The Icon Analyst

In-Depth Coverage of the Icon Programming Language and Applications

February 2001
Number 64

In this issue

Solving Square-Root Palindromes

In the last article on square-root palindromes [1], we investigated constant square-root palindromes (those in which all terms are the same) using a combination of programs, guesses, and deduction.

In this article, we'll take a different approach: solving continued fractions to get data from which to derive more information about square-root palindromes.

In a previous article [2], we showed a method for solving continued fractions for square roots. These continued fractions are infinite, repeating a sequence of coefficients as in

$$x = n + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2n + \cfrac{1}{\cdots}}}}}}}$$

where the subsequent coefficients are 1, 2, 3, 2, 1, 2n, repeatedly.

In order to solve such continued fractions, a closed form is needed. This is obtained by adding n to both sides of the equation.

$$x + n = 2n + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2n + \cfrac{1}{\cdots}}}}}}}$$

Now the second 2n coefficient is the same as the entire continued fraction, and $x + n$ can be substituted for it:

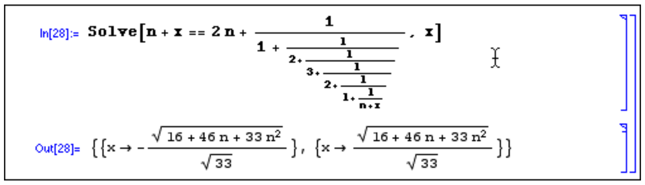
$$x + n = 2n + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{x + n}}}}}}$$

Now it's merely a matter of algebraic manipulation to solve for x. The result is an equation of the form

$$x = \pm \sqrt{n^2 + m}$$

It's the "merely" part that's the sticker. For all but trivial continued fractions, the algebra is tedious, time-consuming, and error-prone if done by hand. (Try solving the continued fraction above if you have doubts about this.)

We use *Mathematica* [3] to solve continued fractions. Figure 1 shows the *Mathematica* display for the example above.



In[28]:= $\mathrm{Solve}\left[n + x == 2n + \cfrac{1}{1 + \cfrac{1}{2. + \cfrac{1}{3. + \cfrac{1}{2. + \cfrac{1}{1. + \cfrac{1}{n + x}}}}}},\ x\right]$

Out[28]= $\left\{\left\{x \to -\frac{\sqrt{16 + 46n + 33n^2}}{\sqrt{33}}\right\}, \left\{x \to \frac{\sqrt{16 + 46n + 33n^2}}{\sqrt{33}}\right\}\right\}$

Figure 1. Mathematica Display

At the top is the continued fraction to be solved. At the bottom are the solutions; we want the positive one. Note that the coefficient of n^2 in the numerator is the same as in the denominator.

Therefore the form is

$$\sqrt{n^2 + m}$$

as we want, namely $m = (16 + 46n)/33$.

Notice that there are three parameters in this expression, as there are for other continued fractions with constant terms. The general form is

$$m = (\mathrm{a} + \mathrm{b}n)/\mathrm{c}$$

Note also that a, b, and c are the coefficients of n^0, n^1, and n^2.

In order to get a solution, m must be an integer. This is a Diophantine equation; one that must have solutions in the integers [4-6]. This particular equation is a linear Diophantine equation in the variables m and n. Rewriting, we have

$$\mathrm{c}m - \mathrm{b}n = \mathrm{a}$$

Such equations may or may not have solutions in the integers. An equation that does not have a solution in the integers is

$$10m - 6n = 1$$

This is easy to see because the left-hand side is even and the right-hand side odd. (This equation is the solution for the palindrome $\overline{3}^{\,2}$, which therefore is not a square-root palindrome.)

For linear Diophantine equations to have a solution, the greatest common divisor of b and c must evenly divide a [7]. In Icon terms, this is

```
if a % gcd(b, c) = 0 then …      #solution
```

Diophantine equations have been studied extensively. The most famous Diophantine equation appears in Fermat's Last Theorem, which states that there are no nonzero integer solutions to

$$\mathrm{a}^n + \mathrm{b}^n = \mathrm{c}^n$$

for $n > 2$.

There is an algorithm for solving linear Diophantine equations and there are methods for other special cases [8]. It is, however, known that no general solution method for Diophantine equations can exist.

