is *tally* which gives the number of items in its argument. Thus the expression follows directly from the definition of the arithmetic mean as "the sum of the observations divided by the number of observations". The calculations may be performed even more simply if we define the verb

```
   am=: +/ % # ,
```

and we have that `am w` is `4.2`.

A sequence of three verbs such as occurs in the definition of the verb `am` is known as a *fork*. It is used in conventional mathematical notation, where, for example, $(f_1+f_2)x$ represents the sum $f_1(x)+f_2(x)$. Another example of a fork is the geometric mean of a list of n observations defined as the nth root of the product of the observations. We may define the verb

```
   gm=.  # %: */ ,
```

where `%:` is the verb *root*, for the geometric mean, and the expression `gm w` is equal to `3.94211`. In general, if `f`, `g` and `h` are verbs and `y` is a noun, then the fork

```
(f g h)y
```

is equal to

```
(f y) g (h y) .
```

We shall conclude this section with definitions of the arithmetic mean in APL and Nial. One definition of an APL function for the arithmetic mean is

AVE: (+/⍵) ÷ ⍴⍵

and is almost identical to the **J** verb given above. A corresponding operator in Nial is

am IS div[sum,tally]

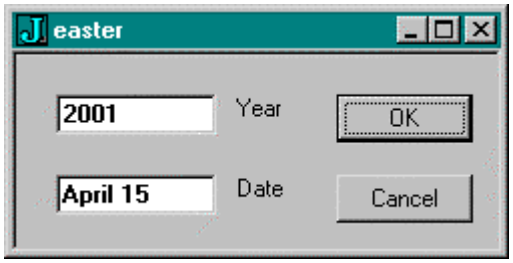
which although very similar in interpretation is very different in form. Whereas the primitives in APL and **J** are represented by symbols, although synonyms with names may be easily defined in either langauge, all operators in Nial are represented by names with only a few having equivalent symbols.

Windows

In the rainfall example we gave the verb `NumGT` so that the calculations could be done very simply by means of the expression `R NumGT A`. Even then, we have to remember that it is this expression and not `A NumGT R` that is used. Also there is no reason why the verb `NumGT` could not have been designed so that the correct expression would be `NumGT A;R`, where `A;R` is a two-item list with the list `A` as the first item and the list `R` as the second. Similar remarks may be made about any **J** verb and about programs in any language. The fewer details that have to be remembered, the easier their use will be and the less will be the likelihood of error.

To simplify the use of verbs in **J** we may construct a Windows form with menus, push buttons, input boxes, etc. so that the user need have no knowledge of the details of the computations being performed. Assembling a Windows form in **J** is somewhat similar to building with LEGO blocks. It isn't quite as much fun, but at least one doesn't have to get down on one's hands and knees to play. The Windows Driver is used to create a blank form. Then the required number of Windows controls - push buttons, check boxes, radio buttons, list and edit boxes of various types, etc. - are placed on the form, and arranged in a pleasing and functional layout. While this is being done the Windows Driver constructs a skeletal **J** program which allows the controls to carry out their assigned tasks. When the layout is complete, the user edits the **J** program, and incorporates whatever **J** verbs are required for the task at hand. The completed package consisting of the Windows interface, control program and specialized **J** verbs may then be used.

Assembling a Windows form in J is somewhat similar to building with LEGO blocks. It isn't quite as much fun, but at least one doesn't have to get down on one's hands and knees to play.



As a simple example, the Windows form given here allows one to find the date of Easter for any given year. Of course, a simpler way to find the date of Easter is to consult a published table of Easter dates. One such table is given in *The Oxford Companion to English Literature*. It begins in the year 1066 and ends, at least in the Fifth Edition, in the year 2000. Determination of the date for subsequent years must be done by hand unless a more recent table is found. Such calculations are straightforward but rather tedious and prone to error. The calculations used in this form are based on an algorithm given in *Calendar. Humanity's Epic Struggle to Determine a True and Accurate Year* by David Ewing Duncan (Avon Books, 1998).

A short summary of J

It may be helpful to gather together in an orderly way those parts of the **J** language which we have discussed so far and to make a few additional

comments at the same time. This will give the reader a review of what has been accomplished and possibly be of assistance in the further use of **J** in what follows. Here, then, very briefly are the main aspects of the **J** language:

- The standard ASCII character set is used.
- The terminology of English grammar is used rather than that of programming languages. Functions are referred to as *verbs*. Their arguments are called *nouns* and *pronouns* instead of constants and variables, although we prefer the use of these latter terms in this paper. Verbs may be modified by *adverbs* and joined by *conjunctions* to give additional verbs. For example, we have used the verb `+/` derived from the verb `+` *plus* by use of the adverb `/` *insert* to give the sum of the items of a list, and the conjunction *at*, represented by `@:` in the defined verb `NumGT` to combine the two constituent verbs `+/` and `>`.
- Primitives, i.e., verbs, adverbs and conjunctions, are represented by a single character or a single character followed by either a period or a colon. For example, `>` is the verb *larger than*, and `6 > 3.5` is `1` and `2 > 7` is `0` indicating that the first relationship is true and the second false. The verb `>.` is *larger of* and gives the larger of its two arguments so that `6 >. 5` is `6`, while `>:` is *larger or equal* and `6 >: 5` is `1` as is `6 >: 6` but `2 >: 7` is `0`.
- Most verb symbols represent one function when used with one argument on the right and another function when used with arguments on the right and left. We have already seen the verbs *reciprocal* and *divided by*, each represented by the symbol `%`, so that `% 8` is `0.125` and `15 % 6` is `2.5`. Both forms may be used in the same expression so that `% 15 % 6` is `0.4` which is "the reciprocal of `15` divided by `6`". As we have just seen the verb `>:` with two arguments represents *larger or equal*, but with a single argument it represents *increment* so that `>: 3.5` is `4.5`.
- Precedence amongst verbs is determined by parentheses, and in their absence the right argument is the entire expression on the right and the left argument is the noun immediately on the left. For example, the expression `% 15 % 6` in the previous paragraph is "the reciprocal of (`15` divided by `6`)" rather than "(the reciprocal of `15`) divided by `6`". Likewise, `2 + 3 * 4` is `14` as `(3 * 4) + 2` but `3 * 4 + 2` is `18`.
- Negative numbers are indicated by a preceding underbar `_` which is considered to be part of the number as is, for example, the decimal point. Also the decimal point is necessarily preceded by at least one digit so that, for example, two-fifths as a decimal fraction is represented as `0.4`.
- Nouns may be single items or *atoms*, one-dimensional arrays or *lists*, two-dimensional arrays or *tables*, or arrays of higher dimension or *reports*. Thus the expression `a + b` is a valid sum as long as `a` and `b` are compatible arrays.
- The verbs appearing in this paper are defined in a *functional* or *tacit* manner without explicit arguments, and, for example, we have seen the verb `am=: +/ % #` for the arithmetic mean. However, *explicit* verbs may be defined where the arguments are specified and where the definition may extend over several lines and involve control structures similar to those in conventional programming languages. Explicit verbs are discussed only briefly in this paper.
- Finally, we mention a construct of considerable usefulness known as a *fork*, an uninterrupted sequence of three verbs, which is a generalization of the notation of conventional mathematics where, for example, *(f+g)x* represents the sum *f(x)+g(x)*. The definition of the arithmetic mean given in the last paragraph is an example.

Part Three. Coupons

Collecting coupons

We shall now introduce a problem which has always been a favourite of mine both because of the simplicity of its solution in an array language and also because of an interesting application. It is known as the coupon collector's problem and describes the random aspects of the repeated purchase of packages of some product such as breakfast cereal until a complete set of prizes, given one prize in each package, is obtained. We are interested in knowing how many packages of cereal we may reasonably expect to purchase in order to

obtain a complete set of prizes.

Mathematical statisticians tell us - and we'll take their word for it - that the average number of packages we will have to buy [to obtain all 5 prizes] may be calculated by finding the reciprocals of the integers 1, 2, 3, 4, and 5, summing these reciprocals, and finally multiplying this sum by 5.

In the following discussion we shall need to introduce the term *reciprocal* which has already appeared in some of the **J** examples of Part Two. It is defined as the quotient of a number divided into unity. For example, the reciprocal of 5 is 1/5 or 0.2 as a decimal fraction. As another example, the reciprocal of 7 is 1/7 or 0.142857... where the ellipsis indicates that the decimal equivalent cannot be expressed exactly to 5 decimal places, or indeed to any finite number of decimal places.

Now let us suppose there are five prizes. It is obvious that we shall have to make at least five purchases. Indeed it is almost certain that we shall have to make more than five since it is unlikely that we shall get a different prize in each of the first five packages that we buy. Mathematical statisticians tell us - and we'll take their word for it - that the average number of packages we will have to buy may be calculated by finding the reciprocals of the first five positive integers, summing these reciprocals, and finally multiplying this sum by 5. We may display the steps in this calculation as follows: The first 5 positive integers are

1, 2, 3, 4, 5;

the reciprocals are

1/1, 1/2, 1/3, 1/4, 1/5

or

1, 0.5, 0.33333, 0.25, 0.2;

their sum is

1 + 0.5 + 0.33333 + 0.25 + 0.2

or

2.28333

which when multiplied by 5 is approximately 11.4. Therefore, on the average 11 or 12 purchases are required to collect all five prizes. The calculation may be stated more concisely as "5 times the sum of the reciprocals of the first 5 positive integers".

We may generalize the problem to an arbitrary number *n* of prizes, where *n* is any positive integer. The problem may now be stated more abstractly as sampling with replacement from the first *n* positive integers until all *n* integers are represented in the sample. Sampling with replacement means that on any draw we have the same chance of selecting any integer, and selecting an integer once doesn't change our chances of selecting it again on a subsequent draw. The expected sample size required to obtain all *n* prizes is then "*n* times the sum of the reciprocals of the first *n* positive integers". If *n* is 5, then the expected sample size is 11.4 as we have seen in the last paragraph. If *n* is 10, a slightly longer but quite straightforward calculation shows that the expected sample size is about 29.3. If there are 26 prizes, say the 26 letters of the alphabet which occurred some years ago on the inside of the lids of tubes of Smarties, the expected number is about 100. Finally, if *n* is 1, the expected sample size is 1 (1 times the sum of the reciprocal of 1, or 1 times 1, or 1) which is reasonable since if there is only one prize it will be obtained on the first purchase.

In the next section we shall discuss the **J** calculations for the expected number of purchases for an arbitrary number of prizes, and give both a **J** verb and a Windows form for performing these calculations. In the following two sections we shall give a few more details on the calculations and also a simulation which will allow an experimental study of collecting coupons without buying or eating any cereal at all.

Coupon calculations

The calculations in **J** for five prizes may be represented as follows with annotations for the reader at the end of the expressions to be evaluated:

```
   i. 5      NB. Non-negative integers
0 1 2 3 4
   >: i. 5       NB. Positive integers
1 2 3 4 5
   % >: i. 5     NB. Reciprocals
1 0.5 0.333333 0.25 0.2
   +/ % >: i. 5  NB. Sum
2.28333
   5 * +/ % >: i. 5  NB. Expectation
11.4167
```

This result is in agreement with that found by hand previously.

Now let us repeat these calculations using a variable `n` to which we initially give the value `5`:

```
   n=: 5
   i. n
0 1 2 3 4
   >: i. n
1 2 3 4 5
   % >: i. n
1 0.5 0.333333 0.25 0.2
   +/ % >: i. n
2.28333
   n * +/ % >: i. n
11.4167
```

If we were to define the value of `n` by the statement

```
   n=: 10
```

and repeat the calculations, we would obtain the result `29.2897`. If the packages of cereal containing any one of the prizes cost $3.59 each, then the expected cost of obtaining all of the prizes would be approximately `29.3 * 3.59` or `105.19` dollars.

To avoid having to assign a value to `n` and then evaluating the expression

```
   n * +/ % >: i. n
```

each time we wish to calculate an expected number of purchases, we may define the verb

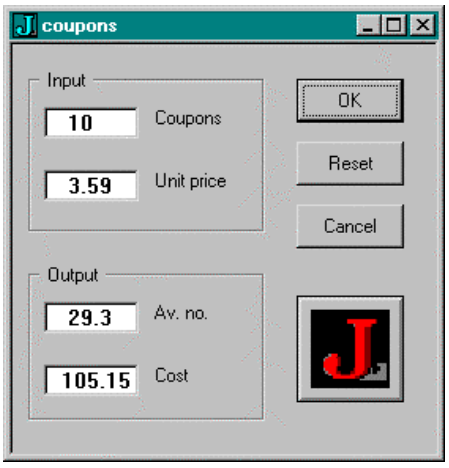
```
   cc=: * +/ @: % @: >: @ i.  .
```

The details of this verb need not concern us here, and we shall simply give a few examples of its use:

```
   cc 5
11.4167
   cc 10
29.2897
   cc 26
100.215
   cc 1
1
   cc 0
0
```

We note that the last result is reasonable since if there are no prizes then there is nothing to purchase.

The figure in the next column shows a Windows form for performing conveniently the calculations for an arbitrary number of prizes. The user enters the number of prizes and the cost per package of the cereal in the boxes in the Input frame, clicks the OK button, and reads the expected number of purchases and the expected cost given in the boxes in the Output frame. The Reset button merely clears the



two output boxes of whatever numbers they contain. The Cancel button is used to exit from the Windows form.

Coupons again

Let us have another look at presenting the results of the calculation of expected values in the coupon collector's problem. We may use rational numbers to calculate exact values of the expectations, and we may also compute tables of expectations, either as decimal or as rational numbers, for a range of values of the number of prizes.

Rational numbers are represented in **J** by the decimal digits of the numerator and denominator separated by an `r` and preceded optionally by a sign. For example, `4r5` is the rational number four-fifths which is represented conventionally as 4/5 and `_3r2` is negative three-halves. The following sequence shows how the calculations for the average number of purchases for five prizes may be performed exactly using rational arithmetic:

```
   1r1 + 1r2
3r2
   3r2 + 1r3
11r6
   11r6 + 1r4
25r12
   25r12 + 1r5
137r60
   5r1 * 137r60
137r12
```

These calculations may be performed more concisely as

```
   5r1 * (1r1 + 1r2 + 1r3 +1r4 +1r5)
```

or

```
5r1 * +/ % 1 2 3 4 5r1
```
or
```
5 * +/ % 1 2 3 4 5r1 .
```
Each of these expressions has the value `137r12` which is the rational representation of the decimal number `137%12` or `11.4167`. The verb `cc` may be used to give a rational result, and `cc 5r1` is `137r12`.

The expression `cc 10r1` has the value `7381r252` and `7381%252` is `29.2897` which is the decimal value obtained previously. Approximate decimal equivalents of rational representations may not be readily apparent for large numerators and denominators, and, for example, `cc 26r1` is
```
34395742267r343219800
```
which is much more difficult to interpret than the approximate decimal value `100.2`.

A table of expectations may be found very simply for arbitrary ranges of row and column arguments. One such table giving the expectations rounded to the nearest integer for `1` to `50` prizes, inclusive, is
```
ÚÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
³  ³  1   2   3   4   5³
ÃÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
³ 0³  1   3   6   8  11³
³ 5³ 15  18  22  25  29³
³10³ 33  37  41  46  50³
³15³ 54  58  63  67  72³
³20³ 77  81  86  91  95³
³25³100 105 110 115 120³
³30³125 130 135 140 145³
³35³150 155 161 166 171³
³40³176 182 187 192 198³
³45³203 209 214 219 225³
ÀÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```
The method of generating bordered tables such as this one will be discussed in a later section.

Collecting free coupons

All of the calculations for the coupon collector's problem have been concerned with the average number of purchases that must be made to collect a given number of prizes. It would be of interest to have some idea how many packages of cereal we might have to buy if we wanted a complete set of prizes. A convenient, and very much cheaper, alternative to actually buying the cereal is to simulate the purchases on a computer. In this section we shall present the results of such a simulation using a number of **J** verbs the details of which will be omitted.

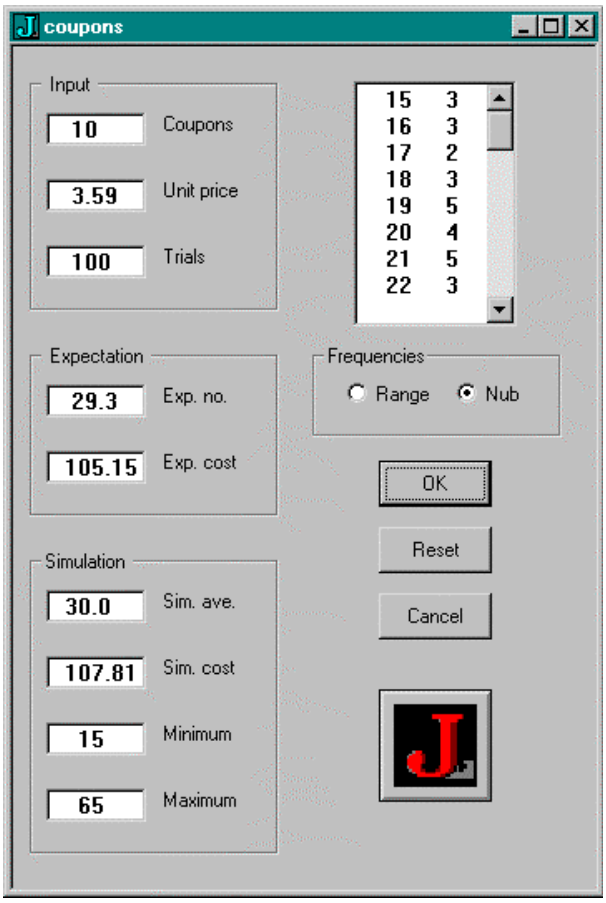
A simulation for `5` different prizes gave the result
```
1 4 3 3 2 1 4 4 5
```
indicating that a total of `9` purchases were required with prize `1` being obtained on the first purchase, `4` on the second, `3` on the third, etc. Two more simulations for `5` prizes gave the results
```
2 3 5 1 1 3 4
```
and
```
1 4 2 4 4 5 2 2 5 4 4 4 1 4 5 2 3
```
with `7` and `17` purchases, respectively, required. We recall that the average number of purchases for `5` prizes is `11.4`. Instead of listing the actual sequence of purchases we could give only the number required to get a complete set of prizes. Again for `5` prizes, `12` simulations gave the sample sizes
```
17 6 8 7 6 14 11 8 24 16 6 13
```
with an average of `11.3`.

Finally let us consider again purchasing packages of cereal at $3.59 a package in order to obtain a set of 10 prizes. One hundred simulations gave the following results which have been arranged for convenience in a ten-by-ten array
```
27 28 28 30 14 14 28 42 30 26
16 29 30 16 23 40 44 14 20 19
28 24 37 30 24 13 23 29 21 27
39 26 28 28 30 26 21 28 18 31
29 29 35 32 31 22 22 29 30 26
49 29 25 49 22 54 24 24 30 21
74 47 33 18 18 14 26 21 47 16
18 29 20 24 26 36 40 39 13 47
28 27 41 45 38 48 34 29 23 29
62 36 35 23 20 29 29 46 28 22
```
The minimum and maximum sample sizes are `13` and `74`, respectively. The average sample size is `29.39` which is extremely close to the expected value of `29.29`. The maximum and minimum costs are `46.67` dollars and `265.66` dollars, respectively. The results of the simulation may also be summarized in the following frequency table where the first column gives the midpoints of the classes and the second column the corresponding frequencies:
```
10  0
15  9
20 16
25 19
```