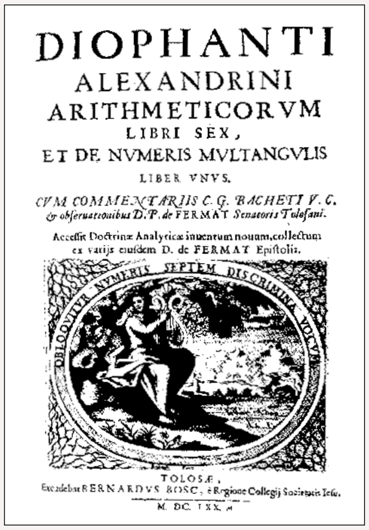
Diophantine equations are an area of number theory for which intelligent guessing and trial-and-error are recommended [4].

Although there is an algorithm for solving linear Diophantine equations, we'll have to deal

Diophantus

Diophantus is a somewhat mysterious figure in the annals of mathematics. Unlike most mathematicians who have had a major impact, the time Diophantus lived is not known. Based on sources he cited and sources that cited him, he could have lived anywhere from the second century B.C. to the fourth century A.D. Most authorities place his birth at around 200 A.D. It was known that he was a Greek who lived in Alexandria, the great center of mathematical learning.

Diophantus, often called the father of algebra, was remarkable in that his form of mathematical thought was entirely different from others of his time. He invented new mathematical notations and wrote extensively, although much of his work is lost.



DIOPHANTI
ALEXANDRINI
ARITHMETICORVM
LIBRI SEX,
ET DE NVMERIS MVLTANGVLIS
LIBER VNVS.

CVM COMMENTARIIS C. G. BACHETI V. C.
& obſeruationibus D. P. *de* FERMAT *Senatoris Toloſani.*

Acceſſit Doctrinæ Analyticæ inuentum nouum, collectum
ex varijs eiuſdem D. de FERMAT Epiſtolis.

OBLOQVITVR NVMERIS SEPTEM DISCRIMINA VOCVM

TOLOSÆ,
Excudebat BERNARDVS BOSC, è Regione Collegij Societatis Iesu.
M. DC. LXX.

with more complicated equations later on, and hence we will use the trial-and-error method (without intelligent guessing).

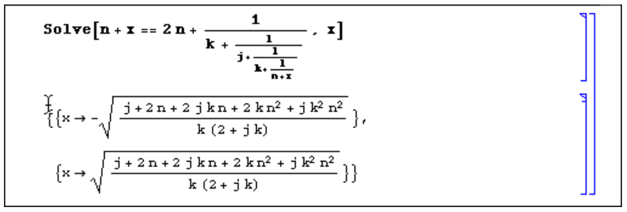
The basic code for solving linear Diophantine equations by trial-and-error is:

```
           …
  if a % gcd(b, c) ~= 0 then fail

  every i := seq() do {
     k := a + b * i
     if n % c ~= 0 then next        # no solution yet
     m := k / c                     # first solution
     break
     }
           …
```

Although it's known that if the divisibility condition is satisfied, there is a solution, it's downright scary to write an endless loop. Furthermore, the solution may be *very* far out. Defensive programming suggests that there should be a limit on the loop.

In the case of general Diophantine equations in which there is no test to assure a solution, the situation is, of course, even more problematical.

Linear Diophantine equations result from continued fractions in which all the coefficients are constants. If the coefficients are variable, the results are more complex, as shown in Figure 2.



Solve[n + x == 2 n + $\dfrac{1}{k + \dfrac{1}{j + \dfrac{1}{k + \dfrac{1}{n+x}}}}$, x]

$\left\{\left\{x \to -\sqrt{\dfrac{j + 2 n + 2 j k n + 2 k n^2 + j k^2 n^2}{k (2 + j k)}}\right\},\right.$

$\left.\left\{x \to \sqrt{\dfrac{j + 2 n + 2 j k n + 2 k n^2 + j k^2 n^2}{k (2 + j k)}}\right\}\right\}$

Figure 2. A Continued Fraction with Variables

We'll return to this later. First we'll explore constant square-root palindromes, in which all the coefficients are constant and the same.

Constant Square-Root Palindromes

These palindromes were explored in the previous article on square-root palindromes [1]. Here we'll see what comes from looking at the problem in a slightly different way — in terms of the a, b, and c coefficients described in the previous section.