```
30 30
35  7
40  7
45  6
50  3
55  1
60  1
65  0
70  0
75  1
```

The first row shows that there are `0` sample sizes of `8`, `9`, `10`, `11` and `12`, the second row that there are `9` sample sizes of `13`, `14`, `15`, `16` and `17`, etc.

We give in the next column a Windows form that allows us to perform very simply all of the calculations and simulations we have discussed. The radiobuttons Range and Nub given in the Frequencies frame allow the display of the frequencies over either the entire range of sample sizes from zero to the maximum or only the non-zero sample sizes which actually occurred in that particular simulation.



Part Four. Numbers

Numbers Without End

J may be used very effectively as a convenient computational supplement to the exposition of many topics. Instead of writing an entire book on some subject for which all too many books may already exist, it is reasonable to write a "**J** companion" to some well-known text, posssibly a classic in its field, that will allow the numerical examples to be simply carried out and extended. Kenneth Iverson began this with his *Concrete Math Companion* (Iverson Software Inc., 1995) which was intended as a supplement to a well-known computer science text. I have attempted this for statistical computations in several papers, with the most recent paper intended as a companion for a typical introductory statistics text.

As an illustration of what may be done consider *Numbers Without End* by Cornelius Lanczos (Oliver and Boyd, Ltd., 1968). This little book - it is a small paperback with only 164 pages - has long been a favourite of mine. In it the author, a respected mathematician and physicist, is concerned with the beauty of elementary mathematics rather than with its practical utility. "Numbers," he says, "are much more than figures in a bank account." He is interested in mathematics as "a cultural subject which [has] played a vital role in the evolution of the human intellect", a subject which deserves study because of its "inherent beauty". A rereading of this book with the possibilities of **J** in mind is illuminating. A few examples must suffice.

We shall make use of the adverb `table` for producing tables which may aid in the understanding of many functions, both those which we have seen before and those which may be new to us. This adverb, which will be considered a "utility" since we shall simply use it without being concerned with its definition, gives a table of values of a given verb for specified left and right arguments which border the table to the left and on the top. For example, if

```
x=: 1 2 3 4 5 6 7 8 9 10 11 12
```

is the list of the first `12` positive integers, then the expression

```
* table x
```

gives the following multiplication table which is still shown in some school exercise books:

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	14	16	18	20	22	24
3	3	6	9	12	15	18	21	24	27	30	33	36
4	4	8	12	16	20	24	28	32	36	40	44	48
5	5	10	15	20	25	30	35	40	45	50	55	60
6	6	12	18	24	30	36	42	48	54	60	66	72
7	7	14	21	28	35	42	49	56	63	70	77	84
8	8	16	24	32	40	48	56	64	72	80	88	96
9	9	18	27	36	45	54	63	72	81	90	99	108
10	10	20	30	40	50	60	70	80	90	100	110	120
11	11	22	33	44	55	66	77	88	99	110	121	132
12	12	24	36	48	60	72	84	96	108	120	132	144

As another example we introduce the verb `|` *residue* which gives the remainder when one integer is divided by another. For example, the remainder when 10 is divided by 7 is 3 which in **J** is given by `7|10` or `3`. As an interpretation of this expression we note that the day of the week which is 10 days from a given day is the same as the day of the week which is 3 days ahead. Thus residue or modular arithmetic may save considerable finger-counting if we are concerned with an elapse of several weeks as, for example, when finding the day which is 90 days from a given day. (Since `7|90` is `6`, the answer is that it is the same day as the day which is `6` days after the given day.) The definition of this verb may be made clearer if we consider its value for some random arguments as shown in the table

	1	3	10	20	90	100
1	0	0	0	0	0	0
2	1	1	0	0	0	0
3	1	0	1	2	0	1
7	1	3	3	6	6	2
10	1	3	0	0	0	0
15	1	3	10	5	0	10
18	1	3	10	2	0	10

which is given by the expression

```
1 2 3 7 10 15 18 | table 1 3 10 20 90 100
```

In a long chapter on divisibility Lanczos makes repeated use of modular arithmetic in which only the remainders with respect to some divisor are considered. In an informal discussion of one result Lanczos presents a modular-7 multiplication table which in **J** is given by

```
7&|@* table 1 2 3 4 5 6
```

which has the value

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

For example, the entry in the fourth row and fifth column is the remainder when `20`, the product of `4` and `5`, is divided by `7`. In **J** the corresponding expression is `7|4*5` which is `6`. In another discussion the following modular-10 multiplication table, given

```
10&|@* table~ 1 3 7 9  ,
```

is used:

	1	3	7	9
1	1	3	7	9
3	3	9	1	7
7	7	1	9	3
9	9	7	3	1

"Numbers," he [Cornelius Lanczos] says, "are much more than figures in a bank account." He is interested in mathematics as "a cultural subject which [has] played a vital role in the evolution of the human intellect", a subject which deserves study because of its "inherent beauty".

In another chapter, "Evolution of the Number Field", Lanczos discusses various classes of numbers - integers, fractions (rationals), etc. - and their addition, subtraction, multiplication and division. One of the many ways **J** could help us here is with rational arithmetic which, the author demonstrates, is rather cumbersome if done by hand. For example, the sum 2/3 + 1/5 which is 13/15 may be calculated in **J** as `2r3 + 1r5` which is `13r15`. As another example, `1r2 * 4r3` is `2r3`, a calculation which would be required if a recipe which called for half a cup of flour were being increased by a third.

No treatment of the number system would be complete without a discussion of prime numbers.

We recall that a prime number is a positive integer which is divisible only by 1 and itself. For example, 2, 3 and 17 are primes whereas 4 and 15 are not. The two **J** verbs *primes* `p:` and *prime factors* `q:` may be used to perform some of the calculations given in Lanczos. For example, since `i. 13` is the list

```
0 1 2 3 4 5 6 7 8 9 10 11 12
```

of the first `13` non-negative integers, the expression `p: i. 13` is the list

```
2 3 5 7 11 13 17 19 23 29 31 37 41
```

of the first `13` primes given by the author. Also the example in the book of the prime factorization of `2772` could be expressed as `q: 2772` with the value

```
2 2 3 3 7 11
```

so that

```
2 * 2 * 3 * 3 * 7 * 11
```

or

```
*/2 2 3 3 7 11
```

is equal to `2772`.

We shall conclude this section with a few more remarks on prime numbers. The verb

```
np=: p:^:_1  ,
```

the details of which need not concern us, gives the number of primes up to and including its argument. For example, `np 10` is `4` since `2`, `3`, `5` and `7` are the primes less than or equal to `10`. Also `np 42` is `13` and `np 1000` is `168`. We may now define the verb

```
ratio=: % np
```

which gives the reciprocal of the fraction of the number of integers up to a given integer which are prime. For example, `ratio 10` is `2.5` since, as we have just seen, there are `4` primes less than or equal to `10`. Values of this verb for other arguments are the following:

```
   ratio 100
4
   ratio 1000
5.95238
   ratio 10000
8.1367
   ratio 100000
10.4254
```

Further calculations involving larger and larger arguments and the calculation of the ratios of successive values would be helpful in an empirical study of the limiting density of prime numbers. This would give some appreciation of the famous Prime Number Theorem which states that this limit is "*n* divided by the natural logarithm of *n*" as *n* becomes infinitely large.

It is tempting to continue this discussion here both because of the intrinsic interest of the subject and because of the ways in which **J** may be used. However, in the interests of brevity we should probably stop here. Persons wishing to pursue such mathematical topics further might consult the very readable *The Mathematical Experience* by Philip J. Davis and Reuben Hersch (Houghton Mifflin Company, 1982).

More numbers

I think it would be a pity to leave a general discussion of numbers without some mention of numbers to bases other than the base-10 or decimal numbers which we use every day. Indeed, Chapter 2 of Lanczos's book is entitled "The Decimal System", although the title does not prevent him from first discussing the positional number systems of the ancient Sumerians, Babylonians, Hebrews and Greeks. In this section, then, we shall make a few remarks about number systems to various bases and give a few applications which the reader may find interesting.

We are so familiar with the decimal number system that we scarcely give it any thought. Let us remind ourselves that any decimal number consists of a sequence of one or more of the ten digits 0, 1, 2, ..., 9 where the value of each digit depends not only on the digit itself but also on its position in the sequence. For example, the number 2485 represents 2 thousands plus 4 hundreds plus 8 tens plus 5 ones, or

$$2 \times 1000 + 4 \times 100 + 8 \times 10 + 5 \ ,$$

or more concisely

$$2 \times 10^3 + 4 \times 10^2 + 8 \times 10^1 + 5 \ .$$

(The last term is 5×10^0 which is 5×1 or 5 since any number raised to the power of 0 is 1.)

I have found it useful in reviewing the decimal number system in classes to introduce a children's guessing game that will help reinforce the positional nature of our decimal numbers. There are several that could be used; the one given here is typical. Ask one of the class to select two digits and perform the following operations on them: Choose one of the digits, multiply by 5, add 7, double, and add the other digit. Then have the student tell you the result, and you immediately tell him or her the two digits

that were selected. This is accomplished by simply subtracting 14 from the result, and the two digits are the digits of the resulting two-digit number. To explain this trick we shall pick two digits, 3 and 8, say, and display the calculations in three columns where the second column gives the result of the operation performed and the third gives a record of the operations performed thus far:

Pick 1st digit	3	**3**
Multiply by 5	15	5 × **3**
Add 7	22	7 + (5 × **3**)
Double	44	2 × (7 + (5 × **3**))
Add 2nd digit	52	**8** + (2 × (7 + (5 × **3**)))
		8 + (14 + (10 × **3**))
		(**3** × 10 + **8**) + 14
Subtract 14	38	**3** × 10 + **8**

The choice of 10 for the base of our number system is due to us having ten fingers. However, any positive integer greater than 1 may be used as a base, and, indeed, negative and non-integer bases may be used. Three number systems which are useful in computing are the binary or base-2, octal or base-8 and hexadecimal or base-16 systems. Each of these will be considered briefly. Also we shall consider so-called mixed-base number systems which may be handled simply in **J** and which have some interesting applications.

"... The ultimate aim is to persuade the whole civilized world to abandon decimal numeration and to use octonal numeration in its place; to discontinue counting in tens and to count in eights instead. "

E. W. Phillips (1936)

A binary number consists of a sequence of the digits 0 and 1 with the value of each digit obtained by its multiplication by the appropriate power of 2. For example, the binary number 11101 represents the decimal number

$$1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1$$

or

$$1 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1$$

or

$$16 + 8 + 4 + 0 + 1$$

or 29. Likewise an octal number consists of a sequence of the eight digits 0, 1, 2, ..., 7, and their values are determined by multiplication by the appropriate powers of 8. For example, the octal number 375 is equivalent to the decimal number

$$3 \times 8^2 + 7 \times 8^1 + 5$$

which when evaluated is 253. Hexadecimal or base-16 numbers require sixteen different digits which by convention are 0, 1, 2, ..., 9 and A, B, C, D, E and F with the letters representing 10, 11, ..., 15, respectively. An example of a hexadecimal number is A6F which represents the decimal number

$$\text{"A"} \times 16^2 + 6 \times 16^1 + \text{"F"}$$

or

$$10 \times 16^2 + 6 \times 16^1 + 15$$

or 2671.

Various proposals have been made to replace our decimal number system by other systems, the most common being the duodecimal or base-12 which, its proponents argue, would be more convenient with the divisors 2, 3, 4 and 6 for its base than our decimal system whose base has only the two divisors 2 and 5. An English actuary, E. W. Phillips, proposed the adoption of the binary system in a paper "Binary calculation" presented in 1936 and which has been reprinted in *The Origins of Digital Computers. Second Edition* edited by Brian Randell (Springer-Verlag, 1975). In the introduction to the paper the author states his purpose as follows:

> ... The ultimate aim is to persuade the whole civilized world to abandon decimal numeration and to use octonal numeration in its place; to discontinue counting in tens and to count in eights instead.
>
> However, it seems unlikely that the whole civilized world will be persuaded to complete this change during the next twelve months, having previously declined similar invitations. Therefore the more immediate aim is the adoption of octonal numeration for scientific and business purposes, for the great mass of figures recorded and manipulated for the benefit only of the scientific and business man, the few final results required for presentation to the layman being transformed into the denary scale of notation from the octonary by means of conversion tables, ot otherwise.

I was introduced to various positional number systems very earlier in my computing career since,

as I mentioned earlier in this paper, the commands for the first computer I programmed, the NCR 102A, were expressed octally. Furthermore, data were entered in decimal but had to be converted to binary before they were used, and conversely any results that were printed had to be converted from binary to decimal.

The hexadecimal system is undoubtedly familiar to many users of computers, especially to those who have made anything but casual use of a dot-matrix printer. Manuals for these printers contained a table giving the decimal and hexadecimal codes for each of the 256 ASCII characters that could be printed. For example, the carriage return CR was decimal 13 and hexadecimal 0D, "A" was decimal 65 and hexadecimal 41, and so on, and so on - at times, seemingly without end. For reasons which fortunately I have now forgotten, it was often necessary to consult these tables in order to have the computer print what we wanted it to print in the format that was required. Today hexadecimal codes appear in the HyperText Markup Language (HTML) used in the preparation of Web pages in, say, specifying font colours. For example, the command

```
<FONT COLOR="#FF0000">
```

gives a red font,

```
<FONT COLOR="#0000FF">
```

a blue font, and

```
<FONT COLOR="#FF00FF">
```

a magenta font which is a combination of red and blue.

One reason, I believe, that I have been interested in positional number systems is that they may be handled very simply in **J** (and previously in APL). The *base* verb `#.` may be used to convert from an arbitrary base to decimal, and the *antibase* verb `#:` for conversion from decimal to an arbitrary base. The following simple examples illustrate their use and should be self-explanatory:

```
   #. 1 1 1 0 1
29
   2 #. 1 1 1 0 1
29
   8 #. 3 7 5
253
   16 #. 10 6 15
2671
   #: 29
1 1 1 0 1
   2 2 2 2 2 #: 29
1 1 1 0 1
   2 2 2 #: 29
1 0 1
   8 8 8 #: 253
3 7 5
   16 16 16 #: 2671
10 6 15
```

The right argument need not be a single integer, and, for example, we have that `#: i. 4` is

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

which gives a table of the binary representations of the integers `0`, `1`, `2` and `3`.

A positional number system may have a mixed based where the multipliers for the successive digits are not successive powers of the base. An especially useful mixed base may be used for conversion from hours, minutes and seconds to seconds. For example, the expression

```
   24 60 60 #. 5 25 10
```

is `19510` which is the number of seconds in `5` hours, `25` minutes and `10` seconds, and

```
   365 24 60 60 #. 1 0 0 0
```

is `86400`, the number of seconds in one day. Conversely,

```
   24 60 60 #: 19510
```

which is equal to `5 25 10` represents the converse operation of converting from seconds to hours, minutes and seconds.

As an example of the use the mixed-base number system introduced in the last paragraph; suppose we have timed an event that began in the afternoon at 25 minutes and 10 seconds after 5 o'clock, and ended about two hours later at 7 hours, 15 minutes and 8 seconds, and that we wish to find the elapsed time in hours, minutes and seconds. First we define

```
   T=: 7 15  8; 5 25 10
```

which is the two-item list

```
   ÚÄÄÄÄÄÄÂÄÄÄÄÄÄ¿
   ³7 15 8³5 25 10³
   ÀÄÄÄÄÄÄÁÄÄÄÄÄÄÙ
```

Then the expression

```
   24 60 60&#. Each T ,
```

where `Each` is a utility adverb which applies its verb left argument to each item of its right argument, gives us the stopping and starting times in seconds,

```
   26108 19510 .
```

The elapsed time may be given as

```
   -/26108 19510
```

or 6598 seconds, or as

```
   24 60 60 #: 6598
```

which is 1 hour, 49 minutes and 58 seconds. These calculations may be summarized in the single expression

```
24 60 60 #: -/ 24 60 60&#. Each T .
```

The binary and hexadecimal number systems may be used to give some understanding of the RGB colour model in which all colours are considered to be some combination of the colours red, green and blue. To give any colour the appropriate proportions of these three colours are used. Colour monitors use three colour tubes and colour printers three coloured inks to give the range of colours. For example, if all three basic colours are absent, the resulting composite colour is black, and if all three are fully used the result is white. We may represent the eight colour combinations resulting from the presence or absence of each of the basic colours by a binary number as shown in the following list:

0 0 0	Black	1 0 0	Red
0 0 1	Blue	1 0 1	Magenta
0 1 0	Green	1 1 0	Yellow
0 1 1	Cyan	1 1 1	White

The colour model may be visualized as a three-dimensional unit cube with the vertices at (0,0,0), (1,0,0), ... Points on the faces of the cube or inside the cube represent other combinations of the three basic colours.

In some representations and use of the RGB model, it is convenient to represent the combinations of colours with hexadecimal, rather than binary, numbers or their decimal equivalents. Thus black would be "000000" or (0,0,0), red "FF0000" or (255,0,0), and magenta "FF00FF" or (255,0,255), etc. Theoretically, at least, one would have a total of 256^3 or 16 777 216 colour combinations.

Really large numbers

In this section we shall have a look at some problems which give rise to very large numbers and see how the exact computations may be handled very simply in **J**. In most of the examples in this section we shall make use of the extended precision integer arithmetic facilities in **J** which allow the computation of integers of arbitrary length. This section may be considered by some readers to be too long or too trivial, or both, and they may be correct. My only excuse for giving these examples is that I have found them interesting and at times instructive.

To begin we shall consider the factorial function which is very simple and easy to understand and yet has some interesting applications. The factorial function is defined in terms of the product of the positive integers 1, 2, 3, ... For example, *factorial 3* is 6 which is the product of the integers 1, 2 and 3, *factorial 5* is 120 which is the product of the first five positive integers. In general, *factorial n* is

$n \times (n-1) \times (n-2) \cdots 3 \times 2 \times 1$.

The factorial function may be used to give concise expressions for representing the number of permutations and combinations of any number of objects taken so many at a time. As just one example, the number of arrangements of the 52 cards in a deck is given by *factorial 52* which is equal to approximately 8 followed by 67 zeros, a very large number indeed.

We may note that *factorial 0* is defined as 1 so that the factorial function will behave reasonably in limiting situations involving, say, permutations and combinations as, for example, in the number of ways of choosing five apples five at a time which is obviously 1.

So there is considerable truth in a recent newspaper statement that "when you shuffle a deck of cards, there's a good chance that the exact order of those 52 cards has never occurred before".

In **J** the factorial function is given by the *factorial* verb `!`, and, for example, `!3` is `6`, `!5` is `120`, and `! i. 9` is

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

which has the value

```
   1 1 2 6 24 120 720 5040 40320 .
```

The factorial verb can easily produce very large numbers, and, for example, `!10` is `3.6288e6`, `!20` is `2.4329e18`, and `!52` is `8.06582e67` as we have seen. To get some idea of the size of this number assume that we could shuffle a deck of cards once every nanosecond, or one billion times a second, and

that we obtained a new arrangement on each shuffle. If this had been done continuously since the creation of the earth some five billion years ago, we would have seen only about `1.5768e17` different arrangements. Suppose, now, to speed up this process we had recruited six billion people, roughly the present population of the Earth, and had been able to put them all to work at the moment of creation with each person shuffling a billion times per second. At the present time we would have seen only about `9.4608e26` different arrangements. So there is considerable truth in a recent newspaper statement that "when you shuffle a deck of cards, there's a good chance that the exact order of those 52 cards has never occurred before".

The factorial function may be computed exactly in **J** by suffixing the argument with `x`, and `!10x` is `3628800`, `!20x` is

```
2432902008176640000  ,
```

and `!52x` is

```
80658175170943878571660636856403766975289505440883277824000000000000 ,
```

an integer with `68` digits. The number of bridge hands may be shown to be

```
(!52x) % (!13x)^4 ,
```

where the denominator is

```
(!13x) * (!13x) * (!13x) * (!13x) ,
```

and is equal to

```
53644737765488792839237440000 .
```

Now that we have introduced extended integers, we may, by way of an aside, refer to a remark by the nineteenth-century British economist and logician W. S. Jevons and quoted by Martin Gardner in one of his many books: "Can the reader say what two numbers multiplied together will produce the number 8,616,460,799? I think it is unlikely that anyone but myself will ever know; for they are two large prime numbers." The answer has undoubtedly been found many times since then for it is a simple calculation with a computer. If we recall that the **J** verb `q:` gives the prime factors of an integer, then the answer is given very simply by

```
q: 8616460799x
```

which has the value

```
89681 96079 ,
```

so that

```
89681 * 96079x
```

is equal to `8616460799`.

"Can the reader say what two numbers multiplied together will produce the number 8,616,460,799? I think it is unlikely that anyone but myself will ever know; for they are two large prime numbers."

W. S. Jevons (1835 - 1882)

There is an old legend repeated in many books that gives rise to a very large number. According to this story an Indian king wanted to reward one of his officials for inventing the game of chess. When asked if there was anything he wanted, the official replied that he would be quite happy with an amount of wheat equal to that obtained by placing one grain of wheat on the first square of the chessboard, two on the second, four on the third, eight on the fourth, and so on with the number of grains of wheat increasing in a geometric progression. The unsuspecting king was only too happy to comply with this request. Of course, the outcome of the story is not known since the total number of grains of wheat required to satisfy the official's request may be shown to be $2^{64} - 1$, or approximately 1.8×10^{19}, in conventional notation. (A calculation I made a few years ago determined that this number amounted to the world's wheat production for approximately 330 years.) The exact value may be computed very simply in **J** as `_1+2^64x` which is

```
18446744073709551615 .
```

Another problem that gives rise to the same large numbers as does the chessboard example is a puzzle which is usually called the Tower of Hanoi. This puzzle consists of a small board to which are fastened three vertical pegs on one of which are stacked a number of discs of diminishing radii with the smallest disc on top, the next smallest disc below it, and so on. The problem is to transfer the discs to a second peg by moving the discs one at a time using the third peg when necessary and never placing a disc on top of a smaller disc. The Tower of Hanoi puzzle was invented by the French mathematician Edward Lucas in the 1880s. It was sold as a toy and described as a simplified version of a mythical Tower of Bramah to which was associated the following story: In a temple in Benares under the

dome which marks the centre of the world there is a brass plate to which are fixed three diamond needles. At the creation of the world God placed sixty-four discs of gold on one of the needles, the largest resting on the brass plate and the others on top becoming smaller and smaller. The priests in the temple have been working day and night transferring the discs one at a time, never placing a disc on a smaller one. When all sixty-four discs have been transferred from the original needle to one of the others, the temple will crumble into dust and the world will vanish in a clap of thunder.

Now consider the minimum number of moves that are required to move an arbitrary number of discs. If there is only one disc, then obviously it may be moved to a second peg in a single move. If there are two discs on the first peg, they may be moved to the second peg in three moves by moving the top disc to the third peg, then moving the larger disc to the second peg, and finally moving the smaller disc from the third peg to the second peg. A little thought, or better still some experimentation with a model, will show that three discs may be moved in a minimum of seven moves. With sufficient patience and interest one could move four discs in fifteen moves with the feeling that it would be impossible to do better. Since these numbers form the series 1, 3, 7, 15 which may be written as $2^1 - 1$, $2^2 - 1$, $2^3 - 1$ and $2^4 - 1$ where the number of discs appears as an exponent, one is tempted to form the conjecture that for any positive integer n the minimum number of moves required to move the n discs is $2^n - 1$. Although it is a relatively simple matter to prove this conjecture, we shall not do so here but we shall return to it at the end of the Appendix to this section. Therefore, for an eight-disc puzzle which I have used while writing this account and which I shall mention again in a later section the minimum number of moves is $2^8 - 1$ or 255. The general expression holds also for no discs for then n is zero and $2^0 - 1$ is $1 - 1$ which is 0, since any number raised to the power of zero is 1. This is reasonable since if there are no discs there is no puzzle to solve. We recall a similar result for the coupon collector's problem where the average number of purchases is zero when there are no coupons to collect.

If we return to the priests in the temple in Benares, we see that the minimum number of moves required for them to complete their task is $2^{64} - 1$ or approximately 1.8×10^{19} which is the same number which occurred in the chessboard problem. If we assume that the priests are able to move one disc a second and work in shifts unceasingly night and day, the total number of years required to complete their task would be about six hundred thousand million years.

The title story ["The Nine Billion Names of God"] was written, for want of anything better to do, during a rainy weekend at the Roosevelt Hotel. Its basic arithmetic was later challenged by J. B. S. Haldane, but I managed to save the situation by alphanumeric evasions whose precise nature now escapes me.

Arthur C. Clark

I was reminded of the story associated with the Tower of Hanoi puzzle when I first read, many years ago, Arthur C. Clarke's "The Nine Billion Names of God" in which the monks in a Tibetan monastary acquire a "Mark V Automatic Sequence Computer" to assist them in enumerating all of the valid names of God, a task which they had been working on by hand for the last three centuries. They believe that when their work is finished, as one of the engineers who accompanied the computer to Tibet finally discovered, the universe will end. This story, first written in 1952, was reprinted in 1966 as the title story in a collection Clarke's best short stories. At the beginning of this story Clark makes the following remarks:

> The title story was written, for want of anything better to do, during a rainy weekend at the Roosevelt Hotel. Its basic arithmetic was later challenged by J. B. S. Haldane, but I managed to save the situation by alphanumeric evasions whose precise nature now escapes me.
>
> ...
>
> Nevertheless, I appear to have created a durable myth: not long ago, a radio talk on the BBC referred to the opening situation of this story as actual fact. And now that IBM computers have entered the field of bibical scholarship, perhaps this theme is coming a little closer to reality.

If the numbers we have introduced so far are not large enough for some readers, we might remark that the distinguished British astronomer Sir Arthur Eddington who died in 1944 once estmated that the total number of particles in the universe was

$3/2 \times 2^{256} \times 138$

which is approximately 2.4×10^{79}. (Different references give slightly different values for this estimate but all are of the same order of magnitude.) In **J** this expression may be computed exactly as

```
207 * 2^256x
```

which is the 80-digit integer

```
23968962472124452452679193896798396
92562688682578759675616771988963801
7835466752 .
```

Lest readers think that extended precision integer arithmetic is required for all calculations involving really big numbers we shall give two examples using compound interest calculations which may be performed very simply with the arithmetic operations with which we are familiar. The first example comes from the short story "John Jones's Dollar" by H. S. Keeler first published in 1927 and republished in *Fantasia Mathematica* by Clifton Fadiman (Simon and Schuster, 1958). The story concerns John Jones who in 1921 invested one dollar at three percent interest compounded annually with the stipulation that the principal and interest were to go to his fortieth descendant. Unfortunately in the twenty-fifth century life expectancy increased to two hundred years so that when the thirty-ninth descendant was born, some 1000 years after the initial dollar was deposited, the money had grown to about seven trillion dollars which was more wealth than was available in all of the inhabited solar system. The author resolved this dilemma by having the thirty-ninth descendant have a disagreement with his fiancée so that the marriage was called off and the man died without heirs. (The calculation may be represented very simply in **J** as `1.03^1000` which has the value `6.87424e12`.)

The second example was proposed by the nineteenth-century German mathematician Johann Martin Bartels and concerns the accumulated amount of one unit of currency over a very long period of time. His problem may be considered to be equivalent to finding the size of a sphere of solid gold that could be purchased at the end of 6500 years by investing one dollar at five percent interest compounded annually. With gold worth $437.09 Canadian a troy ounce and using a value of 0.698 pounds per cubic inch, it is a simple if somewhat tedious calculation to show that this golden sphere would have a radius of about 8.8×10^{26} light years.

Our final example, which will be continued in the Appendix to this section, has been suggested by a discussion in Vikram Seth's novel *A Suitable Boy* (Little, Brown, 1993) set in the newly independent India of the early 1950s. Fairly early in this long novel - approximately 1800 pages in the three-volume paperback edition which I have - we are introduced briefly to Dr. Durrani, a tall white-haired mathematician who is one of the two Fellows of the Royal Society at Brahmpur University. He wanders into a small social gathering with the following problem:

> "Now you see, Patwardhan," - Dr. Durrani treated the whole world on terms of gentle distance - "it isn't just a question of 1, 3, 6, 10, 15 - which would be a, er, trivial series based on the, er, primary combinative opertaion - or even 1, 2, 6, 24, 120 - which would be based on the secondary combinative operation. It could go much, er, much further. The tertiary combinative operation would result in 1, 2, 9, 262144, and then 5 to the power of 262144. And of course that only, er, takes us to the fifth term in the, er, third such operation. Where will the, er, where will the steepness end?" He looked both excited and distressed.
>
> ...
>
> "But, of course, what I am saying is, er, quite obvious. I didn't mean to, er, er, trouble you with that. But I did think that I, er," - he looked around the room, his eye alighting on a cuckoo-clock on the wall - "that I would, well, pick your brains on something that might be quite, er, quite unintuitive. Now take 1, 4, 216, 72576 and so on. Does this surprise you?"

It will be interesting to have a brief look at Dr. Durrani's problem using extended precision arithmetic when necessary. The first series is given by the cumulative sums of the successive positive integers,

1, 1 + 2, 1 + 2 + 3, 1 + 2 + 3 + 4, ...

or 1, 3, 6, 10, ..., and the second series by the

cumulative products, or factorials,

1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, ...

or 1, 2, 6, 24, ...These first two series present no problems as long as it is kept in mind that the factorials can become quite large.

However, the third series represents the cumulative powers of the integers and successive terms become very, very large very quickly. The first term is 1, and the second term is

2 to the power of 1

or 2^1 or 2. The third term is

3 to the power of 2 to the power 1

or 3^2 or 9. The fourth term is

4 to the power of 3 to the power of 2 to the power of 1

or 4^9 or 262144. If we number the terms 1, 2, 3, 4, ..., we see that the value of any term is given by the sequence number of the term raised to a power equal to the value of the previous term, with the proviso that the value of the first term is 1. The fifth term may be seen to be 5^{262144} which is an unconceivably large number. If calculated exactly in **J** by the expression `5^262144x`, we obtain a truly gigantic number of `183231` digits. We give only the first ten and last ten digits,

`6206069878 ... 8212890625` ,

since a listing of the entire number would require some forty-eight pages.

Returning to Seth's novel, we see that Dr. Durrani is grappling with a hierarchy of "super-operations" with each operation being defined in terms of the previous operation. Thus we may define multiplication in terms of addition, and, for example, 2 × 4 is 2 + 2 + 2 + 2 or 8. Then we may define raising to a power in terms of multiplication and 2 to the power of 4 is 2 × 2 × 2 × 2 or 16. The next operation, which we will denote by *f* for want of a better symbol, may be defined in terms of raising to a power, and 2 *f* 4 is "2 to the power of 2 to the power of 2 to the power of 2" or 65536. The operation after this one, *g*, say, may be defined in terms of *f*, and 2 *g* 4 is 2 *f* 2 *f* 2 *f* 2. What is its value, or how many digits does it have, or how would it be computed? Dr. Durrani seems perplexed with these questions - and so are we.

What is the last series that Dr. Durrani mentions? I simply don't know. Even writing the terms as products of their prime factors,

1, 2^2, $2^3 \times 3^3$, $2^7 \times 3^4 \times 7$,

doesn't seem to help.

Let us use **J** to explore Dr. Duranni's problem we have just discussed. We may define multiplication in terms of addition by the verb

```
mpy=: +/@$~
```

where `$` is the verb *shape* and `~` is the adverb *passive* which reverses the arguments of the verb. For example, `3 $ 4` is `4 4 4`, `3 $~ 4` which is equivalent to `4 $ 3` is `3 3 3 3`, and `3 mpy 4` is `12`. Similarly, we may define

```
power=: */@$~
```

and `3 power 4` is `81`. Now we may define the next operation in the hierarchy as

```
f=: ^/@$~  ,
```

and `2 f 3` is `2 ^ 2 ^ 2` or `16`. The next operation is

```
g=: f/@$~
```

and `2 g 3` is `65536`. Thus we may formally at least represent a hierarchy of operations although we may calculate very few values and apparently have little or no understanding of how they may be interpreted.

In our discussion of the Tower of Hanoi puzzle we accepted without proof the expression for the minimum number of moves for an arbitrary number of discs. It may be of some interest to give the proof, which is really quite simple, although any enjoyment of the puzzle is independent of its understnading. The proof depends on what is known as mathematical induction where we first assume that what we have to prove is true for one particular instance - in this example for a given number of discs - and then showing that the result holds for a number of discs one greater than the given number. Then as the result is trivially true for one disc, it follows that it is true for two discs, and then true for three discs, and so for any number of discs however large. The following proof, then, gives this inductive proof in more formal language:

Theorem: The minimum number of moves required to solve the Tower of Hanoi puzzle for *n* discs, where *n* is a positive integer, is 2^n - 1.

Proof: Assume that the theorem is true for *k* discs so that the minimum number of moves is 2^k - 1. Now let us find the minimum number of moves required for *k* + 1 discs. We can visualize

solving the puzzle for $k + 1$ discs by first moving the top k discs from the first peg to the third peg, say, then moving the bottom disc - the $(k + 1)$st disc - from the first peg to the second peg, and finally moving the k discs from the third peg to the second peg. The total number of moves required to move the $k + 1$ discs is thus

$$(2^k - 1) + 1 + (2^k - 1)$$

or

$$(2 \times 2^k) - 1$$

or

$$2^{k+1} - 1 \ .$$

Thus if the theorem is true for k discs, it is true for $k + 1$ discs where k is any positive integer.

Therefore for $k = 1$ the minimum number of moves is $2^1 - 1$ which is $2 - 1$ or 1. This is obviously true since we simply move the one disc from the first peg to the second peg. Therefore, since the result is true for one disc, by the result established in the last paragraph it is true for two discs. Then by the same reasoning since the result is true for two discs it is also true for three discs, and so on. Therefore, the theorem is true for n discs where n is any positive integer.

Part Five. A Miscellany of Examples

An orderly arrangement of data

Given a set of observations arising in some experimental situation we often wish to divide the total variability of the data into several components defined by different factors and then determine whether the variability within or between these components is due to random variation in the data or to some real differences inherent in the factors of classification. For example, the data may be yields of some crop to which different fertilizers have been applied and which have been subject to different methods of cultivation, and we would like to determine if either the type of fertilizer or the method of cultivation affected the yield or if the effect of the fertilizer depended on the method of cultivation. The design of suitable experiments, the analysis of the resulting data, and the interpretation of the results of these analyses constitute a very important application of statistics, one in which computers have played an important role.

In this paper we will be concerned, and only very briefly at that, with one aspect of the calculations required in such analyses regardless of the provenance of the data, the type of design and the interpretation of the results. As is customary in these analyses we shall assume that the data are arranged in a rectangular array, i.e., a one-dimensional list, a two-dimensional table in which the data are arranged in rows and columns, etc. At the heart of any of these calculations is the calculation of some simple sums such as the sum of a list of data, the row and column sums in a table, and the various marginal sums when the data are arranged in more than two dimensions. It is with these sums that we will be concerned.

Before we introduce our example we might remark that the *American Heritage Dictionary* defines an array as "An orderly arrangement, especially of troops" and that the title of this section was suggested by this definition. An alternative mathematical definition is given as "A rectangular arrangement of quantities in rows and columns, as in a matrix." However, the present variation may be preferable as it does not limit us to two dimensions.

As an example we shall use some data taken from *Principles and Procedures of Statistics* by R. G. D. Steel and J. H. Torrie (McGraw-Hill, 1960) and even then we shall use only part of the data. The data are the yield in bushels per acre for each of two varieties of oats and each of three different seed treatments with four replications of each variety-treatment combination.

To begin, consider the four observations

```
62.3 58.5 44.6 50.3
```

for the first variety-treatment combination. There are the observations themselves and also their sum `215.7` which is a measure of the effectiveness of this particular combination.

Now consider all of the four treatments for the first variety which are given by the following two-dimensional array:

```
62.3  58.5  44.6  50.3
63.4  50.4  45.0  46.7
64.5  46.1  62.6  50.3
```

with the rows representing treatments and the columns representing replications. There are now

four different quantities to calculate: the observations themselves, the row sums

```
215.7 205.5 223.5
```

which give a measure of the effectiveness of each of the three treatments, the column sums

```
190.2 155 152.2 147.3
```

which measure the variability between the replications, and the overall sum `644.7` which gives a measure of the yield of the first variety of oats.

If we now consider the second variety of oats, we have the data arranged in the three-dimensional array

```
62.3  58.5  44.6  50.3
63.4  50.4  45.0  46.7
64.5  46.1  62.6  50.3

75.4  65.6  54.0  52.7
70.3  67.3  57.6  58.5
68.8  65.3  45.6  51.0
```

which has `2` levels each with `3` rows and `4` columns with the levels representing the varieties and the rows and columns representing treatments and replications, respectively. If we count the array itself and the total over all of the data, there will be `2 * 2 * 2` or `8` different marginal sums to compute. For example, the sum over the levels

```
137.7 124.1  98.6 103.0
133.7 117.7 102.6 105.2
133.3 111.4 108.2 101.3
```

measures the yields of varieties for both treatments and replications, and the sum over both levels and replications

```
463.4 459.2 454.2
```

measures the treatments.

If this experiment had been repeated for two or more methods of cultivation, then the data would be represented as a four-dimensional array and there would be a total of `2 * 2 * 2 * 2` or `16` different sums that could be computed. The inclusion of a fifth factor, for example, the repetition of the experiment at a different location where the soil was different, would mean `32` marginal sums.

Once the marginal sums, or at least an appropriate selection of them depending on the experimental design, have been computed, it is relatively simple - and we must emphasize the word "relatively" - to find the necessary components of the total variation to test whatever statistical hypotheses are of interest. The construction of appropriate verbs in **J** for the calculation of any or all marginal sums in arrays of arbitrary size and dimension is a problem to which a very considerable amount of attention has been devoted. The resulting verbs have formed the basis of statistical packages of considerable utility.

Patterns

The examples that have been used so far to illustrate the **J** language have involved nothing more than the simple operations of addition, multiplication, division, and finding the larger of two numbers. However, the three examples have been obviously mathematical or statistical involving, respectively, a very simple analysis of some rainfall data, expectations in collecting a complete set of prizes, and marginal totals in rectangular arrays of experimental data.

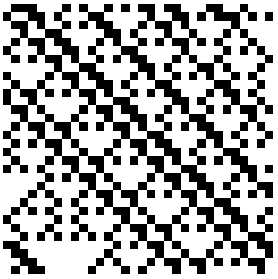
In this section we describe briefly how **J** may be applied to some problems arising in weaving - problems which might not appear to be inherently mathematical or computational. To aid the reader with no knowledge of weaving, we shall first give a few remarks about the arrangement and setting up of looms. Then we shall give an example of a simple loom set-up and the corresponding cloth diagram, and show a Windows form for experimenting with a variety of set-ups. Finally we shall give a discussion, which might be omitted on a first reading, of how **J** may be used to simplify one part of the converse problem of analyzing a given design.

Weaving is the art of forming a fabric by interlacing threads at right angles and is performed on a frame called a loom. The threads which run lengthwise in the loom are the warp threads, and the threads at right angles which are interlaced through the warp threads are the weft threads. The warp threads are alternately raised and lowered to create a shed through which the weft thread is inserted. They are fastened to a warp roller at the back of the loom, pass over a back beam and through the eyes of vertical wires called heddles, and then through reeds which help keep them parallel and in the proper sequence. The woven cloth passes over a front beam and is then wound onto a cloth roller. Each row of heddles is attached to a harness, and there may be two, four, eight or more harnesses. Each warp thread passes through a heddle on any one of the harnesses.

The harnesses are raised and lowered by treadles which are worked by the fingers on a table loom and by the feet on a foot loom. Harnesses may be tied together so that more than one may be operated by the same treadle. As a group of warp threads is raised or lowered by the treddles, the shuttle with the weft thread is inserted into the shed between the two groups of threads.

The instructions for setting up the loom with the warp threads are known as drafts. The threading draft determines the order in which the warp threads are drawn through the heddles, the tie-up draft gives the connection of the harnesses, and the treadling draft gives the order in which the treadles are used. The resulting design is known as the weave draft.

As an example we shall use the weave shown at the end of this paragraph which has been produced on a four-heddle loom which has been set up with a rosepath threading and a standard twill tie-up. So that the representations of the various drafts will be sufficiently small to be conveniently displayed on the page, we shall take only the upper left corner corresponding to the top ten weft threads and the left nine warp threads.



The threading draft is given by

```
1 0 0 0 1 0 0 0 1
0 0 0 1 0 1 0 0 0
0 0 1 0 0 0 1 0 0
0 1 0 0 0 0 0 1 0 .
```

The bottom row shows that the second and eighth warp threads pass through heddles connected to the first harness, the second row that the third and seventh threads pass through heddles connected to the second harness, etc. The tie-up draft is

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

and shows that the first treadle is connected to the first and fourth harnesses, the second treadle is connected to the third and fourth harnesses, etc. Finally, the treadling draft is

```
1 0 0 0
0 1 0 0
0 0 1 0
0 1 0 0
1 0 0 0
0 0 0 1
0 0 1 0
0 1 0 0
1 0 0 0
0 0 0 1  .
```

Reading from the first row down, we see that the treadles are used in the order first, second, third, second, etc.

Given this numerical representation of the threading, tie-up and treadling drafts, the corresponding weave draft is