We'll start by looking at some coefficients for $\sqrt{j^k}$, j odd:

$j = 1$:	k	a	b	c
	1	0	2	1
	2	1	2	2
	3	1	4	3
	4	2	6	5
	5	3	10	8
	6	5	16	13
	7	8	26	21

$j = 3$:	k	a	b	c
	1	0	2	3
	2	1	6	10
	3	3	20	33
	4	10	66	109
	5	33	218	360
	6	109	720	1189
	7	360	2378	3927

$j = 5$:	k	a	b	c
	1	0	2	5
	2	1	10	26
	3	5	52	135
	4	26	270	701
	5	135	1402	3640
	6	701	7280	18901
	7	3640	37802	98145

Examination of this data suggests the following formulas:

$$a_k = b_{k-1}/2$$
$$b_k = 2c_{k-1}$$
$$c_k = jc_{k-1} + c_{k-2}$$

If these formulas are correct, all we need to compute values for any k are the values for $k = 1$ and 2. Here they are:

$a_1 = 0$	$b_1 = 2$	$c_1 = j$
$a_2 = 1$	$b_2 = 2j$	$c_2 = j^2 + 1$

Here is a procedure for generating continued-fraction solutions for $\sqrt{j^k}$, k odd:

```
procedure oddson(j)
   local alist, blist, clist, i

   alist := [0, 1]
   blist := [2, 2 * j]
   clist := [j, j ^ 2 + 1]

   suspend "a=" || alist[1] || ", b=" || blist[1] ||
      ", c=" || clist[1]
   suspend "a=" || alist[2] || ", b=" || blist[2] ||
      ", c=" || clist[2]

   repeat {
      put(alist, blist[–1] / 2)
      put(blist, 2 * clist[–1])
      put(clist, j * clist[–1] + clist[–2])
      suspend "a="  || alist[–1] || ", b=" ||
         blist[–1] || ", c=" || clist[–1]
      get(alist)              # remove debris
      get(blist)
      get(clist)
      }

   end
```

Next, we'll look at $\overline{j}^{k}$, j even:

$j = 2$:	k	a	b	c
	1	0	1	1
	2	1	4	5
	3	1	5	6
	4	5	24	29
	5	6	29	35
	6	29	140	169
	7	35	169	204

$j = 4$:	k	a	b	c
	1	0	1	2
	2	1	8	17
	3	2	17	36
	4	17	144	305
	5	36	305	646
	6	305	2584	5473
	7	646	5473	11592

$j = 6$:	k	a	b	c
	1	0	1	3
	2	1	12	37
	3	3	37	114
	4	37	456	1405
	5	114	1405	4329
	6	1405	17316	53353
	7	4329	53353	164388

Examination of this data suggests the following formulas:

$$
\begin{aligned}
\mathrm{a}_k &= \mathrm{c}_{k-2} \\
\mathrm{b}_k &= \mathrm{c}_{k-1} && k \text{ odd} \\
&= 4\mathrm{c}_{k-1} && k \text{ even} \\
\mathrm{c}_k &= j\mathrm{c}_{k-1}/2 + \mathrm{c}_{k-2}
\end{aligned}
$$

Again, assuming these formulas are correct, all we need to compute values for any k are the values for $k = 1$ and 2. Here they are:

$$
\begin{array}{lll}
\mathrm{a}_1 = 0 & \mathrm{b}_1 = 1 & \mathrm{c}_1 = j/2 \\
\mathrm{a}_2 = 1 & \mathrm{b}_2 = 2j & \mathrm{c}_2 = j^2 + 1
\end{array}
$$

Here is a procedure for generating continued-fraction solutions for $\overline{j}^{k}$, *k even*:

```
   procedure evenson(j)
      local alist, blist, clist, i, count

      alist := [0, 1]
      blist := [1, 2 * j]
      clist := [j / 2, j ^ 2 + 1]

      suspend "a=" || alist[1] || ", b=" || blist[1] ||
         ", c=" || clist[1]
      suspend "a=" || alist[2] || ", b=" || blist[2] ||
         ", c=" || clist[2]

      count := 2

      repeat {
         count +:= 1
         put(alist, clist[–2])
         if count % 2 = 1 then
            put(blist, clist[–1])
         else
            put(blist, 4 * clist[–1])
         put(clist, k / 2 * blist[–1] + clist[–2])
         suspend "a="  || alist[–1] || ", b=" ||
            blist[–1] || ", c=" || clist[–1]
         get(alist)                    # remove debris
         get(blist)
         get(clist)
         }

   end
```

The next question is, given the values for a, b, and c in the solution of a continued fraction, how do we get formulas for n and m in $\sqrt{n^2 + m}$?