```
1 1 0 0 1 0 0 1 1
1 0 0 1 1 1 0 0 1
0 0 1 1 0 1 1 0 0
1 0 0 1 1 1 0 0 1
1 1 0 0 1 0 0 1 1
0 1 1 0 0 0 1 1 0
0 0 1 1 0 1 1 0 0
1 0 0 1 1 1 0 0 1
1 1 0 0 1 0 0 1 1
0 1 1 0 0 0 1 1 0  .
```

The `1`s in this diagram indicate where the warp threads lie on top of the weft threads. Only a few simple **J** operations, which we shall omit, are required to derive this numerical representation of the weave draft from the representations of the three drafts showing the set-up of the loom.

The four drafts may be conveniently given in a rectangular arrangement known as a cloth diagram with the weave draft in the upper left, the threading draft in the lower left, the treadling draft in the upper right, and the tie-up draft in the lower right.

The above discussion, however pleasing it may be to a person interested in a numerical description of the weaving process, is of little help to the person who wishes some immediate help with weaving. It does little to answer questions such as: For given threading, treadling and tie-up drafts, what does the resulting woven cloth actually look like? What does a change in the treadling, say, have do to the appearance of the cloth? and What is the effect of

changing the colours of the yarn?

To answer such questions we have constructed a Windows form shown at the end of the paper which allows the convenient construction and display in graphic rather than numeric form of a number of different weaves for a selection of threading, tie-up and treadling drafts and colours for the warp and weft threads. Any of the resulting weaves may be stored in a file for subsequent inclusion in a document.

The converse problem of the problem we have been discussing is one of analysis rather than synthesis. For a given weave how do we set up a loom to produce cloth with this design? Stated in terms of drafts, for a given weave draft, what are the corresponding threading, treadling and tie-up drafts? Once the weave draft is known, then one must find the corresponding threading, tie-up and treadling drafts which may be recorded on the paper to obtain the cloth diagram for the given weave. Step-by-step procedures for constructing the weave draft from the cloth sample and then constructing the other drafts are given in many books on weaving. They are of necessity somewhat tedious and can be a source of error.

Although the weave draft must be found by visually examining the cloth and recording by hand the visible warp threads, the threading, tie-up and treadling drafts may be found very simply from its numerical representation. We shall discuss one aspect of the computation which is amenable to a very simple solution in **J**. The problem we shall consider is that of distinguishing identical columns in the weave draft. We shall be concerned with the computational rather than the weaving aspects of the problem, and also in showing a little of the generality of **J**.

First of all we shall look at what is known as the distribution of the unique items of a list of numbers. For a simple example we shall use the list

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

which might occur when a four-sided tetrahedral die is rolled 12 times. We see that a 3 occurred on the first, third, fifth, tenth and twelfth rolls, a 1 on the second and ninth rolls, etc. This information on the distribution of the possible outcomes may be given by the table

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

where the 1s in the first row indicate where the 3s (the face which appeared on the first roll) occurred in the twelve rolls, the 1s in the second row indicate where the 2s (the second different face to appear) occurred, etc. The row sums of the distribution table are

```
5 2 3 2
```

and give the frequency of occurrence of the faces 3, 1, 2 and 4, respectively.

If we sort the results of rolling the die, we obtain the list

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

which has the distribution table

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

from which we may obtain the frequencies

```
2 3 5 2
```

of occurrence of the faces 1, 2, 3 and 4, respectively. Given these frequencies, we may display them conveniently in the following frequency table:

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

Now suppose we wish to examine the occurrence of each of various pairs of faces which may occur when two four-sided dice are rolled a number of times. For example, if the two dice are rolled eight times, then the results could be displayed as the eight-item list

```
ÚÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÄ¿
³3 2³4 1³3 3³2 3³4 1³2 4³2 2³1 4³
ÀÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÄÙ
```

where the items are two-item lists giving the results of the various rolls. Now suppose we disregard the order in which the faces occur on each roll, so that, for example, a 3 followed by a 2 will be considered the same as a 2 followed by a 3. Then we may sort each of the items in order and obtain a second eight-item list

```
ÚÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÄ¿
³2 3³1 4³3 3³2 3³1 4³2 4³2 2³1 4³
ÀÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÄÙ
```

The distribution table for this list is

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

whose items give the locations of the various combinations which occur regardless of order. For example, the second row shows that a `1` and a `4` were obtained in some order on the second, fifth and eight rolls.

Now let us return to our weaving example and consider finding, as noted above, identical columns in a given weave draft. Once the distribution of columns has been found, the threading, tie-up and treadling drafts may be found very simply. We shall, however, not consider this last part of the computation. Therefore, we wish to find the distribution of columns showing the visible warp threads in the weave draft in a form which may be conveniently used to give the other drafts in the cloth diagram.

To simplify the discussion we shall consider only the first four rows of the weave draft which are

```
1 1 0 0 1 0 0 1 1
1 0 0 1 1 1 0 0 1
0 0 1 1 0 1 1 0 0
1 0 0 1 1 1 0 0 1  .
```

The problem, then, is to determine that the columns may be divided into the following groups of identical columns: first, fifth and ninth; second and eighth; third and seventh; and fourth and sixth.

From our previous discussion of distribution tables corresponding to lists it should be apparent that the problem is equivalent to finding the distribution table of a list whose items are the columns of the weave draft. This may be done in the following three steps: Firstly transpose the rows and columns of the weave draft so that the columns become the rows and vice versa; secondly construct a list whose items are the rows of the transposed weave draft; and thirdly find the distribution table of this list.

If we return to our simple example, we have the weave draft

```
1 1 0 0 1 0 0 1 1
1 0 0 1 1 1 0 0 1
0 0 1 1 0 1 1 0 0
1 0 0 1 1 1 0 0 1  .
```

Transposing rows and columns we have

```
1 1 0 1
1 0 0 0
0 0 1 0
0 1 1 1
1 1 0 1
0 1 1 1
0 0 1 0
1 0 0 0
1 1 0 1  ,
```

from which we obtain by "boxing" the rows the list

```
ÚÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂ
³1 1 0 1³1 0 0 0³0 0 1 0³0 1 1 1³1 1 0 1³
ÀÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁ
    ÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄ¿
    0 1 1 1³0 0 1 0³1 0 0 0³1 1 0 1³
    ÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÙ   ,
```

and finally the required distribution table

```
1 0 0 0 1 0 0 0 1
0 1 0 0 0 0 0 1 0
0 0 1 0 0 0 1 0 0
0 0 0 1 0 1 0 0 0 .
```

An examination of this table shows that it gives the required distribution of the different warp columns in the weave draft.

To conclude this discussion we shall give a **J** verb for performing the above calculations. We introduce the three verbs `|:` *transpose*, `<` *box*, and `=` *self-classify* which gives the distribution table. We should note each of the three steps required in finding the column distribution is accomplished by a single **J** verb. The verb for doing all three steps in sequence may be written as

```
cdis=: = @ (<"1 @ |:)  ,
```

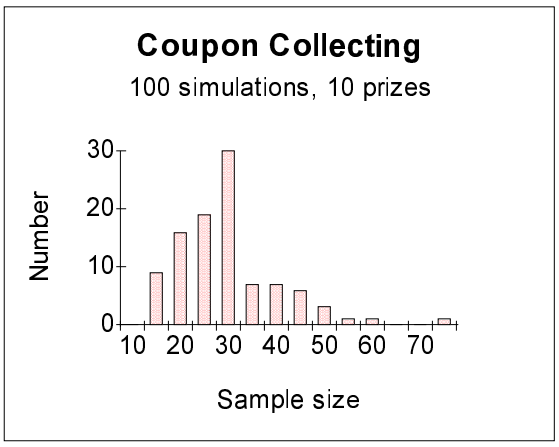
where `@` is the conjunction *atop* (or informally *after*), and read as "distribution after row box after transpose". If the weave draft is represent by `W`, say, then the distribution table of the warp threads is given by `cdis W`.

Pictures

" 'And what is the use of a book,' thought Alice, 'without pictures or conversation?' " What use indeed? Since "pictures", i.e., graphs of some kind or other figures, often tell us much more than the tables of numbers from which they were derived, let us have a brief look at how we might use graphics in some of our numerical examples.

"And what is the use of a book," thought Alice, "without pictures or conversation?"

First of all, let us have another look at the coupon collector's problem and give the bar chart for the frequency table of a previous section giving the sample sizes for 100 simulations. Bar charts and other graphics may be constructed using the graphing facilities of either **J** or of some spreadsheet or graphics package. The chart below, which was produced by MS Works, shows the distribution of sample sizes much more clearly than did the table. Of course, either the table or the original data are required to calculate the average sample size which was almost identical to the average sample size predicted by theory.



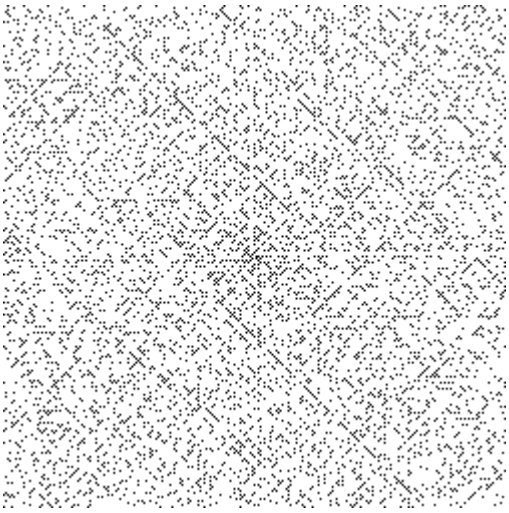
A very interesting problem concerning prime numbers was reported by Martin Gardner in the chapter "Patterns and primes" in *Martin Gardner's 6th Book of Mathematical Diversions from Scientific American* (The University of Chicago Press, 1971). It was observed by a mathematician from the Los Alamos Scientific Laboratory who was idly doodling during what he termed a "long and very boring paper" that if the positive integers were first arranged in a rectangular spiral then the prime numbers tended to lie along diagonal straight lines. For example, the left array in the table below shows a spiral arrangement of the integers from `1` to `81`, inclusive, while the right array is an equivalent representation of the same array with the location of the primes `2`, `3`, `5`, `7`, ... indicated by `1`s and the location of the non-primes by `0`s. It may be seen even with such a small arrangement that there appears to be a tendency for the primes to be located along diagonal lines.

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
³73 74 75 76 77 78 79 80 81³1 0 0 0 0 0 1 0 0³
³72 43 44 45 46 47 48 49 50³0 1 0 0 0 1 0 0 0³
³71 42 21 22 23 24 25 26 51³1 0 0 0 1 0 0 0 0³
³70 41 20  7  8  9 10 27 52³0 1 0 1 0 0 0 0 0³
³69 40 19  6  1  2 11 28 53³0 0 1 0 0 1 1 0 1³
³68 39 18  5  4  3 12 29 54³0 0 0 1 0 1 0 1 0³
³67 38 17 16 15 14 13 30 55³1 0 1 0 0 0 1 0 0³
³66 37 36 35 34 33 32 31 56³0 1 0 0 0 0 0 1 0³
³65 64 63 62 61 60 59 58 57³0 0 0 0 1 0 1 0 0³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

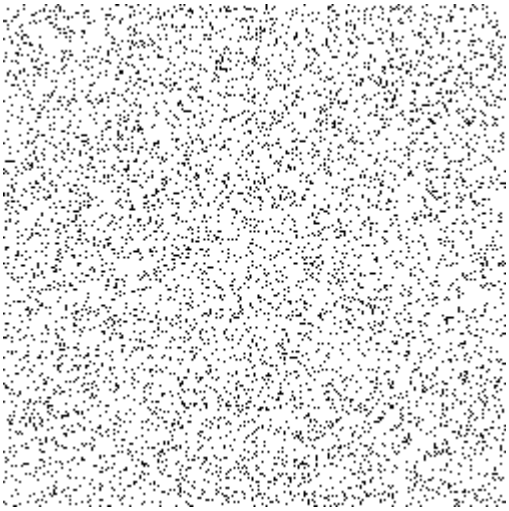
The problem was investigated further using the computer at Los Alamos and a table of the first 90 million prime numbers stored on magnetic tape. Images were produced for various spiral arrangements of consecutive integers including the integers from 1 to about 10 000 and from 1 to about 65 000. This experimental work reinforced the conjecture that prime numbers behaved in this manner when the consecutive integers were arranged in a spiral. Some years ago a visitor to the School of Mathematics at the University of Newcastle-upon-Tyne donated a large painting of this spiral arrangement of primes as an expression of appreciation for his visit.

The first figure on the next page, which was constructed very simply in **J**, gives the image produced by a spiral arrangement with `251` rows and `251` columns of the integers from `1` to `63001`. There are `6320` primes represented in the above array. The second figure gives the image produced for a random rather than a spiral arrangement of integers with the primes marked as in the previous figure.

Some years ago I tiled a coffee table with 16 rows and 43 columns of one-inch tiles in a pattern corresponding to a spiral arrangement of the integers from 1 to 688. The tiles corresponding to primes were of one colour and those corresponding to non-primes were another colour except the tile for the integer 1 which was a third colour. (I can remember buying sufficient tiles for the 124 prime squares and the 564 non-prime squares, and also legitimately getting the single tile to represent the integer 1 without paying for it.) The pattern is shown in the first figure of the Appendix at the end of the paper.



About 30 years ago the Cambridge mathematician John Conway was investigating the



concept of a universal constructor which is a hypothetical machine which could reproduce itself. In doing so he devised a simple game, which he called the Game of Life, that could be played either with counters on a large sheet of squared paper or simulated on a computer. Martin Gardner devoted two columns in *Scientific American* (October 1970 and February 1971) to this game and it immediately became immensely popular with persons in the computing profession who altogether spent probably thousands and thousands of hours of (often ill-gotten) computer time playing the game. I wrote Life programs in APL and Nial and probably in FORTRAN or BASIC, and can attest to the lure of the game.

The basics of the game may be described very simply by imagining a number of counters being placed in some pattern on a sheet of squared paper. Now each square, whether occupied by a counter or empty, has eight neighbouring cells - left, right, above, below, and the four diagonals. (For simplicity we shall for the moment ignore the neighbours of squares at the borders of the sheet although how they are defined can influence the results of any game.) Conway devised a set of rules which governed how a configuration would change from one generation to the next. According to his rules an occupied centre square remains occupied in the next generation if two or three neighbouring squares are occupied. If a vacant centre square has exactly three occupied neighbouring squares, it will be occupied in the next generation. In all other circumstances, a centre square, whether occupied or not, will be empty in the next generation. Each square on the sheet is examined in this manner and its condition in the next generation is noted. Then, after all squares have been considered, all necessary changes for the next generation are made at the same time giving the configuration in the next generation. This process is repeated for successive generations and changes in the pattern of occupied squares is observed.

The appeal of the game for many persons is the unpredictability of successive generations of configurations for arbitrary initial configurations. A very large amount of experimental work has shown that regardless of the initial configuration - and some of these lead to some very interesting sequences of patterns - after a number of generations either all counters are removed from the sheet or an unchanging configuration or one that cycles through a number of identical configurations is obtained.

There is a vast literature on the Game of Life and there are sites on the World Wide Web devoted to the game. Persons wishing further information might consult the last three chapters of Martin Gardner's *Wheels, Life and Other Mathematical Amusements* (W. H. Freeman and Company, 1983).

A much simpler version of Life may be played in one dimension with a strip of squared paper (or the computer equivalent) with some of the squares occupied with counters and the remaining ones unoccupied. The neighbourhood of each square

consists of the square itself and the squares to the immediate right and left with the missing neighbours of the two end squares being the squares at the opposite ends. Given an appropriate rule for going from one generation to the next, the configurations for an arbitrary number of generations may be displayed one below each other in a rectangular array.

As an example consider a one-dimensional game in which there is a random initial configuration with a strip of arbitrary length. The rule for going from one generation to the next is the so-called parity rule in which each centre square takes a value in the next generation so that the total number of occupied squares in the neighbourhood including the centre square is either zero or two. If the initial configuration for a strip with fifteen squares is

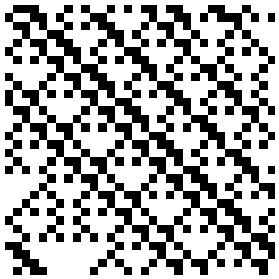
```
ÚÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄ¿
³1³1³0³1³1³1³1³0³0³1³0³0³0³1³1³
ÀÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÙ
```

then the following array gives the initial configuration and those for the next five generations:

```
ÚÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄÂÄ¿
³1³1³0³1³1³1³1³0³0³1³0³0³0³1³1³
ÃÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄ´
³0³1³0³0³0³0³1³1³1³0³1³0³1³0³0³
ÃÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄ´
³1³0³1³0³0³1³0³0³1³0³0³0³0³1³0³
ÃÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄ´
³0³0³0³1³1³0³1³1³0³1³0³0³1³0³0³
ÃÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄ´
³0³0³1³0³1³0³0³1³0³0³1³1³0³1³0³
ÃÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄÅÄ´
³0³1³0³0³0³1³1³0³1³1³0³1³0³0³1³
ÀÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÁÄÙ
```

The figure following the next paragraph shows 64 generations of this one-dimensional game for a random initial configuration of 64 squares.

We might well ask about the value of the examples discussed in this section. Are they anything more than idle games that keep us from engaging in more productive work? There should be no question about the value of bar charts and similar charts. The financial section of the daily paper is filled with various charts - straight line, bar, pie, etc. - which are much easier to understand than the tables of figures from which they were derived. Even our monthly utility bill now has a bar chart showing the actual or estimated monthly consumption for each month during the last year. The prime spiral may be more difficult to justify on purely utilitarian grounds but we should not be too hasty to criticize the study of any aspect of something as fundamental as prime numbers. We might keep in mind that public key encryption of data transmitted by computer relies on the practical difficulties of the prime factoring of very large numbers. (And even if prime spirals never have a practical application, the time spent may be justified on the pleasure to be derived from their study and the beauty of the patterns.) Processes similar to the Game of Life have been extensively studied under the name of cellular automata and have had important applications in solid-state physics, to mention only one example. One-dimensional automata are currently being studied at the Santa Fe Institute in New Mexico as models of very simple biological structures.



A more formal name for creating images from numerical data is “visualization”. Important and very commonly occurring applications include CAT scans and MRIs in medicine, weather maps in meteorology, and the representation of numerical models in mathematics. A beautiful presentation, both mathematically and visually, of this subject has been given by Clifford Reiter in *Fractals Visualization and J* (Iverson Software Inc., 1995). Some of the material in this book was used to derive the images in this section.

Christmas trees

Not all calculations performed in **J**, or in any other language, need have a serious purpose. As a simple example, we shall see how large factorials such as those considered in a previous section may be represented as trees which may be used, for example, in Christmas cards.

To see the connection between the factorial function and Christmas trees we shall turn to an interesting article by Martin Gardner, "Factorial

Oddities" published some years ago in *Scientific American*, and republished in his *Mathematical Magic Show* (Vintage Books, 1978). In this article a tree factorial is defined as a factorial whose digits may be arranged in a triangle with one digit in the first row, three in the second, five in the third, etc. The idea originated with the thought that such a display of a large factorial resembled a Christmas tree and could be used on a Christmas card. Gardner's article gives the twenty tree factorials between seven and one thousand, and displays in triangular form the factorials for 105, and for 508 which has 1156 digits.

We may note that the number of digits in a one-level tree factorial is 1, the number in a two-level tree factorial is 1 + 3 or 4, the number in a three-level tree factorial is 1 + 3 + 5 or 9, etc. In general, the total number of digits in any tree factorial is a perfect square.

With extended integer arithmetic in **J** it is a simple matter to find these tree factorials and list them in triangular form. We have seen that

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

has the value

```
   1 1 2 6 24 120 720 5040 40320  .
```

Each of the first four items is trivially a tree factorial with a single digit, and `!7` or `5040` is a tree factorial which may be displayed as

```
    5
   040 .
```

The next two tree factorials are `!12x` or

```
     4
    790
   01600
```

and `!18x` or

```
      6
     402
    37370
   5728000 .
```

The tree factorial `!105x` with `169` digits is given below:

```
                1
               081
              39675
             8240290
            900504101
           30580032964
          9720646107774
         902579144176636
        57322653190990515
      3326984536526808240
    339776398934872029657
  9938729078134368160972 8
000000000000000000000000 0
```

The **J** calculations to locate tree factorials and display them in triangular form are relatively simple and will not be given here.

Gardner also displays the hexagonal factorial `!477x` with `1073` digits and the octagonal factorial `!2206x` with `6421` digits. To give just one result involving a factorial larger than one given in Gardner we have used **J** to find that `!2248x` is a tree factorial with `6561` digits.

Taking chances

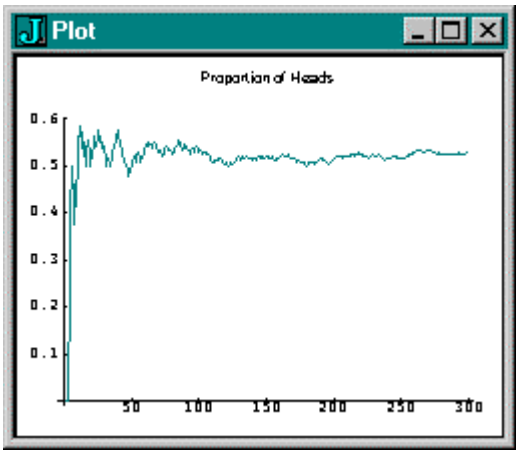
One of the first books I bought when I came to the University of Alberta was a just-published little paperback *Lady Luck: The Theory of Probability* by Warren Weaver (Doubleday, 1963). The author who was in his late 60s when the book was published had had a varied and distinguished career first as a mathematician in academia and later as a scientific administrator in a number of organizations. Amongst his many publications is *Alice in Many Tongues* (The University of Wisconsin Press, 1964), a delightful discussion of the life of Lewis Carroll, the writing of the Alice books, and their translation into over forty languages. *Lady Luck* gives an elementary and very readable introduction to the origins and development of probability theory and the statistics of chance. Weaver does not avoid mathematical notation but introduces and uses it in a manner which enhances rather than detracts from the exposition. *Lady Luck* is a gem of a book which I have used with great pleasure ever since I acquired my first copy. It deserves to remain in print for a long, long time.

In this section we shall use **J** as a "companion" to *Lady Luck* just as we did earlier with the book by Lanczos. We shall perform some of the calculations required to describe and simulate a few of the examples that Weaver discusses so delightfully. All of the examples we shall discuss may be very easily handled in **J**. At the end of the section we shall make a few remarks for the interested reader about the simulation of random numbers in **J** and give the **J**

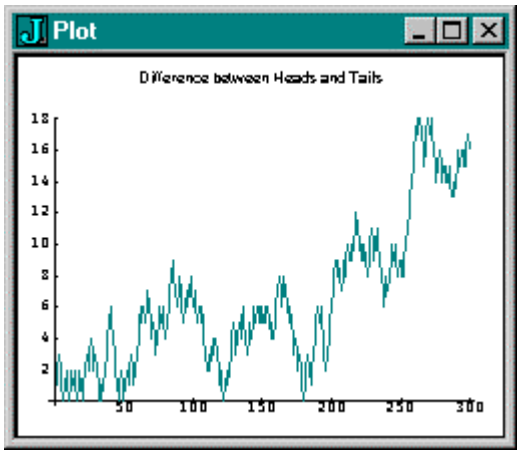
expressions required to obtain a few of the results discussed here.

Of course, all of Weaver's simulations had to be performed using tables of random numbers. He mentions the RAND Corporation's *A Million Random Digits with 100 000 Normal Deviates* published in 1955, and, I believe, reviewed with some disbelief in *The New York Times*.

One simple example which is invariably used in intoducing probability is the tossing of an unbiased coin and the calculation of the ratio of the number of heads to the number of tosses. The convergence of this ratio to the theoretical probability of 0.5 for an unbiased coin giving heads on a single toss may be shown very simply as illustrated by the following graph showing the results of a simulation in **J** of 300 tosses:
However, as Weaver points out, although the ratio of heads to the total number of tosses approaches 0.5 in such experiments, the difference between the numbers of heads and tails tends to grow. The graph in the next column shows for the same simulation the absolute value of the difference between the number of heads and the number of tails.



The difference between the long-term convergence of the ratio of the number of heads and the increase in the discrepancy of the numbers of heads and tails has an interesting application to the martingale gambling system in which one doubles one's bet after every loss and returns to the original bet after every win. For example, suppose one is



betting that a toss of a coin will result in heads and that one starts with an initial bet of one dollar. According to the martingale system, one bets a dollar on each successive win and increases the bet to two dollars on the first loss. If the next toss gives a head, then one recoups the loss and wins a dollar, and then reduces the bet to one dollar. However, if the toss gives a tail, then the bet is increased to four dollars so that on the following toss one either recoups the loss of three dollars and gains a dollar in the event of a win, or raises the bet to eight dollars in the event of a loss. One continues in this manner doubling one's bets for successive losses until a win occurs. This system allows one to recoup one's losses however large, provided one has the resources and the nerve to sustain a run of losses of arbitrary length. The only problem is that such a run of bad luck indicated by a long sequence of tails becomes increasingly common in spite of the convergence of the ratio of the number of heads to the total number of tosses and could easily cause one's ruin.

Weaver uses dice examples on numerous occasions and gives a nice discussion of the origins of probability theory in Chevalier de Méré's dice problem about which he consulted the philosopher Blaise Pascal in 1654. Indeed, there is almost one entire column of entries under "Dice" in the Index. He makes the following statement in Chapter 2 regarding dice: "In case this book falls into completely innocent hands (the probability of which is small) I should perhaps state that a die is a small cube with slightly rounded corners and edges, presumably exactly symmetrical in shape and uniform in material (i.e., not 'loaded'), whose faces bear the numbers 1, 2, 3, 4, 5, and 6. When it is

'thrown' or 'rolled' or 'tossed', a die should be as likely to come to rest on one side as on any other. And therefore a die is as likely to 'show', on its uppermost face, any one of the six numbers as any other one." A more succinct definition of an unbiased die would be difficult to find. Alas, it is not strictly true as sets of dice may be purchased, presumably for use in statistics courses, with four, six, eight, twelve and twenty faces corresponding to the five regular polyhedra. Of course, the rolling of "dice" with any number of non-negative integer sides may be easily simulated in **J** or in any programming language.

Now we shall consider rolling two unbiased dice and recording the sum of the two numbers which appear on the two faces. We are interested in the range of values of the sum and the corresponding probabilities and in comparing the frequencies of the simulation of a sequence of rolls of the two dice with the expected values. The following gives the results of the use of **J** to investigate these questions. The necessary **J** expressions are given at the end of this section for any interested reader.

```
Die faces
   1 2 3 4 5 6

Total number of combinations
   36

Combinations of faces
   ÚÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄ¿
   ³1 1³1 2³1 3³1 4³1 5³1 6³
   ÃÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄ´
   ³2 1³2 2³2 3³2 4³2 5³2 6³
   ÃÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄ´
   ³3 1³3 2³3 3³3 4³3 5³3 6³
   ÃÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄ´
   ³4 1³4 2³4 3³4 4³4 5³4 6³
   ÃÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄ´
   ³5 1³5 2³5 3³5 4³5 5³5 6³
   ÃÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄ´
   ³6 1³6 2³6 3³6 4³6 5³6 6³
   ÀÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÙ

Sums for two rolls
   ÚÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄ¿
   ³ ³1 2 3  4  5  6³
   ÃÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄÅÄÄÄ´
   ³1³2 3 4  5  6  7³
   ³2³3 4 5  6  7  8³
   ³3³4 5 6  7  8  9³
   ³4³5 6 7  8  9 10³
   ³5³6 7 8  9 10 11³
   ³6³7 8 9 10 11 12³
   ÀÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÙ

Grouped sums
   ÚÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂÄÄÄÂ
   ³2³3 3³4 4 4³5 5 5 5³6 6 6 6 6³
   ÀÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁÄÄÄÁ
      ÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÂ
      7 7 7 7 7 7³8 8 8 8 8³9 9 9 9³
      ÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÁ
      ÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÂÄÄ¿
      10 10 10³11 11³12³
      ÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÙ

Sums
   2 3 4 5 6 7 8 9 10 11 12

Frequencies for sums
   1 2 3 4 5 6 5 4 3 2 1

Probabilities (rounded)
   0.02778 0.05556 0.08333 0.11111
      0.13889 0.16667 0.13889 0.11111
      0.08333 0.05556 0.02778

Observed frequencies for 1000 rolls
   23 47 84 117 147 171 130 112 91 59 19

Expected frequencies (rounded)
   28 56 83 111 139 167 139 111 83 56 28
```

Before the use of computers or published tables of random numbers persons wishing to use examples of random events such as coin tossing or dice throwing had to use data obtained from actual experiments. I can remember as a graduate student spending an afternoon tossing a coin and plotting on squared paper the cumulative ratio of the number of heads. Well-known examples - at least well-known to statisticians - are the dice-throwing experiments of the English biologist W. F. R. Weldon (1860 - 1906) which he performed to illustrate his statistical arguments. In one he tabulated the results of rolling twelve dice 4096 times counting as a success the occurrence of 4, 5 or 6. Another set of dice data which is apparently not as well known as Weldon's was generated by a Swiss scientist Rudolf Wolf who rolled a die 100 000 times and obtained the results 16 632, 17 700, 15 183, 14 393, 17 707 and 18 385 for the frequencies of occurrence of 1, 2, 3, 4, 5 and 6, respectively.

"In case this book falls into completely innocent hands (the probability of which is small) I should perhaps state that a die is a small cube with slightly rounded corners and edges, presumably exactly symmetrical in shape and uniform in material (i.e., not 'loaded'), whose faces bear the numbers 1, 2, 3, 4, 5, and 6. ..."

Warren Weaver

One very common gambling game played with dice is the game of craps. In this game a player rolls two dice. The player wins if the sum of the two faces is either 7 or 11, and loses if the sum is 2, 3 or 12. If any other sum occurs, the player continues to roll until either the sum which appeared on the first roll occurs which is a win, or a 7 occurs which is a loss. For example, 9 11 8 5 8 9, 8 9 8 and 11 represent wins, and 10 9 9 2 9 4 4 9 12 5 7, 3 and 6 3 9 7 represent losses. The probability that the player will win in a single game may be calculated to be 0.4929 so that it is favourable in the long run to bet against the player.

"If you keep on gambling, you will lose" is how Weaver introduces the problem of the Gambler's Ruin which dates from the seventeenth century. In this problem a player with a fixed amount of money plays against another person or a gambling house also with a fixed amount of money until either the player or the house is ruined. We shall assume that the amount bet at each play is one dollar and the probability of success on a single play remains constant throughout the game. We are interested in finding the probability of the ultimate ruin of the player for arbitrary initial amounts of capital for the two participants. Here we shall give the results of a few calculations which may be very simply done in **J** and leave a few remarks about the algebraic expressions used until the end of the section.

If the game is fair, i.e., if the probability of a win on a single game is 0.5, then the probability of the player's ultimate ruin is equal to the ratio of the house's initial capital to the total amount of capital involved. Therefore, the more money the house has in relation to the player, the greater will be the player's chances of ruin. The following table shows the probabilities where the player always has a fixed amount of `10` dollars and the house has `10`, `50`, `100` or `1000` dollars:

```
10      10   0.500
10      50   0.833
10     100   0.909
10    1000   0.990
```

We see that even with a fair game the player is easily ruined when playing against an opponent with significantly more capital.

If the game is unfair to the player, i.e., if the probability of the player winning in a single game is less than 0.5, then the probabilities of the player's ultimate ruin change significantly. If the game is only slightly unfair to the player, as in the game of craps with a probability of 0.4929 of winning in a single game, the probabilities for the same fixed amounts as above become

```
10    10   0.570
10    50   0.927
10   100   0.985
10  1000   1.000  .
```

If the player's probability of success decreases to 0.48, then the chances of ruin become the following:

```
10    10     0.690
10    50     0.990
10   100     1.000
10  1000     1.000
```

Now let us apply these ideas to the very popular Lotto 6-49. In this lottery a person purchases a ticket for one dollar and selects six integers at random from the integers between 1 and 49, inclusive. The payout in the weekly draw depends on the number of matches and is zero for 2 or fewer matches and 10, 75, 2500 or 100000 dollars for 3, 4, 5 or 6 matches, respectively. We shall find the probabilities for each of the prizes, assess the game in light of the gambler's ruin, and finally simulate the game for an arbitrary number of purchases and compute the cumulative winnings. All of these calculations may be done quite simply in **J**, but we shall omit any discussion of the details.

It may be shown that the probabilities for `0`, `1`, `2`, ... matches rounded to `5` decimal places are equal to

```
0.43596 0.41302 0.13238 0.01765
          0.00097 0.00002 0.00000
```

Therefore, the probability of not winning anything in a single game is the sum of the first three probabilities or `0.98136`, and the probability of winning something is `0.01864`. We hardly need to appeal to the Gambler's Ruin problem to deduce that the player will be certainly ruined for any reasonable distribution of capital between the player and the house.

To look at the disastrous nature of Lotto 6-49 another way, we performed a simulation in which a person purchased two one-dollar tickets each week for twenty-five years. The following list gives the cumulative annual winnings for the simulation:

```
10 20 40 60 80 110 110 160 170 190
   275 295 315 315 335 355 365 385
   385 415 425 435 495 505 505
```

In other words a total investment over the twenty-five years of `104 * 25` or `2600` dollars gave a return of only `505` dollars.

Lest the reader believe that all random processes are related to the tossing of coins, the rolling of dice or other gambling schemes that can easily lead to the unwary player's financial ruin, we shall conclude with two examples of geometric probabilities involving the fundamental constant π, the ratio of the circumference of a circle to its diameter, which is 3.14159 to five decimal places. One method was given by the eighteenth-century French naturalist and theoretical biologist the Comte de Buffon. A rod, i.e., a match or a needle, of length *c* is thrown at random on a plane surface ruled with parallel lines a distance *a* apart, where *a* is greater than *c*. It may be shown with a little mathematics which we shall ignore that the probability that the rod will cross one of the lines is $2c/\pi a$. Therefore, if we throw the rod a large number of times *N* and observe the number of times *m* it crosses a line, an estimate of π is given by $2cN/am$. Weaver, in a very interesting discussion of Buffon's problem, mentions that "a Professor Wolf of Frankfurt" threw a needle 5000 times and obtained an approximation to π of 3.1596. Then there was "a Captain Fox" who obtained a value of 3.1419 with 1120 trials, and finally "a mathematician named Lazzarini" who with 3408 trials obtained a value of 3.14159265 which differs from the true value in the seventh decimal place. Weaver remarks that there is "something a little fishy" with this last result, possibly because the number of trials might indicate that the experiment may have been stopped only because the estimate of π was so accurate.

There is another random method of estimating π which is more intuitive than Buffon's method. Consider a quadrant of a unit circle, i.e., a circle with a radius of 1, inscribed in a unit square. The area of the square is 1, and, since the area of a circle of radius *r* is is πr^2, the area of the quadrant is $\pi/4$. Thus if we repeatedly select points at random in the square and record how many lie within the quadrant, we may obtain an estimate of π as we did with Buffon's problem. Making use of some simple **J** verbs, we did five simulations each of `10000` repetitions and obtained `3.1364`, `3.14`, `3.146`, `3.1328` and `3.1572` for estimates of π. There is certainly nothing "fishy" with these results.

Random numbers may be generated very simply in **J** with the verb *roll* `?`. For example, `?6` gives an integer picked at random from the first six non-negative integers. Since `>:` is the verb *increment* which adds `1` to its argument which we have used previously in the coupon collector's problem, `>:?6` gives an integer from the first six positive integers and could represent the simulation of rolling an unbiased die once. Since `?6 6` gives two integers picked at random from the first six non-negative integers with repetitions allowed, the expression `+/>:?6 6` may be used to simulate rolling two unbiased dice and finding the sum of the two numbers which occur.

The expression

```
?10 10 10 10 10
```

gives a list of `5` non-negative random integers between `0` and `9`, and could have a value such as `5 8 0 8 1` in which digits may be repeated. Thus,

```
>:?10 10 10 10 10
```

could represent a simulation of the coupon collector's problem for `5` purchases where there are `10` prizes since duplication of prizes is not only allowed but expected. Such a process is known as sampling with replacement. If we are interested in random sampling without replacement, then we may use the verb *deal* also represented by `?`. For example, `5?10` gives a list of `5` integers chosen without duplication from the first `10` non-negative integers, and could have the value `5 6 7 0 9` or

`0 4 8 9 5`, for example, in which the items are all different. As another example, the expression `13?52` could represent dealing `13` cards from a shuffled deck, and could have the value

```
   40 14 43 44 34 46 26 49 12 2 31 33 27
```

if some appropriate interpretation is given to the integers `0`, `1`, `2`, ..., `51`.

Random numbers uniformly distributed over an arbitrary continuous interval may be easily obtained in **J**. For example, the expression

```
   (?999999999) % 1000000000
```

gives numbers uniformly distributed between `0` and `1`, and three repetitions gave the values `0.569399`, `0.883811` and `0.211794`. A **J** verb may be easily constructed for conveniently producing such random numbers, and could be used, for example, in estimating the the value of π by simulating the choosing of points in a square as discussed earlier in the section.

We shall now give the **J** expressions used to construct the examples for the discussion of the rolling of two dice given earlier. A study of these expressions might be deferred until the material in the next section is covered and some familiarity is obtained with the ***J*** *Introduction and Dictionary* which is referred to there. Alternatively, since the rest of this paper is independent of any understanding of this material, it may be omitted altogether.