From the previous article, we know that for j odd, the formulas have the form

$$
\begin{aligned}
n &= \mathrm{A}i - \mathrm{B} \\
m &= \mathrm{C}i - \mathrm{D}
\end{aligned}
$$

(We've used uppercase letters to avoid confusion with the parameters a, b, and c.)

Thus, successive values of n differ by A and successive values of m differ by C. Inspection shows that A = c and C = b. If I is the value of the iteration variable that gives the first solution in the trail-and-error method of solving Diophantine equations, then B = c – I and D = (a + bI)/c. Murky? Trust us.

From the previous article, we also know that for j even, the formulas have the form

$$
\begin{aligned}
n &= \mathrm{E}i + j/2 \\
m &= \mathrm{F}i + 1
\end{aligned}
$$

Here also, E = c and F = b. Trust us.

Here are procedures to produce the formulas for j odd and j even:

```
   procedure mnodd(a, b, c)
      local i, n, m
```

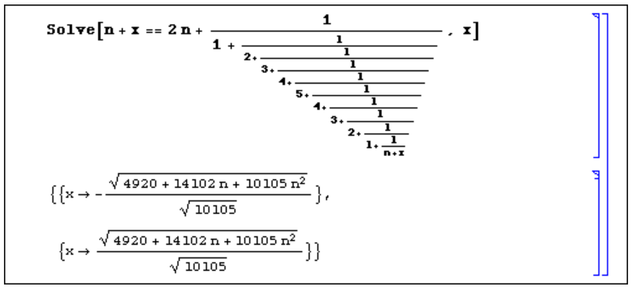
```
  if a % gcd(b, c) ~= 0 then fail

  every i := seq() do {          # scary … be a cowboy
    n := a + b * i
    if (n % c) ~= 0 then next
    m := n / c
    write("n:=", c, "*i–", c – i)
    write("m:=", b, "*i–", b – m)
    exit()
    }

end

procedure mneven(a, b, c, j)
  local i, n, m

  if a % gcd(b, c) ~= 0 then fail

  every i := seq() do {          # scary … be a cowboy
    n := a + b * i
    if (n % c) ~= 0 then next
    m := n / c
    write("n:=", c, "*i+", j / 2)
    write("m:=", b, "*i+1")
    exit()
    }

end
```

Note that mneven() requires *j* as an argument, while mnodd() does not.

We've made a lot of conjectures along the way. The good news is that the procedures above produce verifiably correct results for a wide range of values.

Other Palindromes with Constant Coefficients

We can apply the methods above to other specific square root palindromes, such as 1, 2, 3, 4, 5, 4, 3, 2, 1 as shown in Figure 3.



$$\text{Solve}\left[n + x == 2\,n + \cfrac{1}{1 + \cfrac{1}{2. + \cfrac{1}{3. + \cfrac{1}{4. + \cfrac{1}{5. + \cfrac{1}{4. + \cfrac{1}{3. + \cfrac{1}{2. + \cfrac{1}{1. + \cfrac{1}{n+x}}}}}}}}}},\ x\right]$$

$$\left\{\left\{x \to -\frac{\sqrt{4920 + 14102\,n + 10105\,n^2}}{\sqrt{10105}}\right\},\ \left\{x \to \frac{\sqrt{4920 + 14102\,n + 10105\,n^2}}{\sqrt{10105}}\right\}\right\}$$

Figure 3. The Palindrome 1, 2, 3, 4, 5, 4, 3, 2, 1

This palindrome has the equations

$$n = 10105i - 5280$$
$$m = 14102i - 7368$$

One of the problems with using *Mathematica* is entering and editing complicated expressions. This can be mechanized by creating the text form *Mathematica* uses internally.