```
NB. Die faces
   Faces=: 1 2 3 4 5 6
NB. Total number of combinations
   *: #Faces
NB. Combinations of faces
   {Faces;Faces
NB. Sums for two rolls
   + table Faces
NB. Grouped sums
   </. +/~Faces
NB. Sums
   Sums=: ~.,+/~Faces
NB. Frequencies for sums
   #/. +/~Faces
NB. Probabilities (rounded)
   8.5": Prob=: 36 %~ #/. +/~Faces
NB. Probability table
   Sums,.Prob
NB. Obs. frequencies for 1000 rolls
   +/"1 Sums=/+/ >:?2 1000$6
NB. Expected frequencies (rounded)
   ". 5.0": 1000*Prob
```

Finally we shall give algebraic expressions in conventional notation for the Gambler's Ruin. We shall suppose that the player has a capital of d dollars and the house a capital of D dollars, and that the probability of success in a single game is p. If the game is fair to both parties, i.e., if p is 0.5, then it may be shown that the probability of the gambler's ruin is equal to $D/(d + D)$.
If the game is not fair, i.e., if p is not equal to 0.5, then the expression for the gambler's ruin may be shown to be

$$((q/p)^C - (q/p)^d) / ((q/p)^C - 1))\ ,$$

where $C = d + D$ and $q = 1 - p$.

Part Six. Another Look at J

Notation

In this part we shall have another look at **J** and in so doing review and expand upon some of the material we have already introduced and also introduce some new topics. This section, and indeed all of Part Six, might be omitted on a first reading of the paper or it might be omitted altogether by those who do not wish to continue a study of **J** beyond what has been covered so far. We shall begin with some remarks on mathematical notation and illustrate by several examples why a suitable notation such as that provided by **J** is so important for the clear presentation of mathematical and statistical concepts. Indeed, first APL and then **J** as concise, unambiguous and suggestive executable notations have been the main themes in Kenneth Iverson's long career in the development and promotion of these languages.

Persons wishing to continue their study of **J** should consult ***J*** *Introduction and Dictionary* (Iverson Software Inc., 1998) which gives a complete description of the language. It is an indispensible reference in learning and using **J**.

"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race."

A. N. Whitehead (1861 - 1947)

In my opinion one of the best statements of the importance of mathematical notation has been given by A. N. Whitehead in his little book for the general reader *An Introduction to Mathematics* (Oxford, 1972) which was first published in 1911:

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race ... It is a profoundly erroneous truism, repeated by all copy-books and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them.

Although this was written half a century before the first implementation of APL, I have always considered it to be one of the best justifications that could be given for including a language such as **J** in one's repertoire of programming languages and application packages.

The examples in this section all make use of the adverb represented by the symbol `/` which, depending how the verb derived from it is used, is called either *insert* or *table*. We have seen the use of *insert* in an earlier section, and *table* has been used implicitly in the defined adverb `table` which also appears in an earlier section. In these examples we shall introduce some **J** expressions which although very simple are of considerable generality and illustrate the important role a suitable notation can play in expressing numerical calculations. In addition, the examples will give us the opportunity to review the influence of verbs, adverbs and conjunctions on the order of evaluation of **J** expressions.

We introduced the adverb *insert* by the example

```
+/ 4 6 8 5 2
```

which gives the sum `25` of the numbers in the list. Furthermore `/` is called an adverb because it modifies the verb given to its left, in this instance the verb `+`, so that the new verb `+/` inserts the verb `+` between the items of its argument. Thus

```
+/ 4 6 8 5 2
```

is equivalent to

```
4 + 6 + 8 + 5 + 2 .
```

Similarly,

```
*/ 4 6 8 5 2
```

which is equal to `1920` is equivalent to

```
4 * 6 * 8 * 5 * 2 ,
```

the product of the items in the list. Thus `+/` and `*/` may be considered analogous to the conventional mathematical symbols Σ and Π, respectively.

Now let us define the list

```
p=: 1 2 3 4 5
```

of the first `5` postive integers. Therefore, the expression `+/p` is the sum of the first `5` positive integers and is equal to `15`, and `*/p 5` is their product `120` which may also be expressed as `!5` as we saw earlier. However, the value of `-/p 5` is `3` which may be puzzling. To understand this result let us review the precendence rules for verbs which we introduced earlier but have not emphasized.

The precedence of verbs is determined by parentheses, and in their absence the right argument is the entire expression on the right and the left argument is the noun immediately on the left. Therefore

```
2 * 3 + 4
```

is `14` since the addition is performed before the multiplication. If the multiplication is to be performed first to give the result `10`, then the expression may be written as

```
(2 * 3) + 4 ,
```

or more simply without parentheses as

```
4 + 2 * 3 .
```

Now the step-by-step evaluation of the expression `-/p` is as follows:

```
-/p
-/1 2 3 4 5
1 - 2 - 3 - 4 - 5
```

```
1 - 2 - 3 - _1
1 - 2 - 3 + 1
1 - 2 - 4
1 - _2
1 + 2
3
```

The value of this expression, which is known as an alternating sum, is given as the sum of the alternate items in the list of the first `5` positive integers beginning with the first item minus the sum of the alternate items beginning with the second item, i.e.,

```
(1 + 3 + 5) - (2 + 4)   .
```

For another example of the insert adverb we shall introduce the verb *not-equal* `~:` which gives a result of `1` if its arguments are unequal and `0` otherwise, so that, for example, `4 ~: 8` is `1` and `5 ~: 5` is `0`. If we proceed as we did with the expression for the alternating sum in the last paragraph, we would see that an expression such as

```
~:/0 1 0 1 0 1 1
```

where the right argument has an even number of `1`s with the remander of the items `0`s has the value `1`, and an expression such as

```
~:/ 0 1 1 0 1 1 0 0 0 1
```

with an odd number of `1`s has the value `0`. In general, the verb `~:/` with a list argument whose items are `0`s or `1`s gives a parity check since the result is `1` for an even number of `1`s and `0` for an odd number of `1`s.

The insert adverb may also be applied to arrays of more than one dimension. For example, if `T1` is the table

```
62.3 58.5 44.6 50.3
63.4 50.4   45 46.7
64.5 46.1 62.6 50.3
```

of yields for the first variety of oats used in a previous example, then `+/T1` gives the column sums

```
190.2 155 152.2 147.3
```

for the four replications, and `+/"1 T1`, where `"` is the conjunction *rank*, gives the row sums

```
215.7 205.5 223.5
```

for the four treatments.

Before we consider the adverb *table* we might review our earlier comments about the number of arguments, or valence, of verbs, and then consider the valence of adverbs and conjunctions. The symbols for most primitive verbs are ambivalent in that they may represent either a monadic verb with a single argument on the right or a dyadic verb with one argument on the right and a second argument on the left. For example, `%` represents the monadic verb *reciprocal* and the dyadic verb *divided by*, and `% 2.5` is `0.4` and `15 % 6` is `2.5`. Furthermore, as mentioned earlier, both monadic and dyadic verbs may occur in the same expression, and thus the expression `% 15 % 6` is "the reciprocal of (`15` divided by `6`)" or `0.4` rather than "(the reciprocal of `15`) divided by `6`". The valence of any symbol in an expression and thus the function it represents should be apparent from the context. As another example, `-` represents the monadic verb *negate* and the dyadic verb *minus*, and `-4` is `_4` and `3 - 5` is `_2` and we note the use of the underbar `_` to represent negative numbers.

Unlike verbs, adverbs are monadic with a single verb argument on the left as in the expressions `+/` and `*/`, but produce derived verbs which may be used either monadically or dyadically. Thus the adverb `/` is named either *insert* or *table* depending upon whether the derived verb is used with one or two arguments. Finally conjunctions are dyadic and take two arguments as, for example, `7&|` , where `&` is the conjunction *bond*, which gives a monadic verb for the 7-residue of its argument as we shall see later in this section.

The adverb *table*, then, requires as well as a verb argument array arguments on the left and right. The result is an array formed by inserting the verb between all possible pairs of items chosen from the two arguments. For example, if `p` is the previously defined list of the first `5` positive integers, and

```
q=: 1 2 3 4 5 6 7 8 ,
```

then `p+/q` is the addition table

```
2 3 4 5  6  7  8  9
3 4 5 6  7  8  9 10
4 5 6 7  8  9 10 11
5 6 7 8  9 10 11 12
6 7 8 9 10 11 12 13
```

and `p*/q` is the multiplication table

```
1  2  3  4  5  6  7  8
2  4  6  8 10 12 14 16
3  6  9 12 15 18 21 24
4  8 12 16 20 24 28 32
5 10 15 20 25 30 35 40   .
```

We might remark here that adverbs and conjunctions take precedence over verbs with the

left argument being the entire verb phrase on the left. A good example is the modular-7 multiplication table discussed in an earlier section which is given by the expression

```
7|*/~ 1 2 3 4 5 6
```

which is equal to

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

In this expression the adverb / has the verb argument 7|*. The adverb ~ *reflex* whose argument is the derived verb 7|*/ gives this verb the equal left and right arguments of 1 2 3 4 5 6. The entry in the fourth row and fifth column is 6, the remainder when the product of 4 and 5 is divided by 7, or 7|4*5.

As another example let

```
D=: 4 1 4 1 1 2 4 4 2 2
```

represent the results of rolling a tetrahedral die 10 times. Then the expression

```
1 2 3 4 =/D  ,
```

where = is the dyadic verb *equal*, has the value

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

and gives the distribution of the faces in the 10 rolls. The first row shows that a 1 occurred on the second, fourth and fifth rolls, the second row that a 2 occurred on the sixth, ninth and tenth rolls, etc. The row sums,

```
+/"1 (1 2 3 4 =/D)  ,
```

are 3 3 0 4, the frequencies of occurrence of the faces 1, 2, 3 and 4 on the 10 rolls.

The arguments of the table adverb need not be restricted to one-dimensional lists. For example, let DD be the two-dimensional array

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

where the rows give the results of three repetitions of rolling the tetrahedral die 10 times. Then

```
1 2 3 4 =/DD
```

is the three-dimensional array

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

0 0 0 0 0 0 1 0 0 1 1
0 0 1 0 0 1 0 0 1 1
0 0 0 0 1 0 1 0 1 0

0 0 0 0 0 0 0 0 0 0
1 1 0 0 1 0 1 1 0 0
1 1 1 1 0 0 0 1 0 0

1 0 1 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0  ,
```

and

```
+/"1 (1 2 3 4 =/DD)
```

gives the frequencies

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

of the faces 1, 2, 3 and 4 for each of the three repetitions of 10 rolls.

APL has long been criticized for being different than other languages and for obscuring much of the detail that is required in programming with conventional programming languages. Many of the critics of the language have described it as "the language with all the funny symbols". **J** is undoubtedly being criticized for many of the same reasons with the change that the symbols used, which are the standard ASCII characters and which are therefore not funny, are being used in funny ways.

Such criticisms of array languages remind me of

His [Oliver Heaviside's] use of unconventional mathematical notation, including that of vector calculus which was not generally used at the time, was greeted with scorn by other physicists. Because of his use of unorthodox methods he was forced to publish some of his work at his own expense.

---

the reception given to the work of Oliver Heaviside (1850 - 1925), the eccentric English physicist and engineer who worked on the theory of electrical circuits and who was the first person to write Maxwell's equations for the electromagnetic field in

modern form. His use of unconventional mathematical notation, including that of vector calculus which was not generally used at the time, was greeted with scorn by other physicists. Because of his use of unorthodox methods he was forced to publish some of his work at his own expense. Once when asked how he could use formal manipulations without understanding how they worked, he is said to have replied "Should I refuse a good dinner simply because I do not understand the processes of digestion?" The parallels between Heaviside's experiences and those of the advocates of array languages are all too apparent.

About verbs

A style manual which I picked up shortly after it was published is *The Writer's Hotline Handbook* by Michael Montgomery and John Stratton (Mentor, 1981). This little paperback is the outgrowth of a "hotline" radio program staffed by two members of the English Department at the University of Arkansas at Little Rock. I probably use it more than any other style manual I have. I think that if I consulted it more often, I would be a better writer.

In any event, the other day I happened to turn to the very beginning of Chapter 1, "Verbs: The Essential Words", and read the following:

> Simply put, verbs are the most important parts of sentences. Sentences are built around them, and a writer who can handle them with skill and accuracy has traveled a long way toward becoming an effective writer.

The chapter of some fifteen pages gives a discussion of grammatical properties - number, mood, tense, voice, etc. - verb tables, main clauses and that-clauses, and the use of the troublesome verb pairs "shall" and " will", and "should" and "would".

In some ways verbs are much simpler in **J** than they are in English. We need only be concerned with their valence - either monadic with one argument or dyadic with two - and whether they can be used ambivalently, i.e., either monadically or dyadically as appropriate, and whether they are defined tacitly or explicitly. Furthermore, it is we who define them, and, when we do, it is we, like Humpty Dumpty, who are in charge and decide what they do.

On the other hand, we do have the responsibility that the verbs we define are correct and do what they are supposed to do. We also are responsible for defining the verbs in a style which is readable both to ourselves and to others who may be interested in what we are doing. Finally we are responsible that verbs have meaningful names that are suggestive of their function. Indeed, giving them names such as `xxx` or even worse the only too common `foo` which appears in many programming examples (unfortunately even in **J**) is only an admission that we really do not understand them. The observant reader may respond that I have defined the verbs `f` and `g` in discussing Dr. Durrani's musings on large numbers. Yes, I must confess I did, but I did admit there that I, like Dr. Durrani, couldn't grasp the significance of the large numbers being expressed.

Speaking of the importance of names, I am reminded of a passage in Peter Ackroyd's biography *Dickens* (Mandarin, 1994) when he was discussing Dickens' attempt to find a surname for Martin Chuzzlewit - apparently he had no trouble deciding that Martin was a suitable first name - having first rejected the names Chuzzlewig, Chubblewig, Chuzzletoe and Chuzzlebog, amongst others. Then Ackroyd continues as follows:

> For names were very important to Dickens. When he started a new periodical he told Forster [a close friend and his first biographer] that "I shall never be able to do anything for the work until it has a fixed name", and it is the same with his characters also. They did not exist for him until he had given them a name and it is that which, like a spell, brings forth their appearance and behaviour in the world.

And what names he selected for the illumination of his stories and for our own delight: Mr. Turveydrop, and Sir Leicester and Lady Dedlock in *Bleak House*; Mrs. Wititterly and her doctor Sir Tumley Snuffim in *Nicholas Nickleby*; the Murdstones (brother and sister) and Mr. Dick in *David Copperfield*; and ... . But in my enthusiasm for Charles Dickens, I digress. This section is supposed to be about verbs.

We shall return one last time to the coupon collector's problem and discuss the verbs used there. Let us recall - again for the last time - that the

expected sample size required to obtain *n* different prizes is "*n* times the sum of the reciprocals of the first *n* positive integers". We have seen that the corresponding **J** expression, not a verb, for any non-negative value of `n` is

```
n * +/ % >: i. n .
```

Before discussing the **J** verbs for this problem we shall introduce two utility verbs and review the important concept of uninterrupted sequences of verbs.

A useful verb, which we shall use here and in a later section, is

```
pos=: >: @ i. ,
```

"increment after integer", which gives a list of positive integers, and, for example, `pos 5` is `1 2 3 4 5`. A related verb, which we shall not use here but may be of interest, is

```
ei=: i. @ >: ,
```

"integer after increment", which gives "extended integers", and, for example, `ei 5` is `0 1 2 3 4 5`.

An uninterrupted sequence of an arbitrary number of verbs is called a *train*. We have mentioned much earlier in this paper that a sequence of three verbs is a *fork* and have given the verb

```
am=: +/ % #
```

for the arithmetic mean as an example. For the list

```
w=. 2.3 5 3.5 6
```

the expression `am w`, which has the value `4.2`, is equivalent to

```
(+/ % #) w
```

which is

```
(+/w) % #w
```

or `16.8 % 4` or `4.2`. A sequence of two verbs is a *hook*, and as an example the verb

```
dev=: - am
```

gives deviations from the mean, and `dev w` is equivalent to

```
(- am) w
```

which is

```
w - am w
```

or

```
_1.9 0.8 _0.7 1.8 .
```

An arbitrary sequence of verbs may be evaluated by repeated resolution into zero or more forks followed by a final fork or hook. For example, we may combine the above fork for the arithmetic mean and the hook for deviations from the mean into the single train `- +/ % #`, and

```
(- +/ % #) w
```

gives the deviations

```
_1.9 0.8 _0.7 1.8 .
```

We might note that this last expression is not the same as the expression

```
- +/ % # w
```

which has the value `_0.25` since without parentheses the normal rules of precendence apply.

Now let us return to the coupon collector's problem and consider first a verb for the expected sample size. We shall give the verb the name `ccexp` which is short and suggestive of its function although previously we have used the shorter `cc`. The calculation of the expected sample size as "n times the sum of the reciprocals of the first n positive integers" suggests the use of a hook with the left verb "times" and the right verb a compound verb giving "the sum of the reciprocals of the first n postive integers". The left verb is obviously `*`, and the right verb may be expressed as "`+/` after `%` after `pos`" or

```
+/ @: % @ pos .
```

We note the use of the two conjunctions *at* `@:` and *atop* `@` for *after*, the first being required so that the sum is applied over the list of reciprocals rather than to each item in the list. Therefore, the verb for the expected sample size may be written as

```
ccexp=: * +/ @: % @ pos
```

and, for example, `ccexp 5` is approximately `11.4`.

All of the verb definitions we have seen so far in this paper have been functional or tacit without explicit arguments. Although we have made one mention in passing of verbs with explicit arguments and have used them in obtaining some of the results we have quoted, we have yet to discuss their definition. Since we shall need two explicit verbs in a further treatment of the coupon collector's problem, we shall now discuss them briefly.

To illustrate some of the main features of explicit definition we shall define the following three very simple verbs `f1`, `f2` and `f3`:

```
f1=: 3 : 0
% y.
)
```

```
f2=: 3 : 0
:
x. % y.
)
```

```
f3=: 3 : 0
% y.
:
x. % y.
)
```

The monadic verb `f1` gives the reciprocal so that, for example, `f1 2.5` is `0.4`, and the dyadic verb `f2` is divide so that `15 f2 6` is `2.5`. The verb `f3` is ambivalent so that it gives the reciprocal when used monadically and the quotient when used dyadically, i.e., `f3 2.5` is `0.4` and `15 f3 6` is `2.5`. The first line in each definition gives the name and specifies that the definition is for a verb. The last line in a definition must be right parenthesis `)`. A colon `:` separates the monadic and dyadic definitions and is omitted for a monadic verb. Left and right arguments are represented by `x.` and `y.`, respectively.

To give an example of an explicit verb written with control structures in the style of a conventional programming language we give the following version of `ccexp`:

```
ccexp=: 3 : 0
n=. y.
i=. 1
sum=. 0
while. i <: n do.
   sum=. sum + %i
   i=. >: i
end.
n * sum
)
```

We note the use of *local is* `=.` for assignment within this definition to avoid any effect on other variables with the same names which may be in use elsewhere. On the other hand if we wanted the value of `sum` to available outside of `ccexp` we could use the *global is* `=:`, which we have used throughtout this paper, in the two statements

```
sum=: 0
```

and

```
sum=: sum + %i  .
```

In our earlier discussion of the coupon collector's problem we considered the simulation of collecting coupons for an arbitrary number of different coupons to avoid the expense of having to buy packages of cereal. This simulation was accomplished very simply by repeatedly sampling from the required non-negative integers using the monadic verb *roll* `?` until all of the integers were present in the sample. This is equivalent to repeated sampling until the nub of the sample, i.e., the number of distinct items, is equal to the number of coupons. This simulation is accomplished by the following verb:

```
ccsim=: 3 : 0
n=. y.
Sample=. i. 0
while. n > # ~. Sample do.
   Sample=. Sample, ?n
end.
>: Sample
)
```

The following are three examples of the use of this verb:

```
   ccsim 5
5 3 1 4 3 4 5 4 2
   ccsim 5
1 4 2 4 4 5 2 2 5 4 4 4 1 4 5 2 3
   ccsim 5
4 3 2 2 2 1 3 5
```

Finally we consider repeated simulations for a given number of coupons so that we may investigate the distribution of the sample size. This will be accomplished by the dyadic verb `ccsize` so that, for example, `10 ccsize 5` will give the results for `10` simulations for `5` coupons and could have a value such as

```
   7 12 13 19 16 8 9 12 8 17 .
```

The verb `ccsize` may be defined either tacitly or explicitly depending on a person's experience - or whims. The tacit version is

```
ccsize=: (#@ccsim)"0 @ #
```

and the explicit, possibly more readable, version is the following:

```
ccsize=: 3 : 0
:
Reps=. x.
n=. y.
SampleSize=. i. 0
while. Reps > # SampleSize do.
   SampleSize=. SampleSize, ccsim n
end.
)
```

We shall conclude with a few general remarks about trains, forks and hooks, and shall introduce the verb *cap* `[:` which is at times useful in expressing a sequence of functions using trains.

In general, if `f`, `g` and `h` are verbs and `y` is a noun, then the monadic fork

```
(f g h)y
```

is equal to

```
(f y) g (h y).
```

An example is, of course, the arithmetic mean

```
am=: +/ % #
```

which we have seen several times in this paper. Forks may also be dyadic with two arguments, and, if `x` and `y` are nouns, then the dyadic fork

```
x(f g h)y
```

is equal to

```
(x f y) g (x h y) .
```

One example is the expression

```
7 (+ * -) 3
```

which is equivalent to

```
(7 + 3) * (7 - 3)
```

and is equal to `40`.

Hooks, also, may be either monadic or dyadic, and the monadic hook

```
(g h)y
```

is equal to

```
y g (h y),
```

and the dyadic hook

```
x (g h) y
```

is equal to

```
x g (h y).
```

Suppose now we wish to construct a verb to find the sum of squares of the deviations of a list of observations from their arithmetic mean. If we use the verb

```
dev=: - am
```

which we defined earlier in the section, then a verb `ss` for the sum of squares may be defined as

```
ss=: +/ @: *: @ dev ,
```

and for

```
w=. 2.3 5 3.5 6
```

we have that `ss w` is `7.98`. Alternatively, we may define the verb as

```
ss=: [: +/  [: *: dev
```

where the verb cap `[:` caps the left branch of each of the forks and allows the verbs `*:` and `+/` to be used monadically. Finally we may define

```
ss=: [: +/ [: *: [ - +/ % #
```

where the verb is now a train of primitive verbs. The choice between the use of the conjunctions `@` and `@:` or `[:` appears to be a matter of style and taste for which no useful rules may be given.

Going in circles

All of us have undoubtedly heard stories such as the following and have possibly told them when we were children:

> It was a dark and stormy night, the rain came down in torrents, there were brigands on the mountains, and thieves, and the chief said unto Antonio: "Antonio, tell us a story." And Antonio, in fear and dread of the mighty chief, began his story: "It was a dark and stormy night, the rain came down in torrents, there were brigands on the mountains, and thieves ..."

This example is given in *Vicious Circles and Infinity* by Patrick Hughes and George Brecht (Penguin Books, 1978), as an illustration of how self-reference in story-telling can lead to an infinite regression. The authors also quote the following example taken from a work published in 1899 of how a visual self-reference can lead to an infinite regression:

> Let us imagine that a portion of the soil of England has been levelled off perfectly and that on it a cartographer traces a map of England. The job is perfect: there is no detail of the soil of England, no matter how minute, that is not registered on the map; everything has there its correspondence. This map, in such a case, should contain a map of the map, which should contain a map of the map of the map, and so on to infinity.

The appeal of such stories to children is, of course, that they go on forever. There is no restriction that Antonio tell his story only a specified number of times or that persons drawing the sequence of maps stop when the current map becomes too small to be read.

Another example of an endless cycle, and one which presents no problems for us, is the definition of our ancestors. They are obviously our parents, and our grandparents, and our great-grandparents, and so on. Alternatively we may define our ancestors as our parents, and the ancestors of our parents who are in turn their parents and their parent's ancestors, and so on. We know that this definition is cyclical and endless, but we simply stop the cycle when we have

gone back a sufficient number of generations for our purposes.

Verbs in **J**, and procedures and functions in most programming languages, may be defined in terms of themselves provided there is some mechanism to prevent them from cycling endlessly. Such verbs are said to be recursive from the Latin meaning a return. I have found that I seldom need to define such verbs in **J** but nevertheless I think that the concept is of sufficient interest and importance to merit a little consideration here. By way of introduction we shall use two simple examples we have already discussed, viz., the factorial function and the Tower of Hanoi puzzle.

In the verbs that are given for these two examples we shall use the utilities `ELSE`, `IF` and `RIGHT` which will be discussed in the Appendix to this section where a third recursive verb will be given. I believe that their use simplifies the definition of recursive verbs, and the first two at least make these definitions more readable. The third one refers to the right argument of the function being defined. I am indebted to Howard Peelle of the University of Massachusetts in whose work I first saw the definition and use of similar utilities for recursive verbs.

We recall that the factorial function is defined in terms of the successive positive integers, so that, for example, the factorial of 6 is $6 \times 5 \times 4 \times 3 \times 2 \times 1$ or 120. Since the factorial of 5 is equal to the product of the first five positive integers, we could alternatively say that the factorial of 6 is 6 times the factorial of 5. Then we could define the factorial of 5 as 5 times the factorial of 4, and so on until we reach the factorial of 0 which is 1 as we previously noted. This alternative definition of the factorial function is recursive since we are defining the factorial function in terms of itself. However unlike the examples given above it does not go on forever but stops when we reach the factorial of 0. The sequence of steps we have just described may be represented in **J** as follows:

```
!6
6 * !5
6 * 5 * !4
6 * 5 * 4 * !3
6 * 5 * 4 * 3 * !2
6 * 5 * 4 * 3 * 2 * !1
6 * 5 * 4 * 3 * 2 * 1 * !0
6 * 5 * 4 * 3 * 2 * 1 * 1
```

If we evaluate this last expression, we obtain the value of `720` for the original expression `!6`.

A recursive verb for the factorial function is given by

```
factorial=:
   (RIGHT*factorial@(RIGHT-1:))
      ELSE 1: IF (RIGHT=0:)
```

(We note in passing that `0:` and `1:` are verbs that have the values `0` and `1`, respectively, regardless of their arguments. In reading the above definition we may think of them simply as the constants `0` and `1`.) We have, for example, that `factorial 6` is `720`, and any reader who works through the successive evaluations of the verb for this calculation will see that they are identical to those in the example in the previous paragraph. This recursive verb may be used for extended integer calculations and we have that `factorial 52x` is

```
80658175170943878571660636856403766975289505440883277824000000000000
```

which is, of course, the value of `!52x` as we have seen in a previous section.

We recall that the Tower of Hanoi puzzle consists of a small board to which are fastened three vertical pegs on one of which are stacked a number of discs of diminishing radii. The problem is to transfer the discs to a second peg by moving the discs one at a time using the third peg when necessary and never placing a disc on top of a smaller disc. For a given number of discs we may list the minimum moves required to solve the problem by simply giving the number of the disc to be moved with the smallest disc being number 1, the next smallest disc being number 2, and so on. If there are four discs, the list of moves is

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

which represents a solution if we think of the pegs as being near the vertices of a triangular base and noting that odd-numbered discs always are moved in a clockwise direction and even-numbered discs in a counter-clockwise direction.

A recursive solution to this puzzle, which may be simply expressed as a recursive verb in **J**, is suggested by an argument very similar to the inductive proof for the expression $2^n - 1$ for the minimum number of moves for n discs. To solve the

puzzle we simply move all discs except the bottom one all at once to the third peg, then move the remaining largest disc to the second peg and make a note of its number, and finally move the discs on the third peg all at once to the second peg. Thus by a little cheating we have solved the problem. However, being honest, we realize we still have to solve the problem of the two moves of all of the discs except the largest one which has been moved legitimately. We now solve each of these reduced problems in the same way by moving all of the discs except the bottom one in the reduced pile, then moving the bottom one, and finally moving all of the discs in the moved reduced pile all at once. We keep repeating this procedure, recording the one legitimate move in each trio of moves, until finally we have the pile reduced to a single disc which can be legitimately moved. A verb for carrying out these moves and recording the disc number involved in the middle move in each trio of moves is given by

```
Hanoi=: ((Hanoi@(RIGHT-1:)),
            RIGHT,
          (Hanoi@(RIGHT-1:)))
         ELSE 1: IF (RIGHT=1:)  ,
```

and the expression `Hanoi 4` will give the sample results in the last paragraph for four discs.

I believe that I first began to understand recursive definition by thinking about the Tower of Hanoi puzzle. It has the great advantage of being easy to visualize especially if one has a model to work with. The one I have had beside me as I wrote these last two paragraphs was made by Brio, the Swedish maker of high-quality children's toys and was given to many years ago by a Danish colleague to whom I am still grateful. It has a black triangular base, red, blue and green pegs, and eight discs of the same colours. It is in my study along with a Brio dachsund dog which I bought for my daughter when she was a little girl. I also have a less elegant model in my office which has been used by myself and many of my departmental colleagues for teaching recursion for almost thirty years.

One example of recursive definition that has always interested me is its use in the formal specification of the syntax, i.e., grammar, of a language. It was first used for the description of programming languages by John Backus, the manager of the programming research group in IBM during the development of FORTRAN in the 1950s, when he was working on ALGOL 58, the first of the programming languages whose name is an acronym for "algorithmic language". It was further developed by Peter Naur who was the editor of the paper "Report on the algorithmic language ALGOL 60" published in 1960 and of a revised report which was published three years later. These documents gave in some fifteen pages a formal description of the entire language, and were described by one person writing on programming languages at the time in terms something like "the most unintelligible pieces of meaningful English prose ever written". The method of description is referred to as BNF which stands either for Backus Normal Form or Backus-Naur Form depending on whom you ask.

The use of BNF requires only the introduction of the simple terms "the class of all objects" and "is defined as" and the conjunctions "or" and "and". The class of all objects is represented by enclosing the name of the class of objects in angular brackets so that, for example, <cats> represents the class of all (domestic) cats. The second concept is represented by "::=" which may considered to be the copula "is". The conjunction "or" is represented by "|", while the conjunction "and" is expressed not by a symbol but by the juxtaposition of the two objects being joined. Thus "the class of all objects", "is defined as", "or" and "and" may be considered the metalanguage which we shall use to describe the grammar of another language. The first example we shall give is from the English language and the second from the **J** language.

The *American Heritage Dictionary* defines the subject of a sentence as "A word or phrase in a sentence that denotes the doer of the action, ...". We shall assume that a subject will be simply a noun preceded optionally by one of the articles "a", "an" or "the". Thus "<subject>" represents the object being defined, and "<noun>" represents the class of all nouns which can be used in a subject and which we will not specify further. We can define the three articles by the statement

<article>::= a|an|the

which may be read as "an article is defined as "a" or "an" or "the". Now we can define "subject" by the statement

<subject>::= <noun>|<article><noun>

which may be read as "a subject is defined as a noun, or an article followed by a noun".

Now let us formally give the definition of an integer in **J**. We know that an integer consists of one or more of the digits `0`, `1`, `2`, ... preceded when necessary by the plus sign `+` or the negative sign `_` although the positive sign is usually omitted. We do not specify the maximum number of digits that may be accommodated without using extended precision arithmetic as this may depend on the implementation and can always be given as a comment. Therefore in BNF the definition of an integer may be written as follows:

```
<digit>::= 0|1|2|3|4|5|6|7|8|9
<sign>::= +|_
<unsigned integer>::= <digit>|
        <unsigned integer><digit>
<integer>::= <unsigned integer>|
        <sign><unsigned integer>
```

(The recursion occurs, of course, in the definition of an unsigned integer.) What could be simpler?

Two languages with which I am familiar and which have a BNF description are ALGOLW and Pascal. Occasionally I have found these descriptions quite useful. However one may use any language quite effectively without consulting its formal description as do almost all speakers of English or any other natural language. However, one place where a knowledge of BNF may have been useful is in understanding, possibly "deciphering" would be a better word, the descriptions given for some of the commands in DOS, the operating system which fortunately has been replaced by the various versions of Windows and other graphical user interfaces. For example, copying, which may be done with a few mouse operations in Windows, requires the DOS "copy" command which is defined formally as follows:

To copy files:

copy [*drive*:]*pathname1* [*drive*:]
[*pathname2*][/**v**][/**a**][/**b**]

or

copy [*drive*:]*pathname1* [/**v**][/**a**][/**b**]
[*drive*:][*pathname2*]

To append files:

copy *pathname1* + *pathname2* [...] *pathnameN*

This description is followed by almost three pages of notes.

In the two examples of recursive verbs given in this section we made use of the utilities `ELSE`, `IF` and `RIGHT` to make the verbs more readable. The first two utilities are defined as

```
ELSE=: `
```

and

```
IF=: @. ,
```

where `` ` `` and and `@.` are the conjunctions *tie* and *agenda*, respectively, while the third is

```
RIGHT=: ]  ,
```

where `]` is the verb *right*. We shall discuss these utilities, and give the two recursive verbs discussed earlier in the section, and one additional verb, in the manner more commonly used in **J**.

The Writer's Hotline Handbook, mentioned in the previous section, says that "Gerunds are verb forms (such as *swimming* and *working*) that end in *ing* and function as nouns." and gives as one example "*Swimming* is a favorite summer pastime." In **J** the tie conjunction may be used with two verb arguments to form a gerund whose elements may be selected, just as the constant items of a list may be selected, and executed. An example is

```
+`*/1 2 3 4 5  ,
```

or

```
+`*`+`*/1 2 3 4 5  ,
```

which is equivalent to `1+2*3+4*5` or `47`. A gerund may be used with `@.` for selecting a verb depending on the value of its arguments. For example,

```
2 +`* @. > 4
```

is equal to `6` since `2 > 4` is `0` since `2` is not greater than `4` so the verb `+` is selected, and

```
4 +`* @. > 2
```

is equal to `8` since `4 > 2` is `1` since `4` is greater than `2 and *` is selected. The verb `]` always gives its right argument, and, for example, `3]8` is `8`.

Alternative definitions for the two recursive verbs given earlier are

```
factorial=: 1:`(]*factorial@<:) @.
                (]>0:)
```

and

```
Hanoi=: 1:`((Hanoi@(]-1:)) , ] ,
                (Hanoi@(]-1:)))
                @.(]>1:)  .
```

As a final example we shall give a verb for finding the sum of the items of a list where the sum is defined recursively as the last item plus the sum of all of the items other than the last. In this definition we shall need the verbs *tail* `{:` and *curtail* `}:` for the last item of a list and all items except the last, respectively. For example,

```
    {: 2.3 5 3.5 6
```
is `6`, and
```
    }: 2.3 5 3.5 6
```
is
```
    2.3 5 3.5 .
```
Recursive definitions for the verb are given by
```
    rsum=: ({:@RIGHT + rsum@}:@RIGHT)
               ELSE 0: IF (0: = #@RIGHT)
```
and
```
    rsum=. 0:`({:+rsum@}:) @. (#>0:) ,
```
and with either definition
```
    rsum 2.3 5 3.5 6
```
is `16.8`.

It is worthwhile examining the execution of this verb for the simple example given above where we note that `i.0` occurring in the third last line represents an empty list:
```
    rsum 2.3 5 3.5 6
    6 + rsum 2.3 5 3.5
    9.5 + rsum 2.3 5
    14.5 + rsum 2.3
    16.8 + rsum i.0
    16.8 + 0
    16.8
```
We see that the recursive evaluation of the sum terminates when the number of items in the list is zero, i.e., when the list argument by successive elimination of its last item becomes empty and we assume that the sum of an empty list is zero. In the remainder of this section we shall justify this assumption for the sum of the items in an empty list and make some further remarks about empty lists.

Suppose, now, that we are interrupted while are finding the sum, either by hand or by using a pocket calculator, of some arbitrary list of numbers. It would be reasonable to make a note of the partial sum so that after the interruption we could add up the remaining numbers, and then find the required sum by adding the two partial sums. Such a preocedure will always give the correct sum regardless of how many numbers we have added before we were interrupted. If there was a total of twenty numbers, there could be five in the first sum and fifteen in the second, or sixteen in the first and four in the second, and so on for any arbitrary division of the twenty numbers into two groups. The required sum will always be the sum of the two partial sums. We shall insist that this relationship hold if there are no numbers in either of the groups, i.e., if we are interrupted just before we begin our work when there will be no numbers in the first group and all of the numbers in the second group or just after we have completed it when all of the numbers will be in the first group. This requires that we make the assumption that the sum of a list of no numbers, i.e., the empty list, is zero. Formally we say that the identity function for addition is zero, and in **J** we have that `+/i.0` is `0`. Further identity functions in **J** are `1` for multiplication, negative infinity for maximum, and infinity for minimum, or `*/i.0` is `1`, `>./i.0` is `__` and `<./i.0` is `_`. As we have seen with addition, identity functions are useful in ensuring that obvious relationships hold with partitioned lists when one of the partitions is empty.

Similar identity functions are valid in APL. Some years ago a good friend and colleague gave me a wonderful wood carving he had made of the identity function for multiplication:

$$1 = \times / \iota\ 0$$

I have it in my study where it often reminds me of our long collaboration with APL and Nial.

Watching television

As a final example we shall analyze some data on the lengths of program segments and commercial breaks during a television program. The data come from a viewing of a half-hour sitcom picked almost at random from the innumerable inane programs that appear to take up so much of the time on commercial television these days. The data are given by the list `Times` which is
```
    ÚÄÄÄÄÄÂÄÄÄÄÄÄÂÄÄÄÄÄÂÄÄÄÄÄÄÄÂ
    ³9 0 8³9 2 19³9 5 0³9 13 27³
    ÀÄÄÄÄÄÁÄÄÄÄÄÄÁÄÄÄÄÄÁÄÄÄÄÄÄÄÁ
         ÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄ¿
         9 16 32³9 26 41³9 29 25³9 30 0³
         ÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÙ
```
We see that the program began at 9:00:08 p.m., the first commercial started at 9:02:19, and so on with the program finishing at exactly 9:30. From these

data we wish to find the total amount of program time and of commercial time.

We recall the *base* verb `#.` introduced earlier and its use with mixed base number systems to convert from hours, minutes and seconds to seconds. For this purpose we shall define the verb

```
tosecs=: 24 60 60&#. ,
```

so that, for example,

```
tosecs 5 25 10
```

is equal to `19510`, the number of seconds in `5` hours, `25` minutes and `10` seconds. Then the number of elapsed seconds since noon for each item of the list `Times` is given by

```
tosecs Each Times ,
```

where `Each` is a utility adverb which has been used previously, which has the value

```
32408 32539 32700 33207 33392 34001
   34165 34200 .
```

Therefore, the length of first program interval is

```
32539 - 32408
```

or `131` seconds, the length of the first commercial break is

```
32700 - 32539
```

or `161` seconds, etc. Thus the lengths of the alternating program and commercial segments of the program are given by what is known as the "first differences" of the list of the elapsed times in seconds, i.e., second item minus the first, third item minus the second, etc. If we introduce the utility verb `diff` for first differences, we may compute the elapsed times as

```
t=: diff tosecs Each Times
```

which has the value

```
131 161 507 185 609 164 35 .
```

We note that the lengths of the program intervals are the items with the even indices `0`, `2`, `4` and `6`, and the lengths of the commercial intervals have the odd indices `1`, `3` and `5`. Thus we may select the lengths of the program intervals by the statement

```
tp=: (0=2|i.7) # t ,
```

since `0=2|i. 7` is the list `1 0 1 0 1 0 1` and `#` is the verb *copy* which copies the items of its right argument corresponding to the `1`s in its left argument, and `tp` has the value

```
131 507 609 35 .
```

Similarly, the lengths of the commercial intervals is given by

```
tc=: (1=2|i.7) # t
```

and `tc` is equal to

```
161 185 164 .
```

The total times for the program and the commercials are

```
60 %~ +/tp
```

or `21.3667` minutes, and

```
60 %~ +/tc
```

or `8.5` minutes, respectively. Finally, the percentage of commercial time is

```
100 * (+/tc) % +/tp,tc
```

which is approximately `28.5`.

A similar analysis was done for two James Bond movies, *The Spy Who Loved Me* and *Dr. No*, and gave for each movie a total commercial time of twenty-one percent.

The following ambivalent verb `TVcomm` allows these compuations to be performed simply:

```
TVcomm=: 3 : 0
0 TVcomm y.
:
Switch=. x.
Times=. y.
t=: diff tosecs Each Times
'tp tc'=. Switch|.(2|i.#t)</.t
(60%~+/tp,tc),(100*(+/tc)%+/tp,tc),#tp
)
```

Either of the expressions

```
TVcomm Times
```

or

```
0 TVcomm Times
```

is equal to

```
29.8667 28.4598 4
```

giving the total time of the program in minutes, the percentage of time for commercial breaks, and the number of commercial breaks. All of the calculations discussed here assume that the program begins with a program rather than a commercial segment. If the program were to begin with a commercial, then the verb may be used with a left argument of `1` so that with the same data we have that

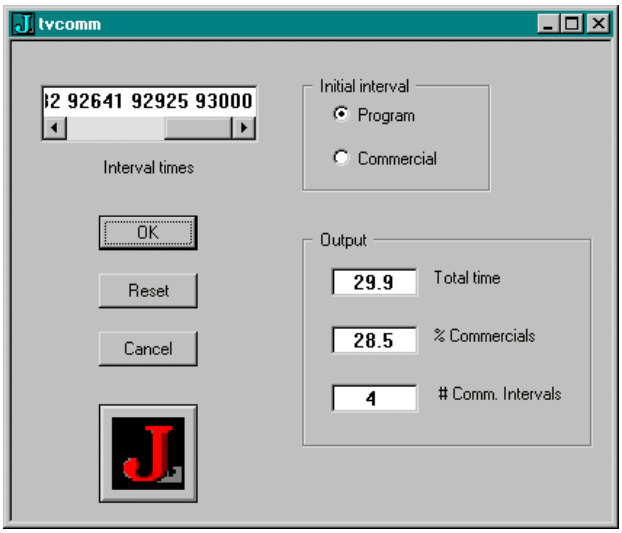
```
1 TVcomm Times
```

is equal to

```
29.8667 71.5402 3
```

which, although based on an incorrect assumption, is reasonable since it complements the correct results.

The Windows form on the next page allows all of these calculations to be performed simply without an understanding of the verbs on which they are based.



We shall conclude our further look at **J** by discussing briefly the adverbs *prefix* and *suffix*, both represented by `\`, and *key* represented by `\.`. The second of these has been used in the verb `diff` for first differences and the third in `TVcomm` for grouping the lengths of the time segments into those for programs and those for commercials.

The monadic adverb *prefix* applies its verb left argument to each of the prefixes of its noun right argument, i.e., to the first item, the first two items, the first three items, etc. For example, since `pos 5` is the list of the first `5` positive integers, the prefixes are given by `<\pos 5` and are

```
ÚÄÂÄÄÄÂÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄ¿
³1³1 2³1 2 3³1 2 3 4³1 2 3 4 5³
ÀÄÁÄÄÄÁÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÙ
```

and `+/\ pos 5` gives the cumulative sums

```
1 3 6 10 15
```

and `*/\ pos 5` the cumulative products or factorials

```
1 2 6 24 120 .
```

The expression `+/+/\pos 12` which is the sum of the cumulative sums of the first `12` positive integers is equal to `364` and may be interpreted as the total number of gifts given in the English carol "The Twelve Days of Christmas".

The dyadic adverb *infix* `\` applies its verb argument to infixes of its right noun argument, overlapping infixes if the left noun argument is non-negative and non-overlapping of it is negative, as illustrated by the following examples:

```
   2 <\ pos 5
ÚÄÄÄÂÄÄÄÂÄÄÄÄÂÄÄÄ¿
³1 2³2 3³3 4³4 5³
ÀÄÄÄÁÄÄÄÁÄÄÄÄÁÄÄÄÙ
   2 +/\ pos 5
3 5 7 9
   _2 <\ pos 5
ÚÄÄÄÂÄÄÄÂÄ¿
³1 2³3 4³5³
ÀÄÄÄÁÄÄÄÁÄÙ
   _2 +/\ pos 5
3 7 5
```

If the list of start times for the program and commercial segments is given by

```
s=. 32408 32539 32700 33207 33392
          34001 34165 34200 ,
```

then `2<\s` is the list

```
ÚÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄ¿
³32408 32539³32539 32700³32700 ...
ÀÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÙ
```

of the overlapping pairs of items, and `2-~\s` is the list

```
131 161 507 185 609 164 35
```

of the lengths of the program and commercial segments. Thus it is easy to see that the verb

```
diff=: 2: (-~/\) ]
```

gives the first differences of its argument.

The dyadic adverb *key* `/.` groups items of the right noun argument according to the key given by the left noun argument and then applies its verb argument to each group. For example, for the dice data `D` given by the list

```
4 1 4 1 1 2 4 4 2 2
```

representing `10` rolls of a tetrahedral die given earlier, the expression `D > 3` is the list

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

where the `1`s give the positions of the items of `D` which are greater than `3`. Then

```
(D > 3) </. D
```

is the two-item list

```
ÚÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄ¿
³4 4 4 4³1 1 1 2 2 2³
ÀÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÙ
```

where the first item gives those items of `D` which are greater than `3` and whose second item are those items of `D` which are less than or equal to `3`. The expression

```
(D > 3) #/. D
```

which is equal to `4 6` gives the number of items satisfying each of these relationships.

Now we recall that the lengths of the segments of the television program are given by the list `t` which is equal to

```
131 161 507 185 609 164 35 ,
```

and that the even-indexed items represent program segments and the odd-indexed items represent commercial segments. Thus it may be seen that the expression

```
(2|i.#t)</.t
```

gives the two-item list

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄ¿
³131 507 609 35³161 185 164³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

partitioning the segments into the required groups.

Part Seven. From J to Japanese

Teaching languages

Shortly after I retired I started to study Japanese. There were several reasons but the main one was probably that I just like languages. I'm not sure what I expected to learn, but now I can read and write a little, speak very little, and hear almost not at all. What I didn't expect, though, was to become interested in the teaching of Japanese. My enjoyment in seeing how the language is presented helps compensate for my rather spectactular lack of progress in it.

A colleague once remarked to me that most introductory programming courses were as interesting as a course in the conjugation of verbs. I didn't disagree with him.

Most of my Japanese texts teach the language by the telling of some continuing story which although fictional is intended to be realistic. In one book the story is about a young American who goes to Yokohama to teach in a Japanese school. In another we learn something of the day-to-day life of an American college student who is spending some time in Japan as an exchange student. In other books the stories are based on the experiences of business people, often accompanied by their families, working in Japan.

Let me mention my favourite book very briefly. It is *Business Japanese* by Michael Jenkins and Lynne Strugnell (NTC Publishing Group, 1993) and is in the well-known English "Teach Yourself Books" series. The story revolves around Wajima Trading Company in Tokyo and the British company Dando Sports which wants to market its sporting equipment and clothing in Japan through Wajima. We are introduced to various members of the staff at Wajima and learn about the company's organization and how business operates in Japan. One of the main characters is Mr. Lloyd, marketing manager for Dando, who visits Japan on two occasions to draw up a contract. We follow Mr. Lloyd as he works with the company and meets some of the staff socially. On his second visit he is accompanied by Barbara Thomas who is in charge of advertising at Dando Sports and who is interested in learning more about the Japanese way of doing business. In one chapter she visits an automobile assembly line and we have a lecture on quality control.

The Japanese hiragana and katakana syllabics are introduced in Chapter 1 and the kanji (Chinese) characters are introduced a few at a time very shortly thereafter. The three classes of characters soon take precendence over the romaji (Roman) characters used for transliteration. There are twenty chapters and the story begins in Chapter 2 with an assistant manager of Wajima checking out of a hotel before returning to Tokyo. Each chapter has the same format: a summary of the story so far (in Japanese beginning in Chapter 12), another installment of the story, in both Japanese characters and in romaji; new vocabulary; grammatical notes; exercises; a short reading exercise; and a one-page essay in English on some aspect of Japanese business. There are several Appendices with grammar summaries, and English-Japanese and Japanese-English glossaries. From the very beginning I had the feeling of meeting real Japanese people working in Japan and living as Japanese people live.

Contrast the introduction to Japanese in these books to the introduction to a programming language in most programming texts and adopted in most introductory courses. (There are a few texts which are exceptions but they do not appear to be very popular.) The language does not seem to

matter. It can be BASIC, Pascal, C/C++ or Java. The texts and courses are really introductions to syntax with numerous examples and exercises intended to illustrate and reinforce grammatical principles. (A colleague once remarked to me that most introductory programming courses were as interesting as a course in the conjugation of verbs. I didn't disagree with him.) Furthermore, many of the exercises are artificial and even juvenile. For example, the first example in Chapter 4 of one text was a program to print either “ho-ho”, “he-he” or “ha-ha”. It was then modified to print “yuk-yuk”. As bad as is the pedagogy, the writing is even worse in some of the books. Good scientific and technical writing does exist, of course, but little is to be found in computing texts.

Of course, expositions of array languages can be poorly organized and presented too. All of us who have worked with APL, Nial or **J** have certainly given lectures and written papers which could have been much better, or possibly even not given or written at all; I know I have. However, one of the advantages of these languages is that they may be used almost immediately to do something useful without first introducing the amount of detail required with conventional languages. Put another way, they may be easily used in the exposition of some subject - statistics, logic, some branch of arithmetic or algebra, say - without having details of the language intrude on the subject matter. Much of my later work has been directed toward this end.

"Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects."

Kenneth Iverson

My attitudes towards the teaching and use of programming languages have been influenced by Kenneth Iverson who has been giving his views in lectures, technical reports and books for almost forty years. Writing on the 25th anniversary of APL in 1991 in a paper giving one of the first published accounts of **J** he wrote "…Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects." Implicit in this statement is the conviction that the details of a language, whether it be **J** or Japanese, should be introduced as needed in the exposition of the subject whether it be teaching multiplication tables in a Canadian classroom or introducing the quality control methods of the American W. Edwards Deming to Japanese assembly lines.

Which language?

In addition to working with **J**, and trying to learn Japanese, and reading, and having lunch at the Faculty Club with my friends, I do a considerable amount of gardening. For my gardening I have a small selection of gardening tools: a rake for raking, a shovel for digging, a trowel for planting, cutters for cutting, and so on. The answer to the question “What tool do I use in the garden?” would depend on what I was doing. If I were planting bedding plants in the spring, I would probably be using a number of different tools. If I were raking the autumn leaves, I would probably be using only a rake.

A similar answer may be given to the question “What programming language do we use or should we use?” The answer depends on the task at hand. In the preparation of this paper, I have used **J**, of course, and for the rainfall example BASIC in addition. I could have used **J** to construct the bar chart for the coupon collector’s problem, but I preferred to use a graphics package.

I might mention here that there is a danger in using a software package or specific program that is designed for one particular calculation or class of calculations. One might use the program just because it is available rather than because of its appropriateness for the analysis of the data. The advantage of a language such as **J**, used either by itself or together with one or more other programs or packages, is that it may prevent one from carrying out an analysis from which completely erroneous conclusions may be drawn.

The question of the best programming language is similar to the question of what is the best natural

language. English? or Japanese? or German? or ...? The answer depends on who we are, where we and our families have come from, what we are now doing, and for whom we are doing it. Regardless of how many languages we know, there will always be one or two which we find most useful and with which we are the most comfortable.

I feel the same way about programming languages. I have used many different ones and have liked them all. As I said at the beginning of this paper, I just like programming. However, it is the array languages - first APL, then Nial and now **J** - that I have found the most appealing intellectually, the most pleasant to use, and the best able to satisfy my computational needs.

Part Eight. Epilogue

Readings

Judging from the number of books I have in my library, I probably spend too much time in bookstores. However, whenever I am in one, I seldom look in the computer section. Indeed, my favourite bookstore hasn't had a separate computer section for some years, preferring to put the few books about computers that it stocks in its science section. In my opinion most computer books are poorly written, overpriced, overweight, and, since they are concerned with the minutiae of the latest software, soon out-of-date. There are some books, though, that I have especially enjoyed reading and rereading, and I would like to mention a few of them now. At the end of this section I shall get back to array languages and give some references for further reading.

The first computer book that I bought was *Faster than Thought* which was edited by B. V. Bowden (Pitman, 1953), and was subtitled "A Symposium on Digital Computing Machines". It is a collection of twenty-four papers written by persons who were working in the new field of digital computation, some of whom are now considered to be amongst the great pioneers of computing. The book was reprinted seven times in the first fifteen years after its publication and still makes enjoyable reading. The editor contributed a Preface and four chapters, the most noteworthy in my opinion being the first, "A Brief History of Computing", which can be read for its literary style alone. Furthermore, Viscount Halsbury, who wrote the Foreword, comments on "Dr. Bowden's mastery of ludricous interpolation [which] is always lurking round the next page as an encouragement to read on". The first example of the editor's humour may be found in the quotation from Edmund Burke with which he begins the Foreword: "The age of chivalry is gone. That of sophisters, economists, and calculators, has succeeded; and the glory of Europe is extinguished for ever." What would Burke say now, just over two hundred years later?

A little book which I enjoyed reading and which I made considerable use of in both my teaching and research was *Electronic Computers* by S. H. Hollingdale and G. C. Tootill (Penguin, 1970). It was published first in 1965 and revised in 1970 and 1975. This book contains an excellent account of the history of computing, a discussion of the design of both analogue and digital computers, a treatment of computer programming, and finally a discussion of various applications of computers. The discussion of programming includes both the machine-language programming of a hypothetical computer and a comparatively lengthy discussion of ALGOL which was the popular language for scientific programming in Europe when the book was written. I have used the machine-language example as a model for some of the simulations of simple computers I have done in APL, Nial and **J**. It is a pity that a modern version of this admirable little book is not available today.

"The age of chivalry is gone. That of sophisters, economists, and calculators, has succeeded; and the glory of Europe is extinguished for ever."

Edmund Burke (1729 - 1797)

We might note that Professor Hollingdale published *Makers of Mathematics* (Penguin, 1989) at the age of 79. In the Preface he remarks that he felt no need to include scholarly footnotes and that the references were "limited, with a few exceptions, to sources from my own library which I consulted while writing this book". A more pleasant way to spend part of one's retirement is difficult to imagine!

Two books have recently come to my attention which provide an interesting contrast to the two books just discussed. They are *Frontiers of Complexity* by Peter Covey and Roger Highfield (Fawcett Columbine, 1995) and *Darwin among the Machines* by George B. Dyson (Perseus Books, 1997). Both books are written in an engaging style, take a strong historical approach to their subject, and have many references. The first is an introduction to the relatively new subject of complex systems and its applications in mathematics, physics, chemistry, biology and the social sciences; the second gives the author's idiosyncratic view of the evolution of computers. Indeed, Covey and Highfield's book with its over sixty pages of endnotes, ten-page glossary, and eleven pages of references may be recommended as a superb introduction to computers for the general reader.

Even this very brief discussion of a few of my favourite computing books would be incomplete without some mention of *Alan Turing. The Enigma of Intelligence* by Andrew Hodges which was originally published by Burnett Books in 1983 and has been published since then in at least two paperback editions. One reviewer described it as "one of the finest pieces of scholarship to appear in the history of computing". This long biography - almost 600 pages in the original edition - gives an incisive account of the life and work of one the pioneers of computing who was the originator of the eponymous Turing machine, a theoretical device for studying the limits of computability. It also sheds light on the English class and education systems, code-breaking during the Second World War in which Turing played a decisive role, and events leading to the reform of the law regarding homosexuality in Britain. As with the two books mentioned in the last paragraph there are extensive endnotes with many references. Incidentally, Turing figures briefly in *Enigma* by Robert Harris (Hutchinson, 1995), a fictional account of code-breaking which as well as being a most satisfying thriller gives references which were not available when Hodges's book first appeared.

Shortly after first writing the above paragraph I discovered the Alan Turing Home Page, a truly remarkable Web site maintained by Alan Hodges. It was recently judged by a trade publication as "one of the world's top 100 websites". There is a wealth of information on Turing and his life and work, information on the numerous editions and translations of Hodges's biography, and links to numerous sites related to the history of computing. The address is *www.turing.org.uk/turing/* .

Now let us turn to some references to the subject of this paper. APL is well-served, admittedly in the technical rather than the popular sense, by the issue of the *IBM Systems Journal* (vol. 30, no. 4, 1991) which marked the twenty-fifth anniversary of APL. Of particular note are Donald McIntyre's encyclopaedic "Language as an intellectual tool: From hieroglyphics to APL" which reads almost as a hymn to Kenneth Iverson and his work with APL and what was then the new dialect **J**, and Iverson's "A personal view of APL" which gives one of the first published accounts of the evolution of APL into **J**. I have quoted briefly from this latter paper in an earlier section.

Unfortunately there appear to be few, if any, readily available papers on **J**. This situation might be best explained by the following quotation from Viscount Halsbury's Foreword to *Faster than Thought*:

> Preoccupation with the serious mental striving that is entailed in winning new knowledge leaves little time for writing popular accounts of what is being done, and the student who wishes to follow in the footsteps of the pioneers finds that he must delve through pages of highly technical literature written in a new and unfamiliar jargon *by* those who are doing the work *for* those who are doing similar creative work in the same field. Gradually, however, the strain eases, the scaffolding is removed, and it becomes possible for students to see a little of the new building. Didactic exposition, with its own problems of what to teach and how to teach it, then begins.

We certainly hope that a few glimpses may now be obtained of the new building of **J**, and a beautiful building I consider it to be, even as work on it continues. The interested reader may consult the Iverson Software Inc. Web site at

www.jsoftware.com

which has several tutorials and introductory papers and many links to related pages. My own **J** page,

called rather simply “My **J** Page”, may be found at *www.cs.ualberta.ca/~smillie/Jpage.htm* .

Conclusion

So, this is what I have been doing for so long: writing programs, writing about programs, and teaching programming. I have derived tremendous satisfaction from these activities although professional life has not been without its disappointments and frustrations. As our understanding of programming languages deepens and as changes in technology allow us to exploit this understanding more effectively, there will be new problems to solve and new ways to look at old problems. Perhaps I’ll even find something new to say about the coupon collector’s problem!

When I first thought of writing this paper, I didn't know what form it would take or just what it would contain. Now that I have come to the end of a second version of it, I am not sure of what I have accomplished or indeed whether it is finished. The best that I can say is that I am reminded of the following words by Winnie-the-Pooh after he had written a poem in honour of Piglet's bravery when Owl's house had been blown over in a storm:

> "So there it is", said Pooh, when he had sung this to himself three times "It's come different from what I thought it would, but it's come. Now I must go and sing it to Piglet."

Acknowledgements

I began the first version of this paper as an attempt to explain to my daughter, Alison, what I have been doing for most of my professional career. I certainly know she has often wondered what has kept me so busy since I retired seven years ago. I hope I have answered some of her questions. She also provided some helpful comments on an early draft for which I am grateful.

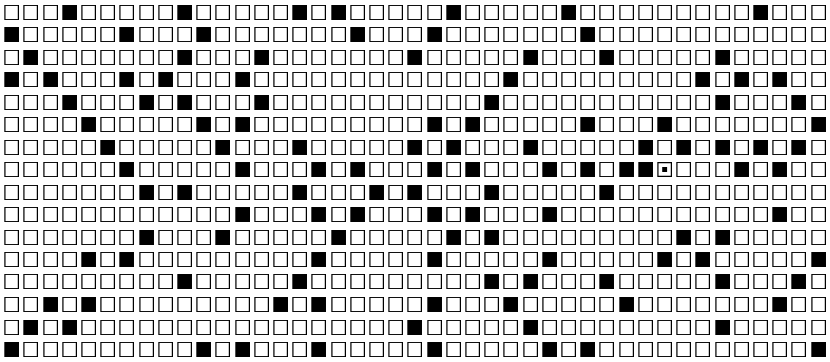
Cliff Reiter also read a draft, and, as he did with another of my papers a year or so ago, had many helpful comments which encouraged me to continue with the writing. In addition, Kenneth Iverson and Howard Peelle read copies of what was supposed to have been the final version and also made a number suggestions which were much appreciated.

Finally, I would like to thank my cat, Torako, for her companionship while I was working on this paper (and at all other times), and for not walking on the computer too often while I was writing.

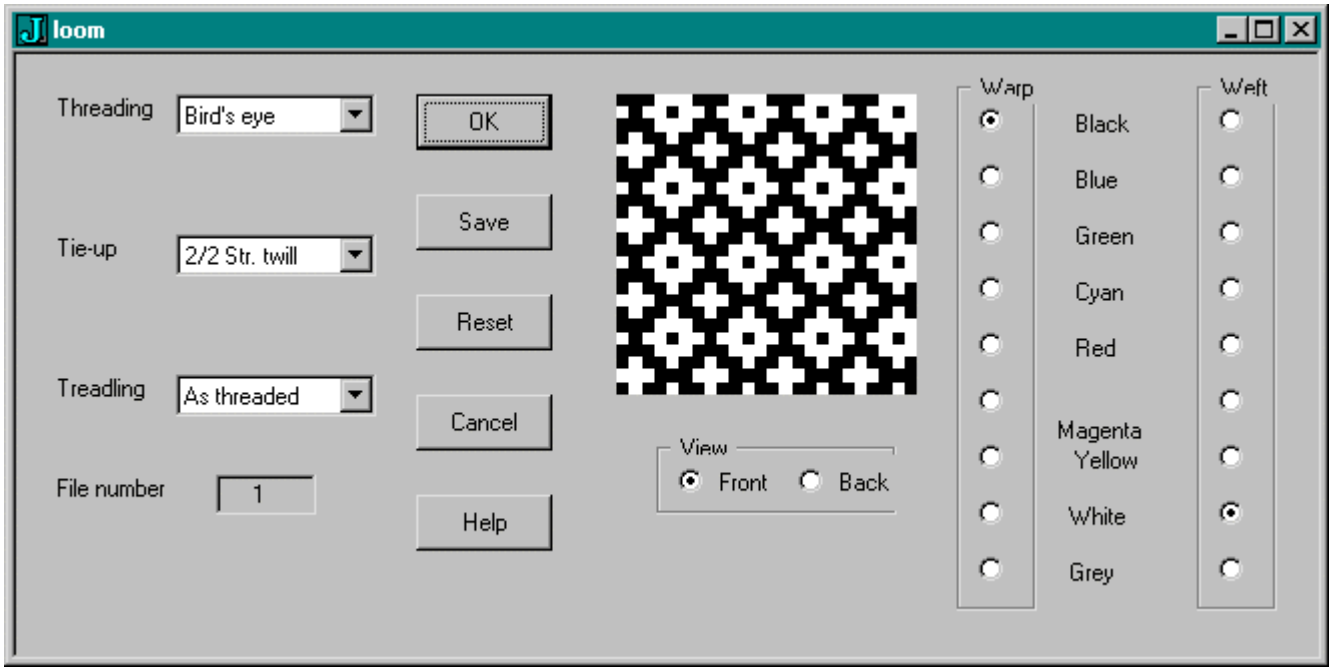
Finally, I would like to thank my cat, Torako, for her companionship while I was working on this paper (and at all other times), and for not walking on the computer too often while I was writing.

Keith Smillie is Professor Emeritus of Computing Science at the University of Alberta, Edmonton, Alberta T6G 2H1. His email address is smillie@cs.ualberta.ca.

Appendix



Coffee table spiral



Windows loom