The *Mathematica* text form for the solution of the palindrome 1, 2, 3, 4, 5, 4, 3, 2, 1 is

```
\!\(Solve[ n + x == 2\ n +
     1\/\(1 +
        1\/\(2 +
           1\/\(3 +
              1\/\(4 +
                 1\/\(5 +
                    1\/\(4 +
                       1\/\(3 +
                          1\/\(2 +
                             1\/\(1 +
                                1\/\(n + x
\)\)\)\)\)\)\)\)\)\), x]\)
```

This is hardly a pretty sight, but internal data formats rarely are.

Experiment shows that blanks and newlines are optional. It's not necessary to understand the syntax (as far as we know, it's not documented). For our purposes, it's enough to see what the basic segment for a coefficient is; the rest can be copied without understanding it.

Here's a program that outputs *Mathematica* text for solving a square-root palindrome, which is input one coefficient per line:

```
procedure main()
  local head, pre, post, nfact, tail, count, k

  head := "\\!\\(Solve[ n + x == 2\\ n + "
  pre := "1\\/\\("
  post := " +"
  nfact := "1\\/\\(n + x\\)"
  tail := ", x]\\)"

  writes(head)

  count := 0

  while k := read() do {
    writes(pre, k, post)
    count +:= 1
    }

  writes(nfact)
  writes(repl("\\)", count))        # closing parentheses
  write(tail)

end
```

All that is necessary is to run this program with the desired palindrome (the program does not check that it *is* a palindrome) and import the result into *Mathematica*.

There probably is some limit to the size of continued fractions *Mathematica* can handle, if only the amount of memory available to it. We, however, have not exceeded this limit so far.

Note that the input to this program need not consist of integers. This allows for entering palindromes with variable coefficients, such as *k*, *j*, *k*, *j*, *k*, *j*, *k*.

Next Time

As you can guess, the next article on square-root palindromes will deal with coefficients that are variables. This will lead us to complex Diophantine equations and a variety of programming challenges.

References

1. "Constant Square-Root Palindromes", Icon Analyst 63, pp. 1-7.

2. "Continued Fractions for Quadratic Irrationals", Icon Analyst 61, pp. 9-15.

3. *The Mathematica Book*, Stephen Wolfram, Cambridge University Press, 3rd ed., 1996.

4. *Continued Fractions*, C. D. Olds, Mathematical Association of America, 1963.

5. *Diophanus and Diophantine Equations*, Isabella Grigoryevna Bashmakova, Mathematical Association of America, 1997.

6. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1999, pp. 437-439

7. *Number Theory*, George E. Andrews, Dover, 1971, p. 23.

8. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1999, pp. 446-447.

Periodic Sequence Curios

In reading about periodic sequences, we've come across a couple of curiosities that we'd like to pass on to you.

A Sequence with A Curious Period Pattern

Given the sequence of powers of two, 2, 4, 8, 16, 32, 64, 128, 256, ... the units digits cycle with the pattern 2, 4, 8, 6.

The tens digits are periodic also, as are the hundreds digits, and so on.

The period is always $4 \times 5^{n-1}$, where *n* is the position of the digit from the *right*. Notice the period increases very rapidly with *n*.

There are other patterns in the powers-of-two sequence. See Reference 1.

A Sequence Whose Period is Impervious to the Values its Initial Terms

The following recurrence is due to Morton Brown:

$$x_n = |x_{n-1}| - x_{n-2}$$

The sequence it produces is claimed to have period 9 for any initial values for x_1 and x_2 — even real values. (This is not true if both x_1 and x_2 are 0.)

Here's a procedure; give it a try.

```
procedure brown9(x1, x2)
   local t

   repeat {
      t := abs(x2) – x1
      suspend t
      x1 := x2
      x2 := t
      }

end
```

Can you prove the conjecture (aside from the problem with zero initial values)? We've seen a proof, and it is not elementary.

Reference

1. *Excursions in Number Theory*, C. Stanley Ogilvy and John T. Anderson, Dover, 1966, pp. 89-90.

Understanding T-Sequences II

In the first article on understanding T-sequences [1], we showed some simple examples and developed a set of operations that could be used to describe and construct these sequences, For reference, here are those operations:

$i \xrightarrow{k} j$	*run*
$\overline{S}^{\,i}$	*repeat*
$S \Rightarrow i$	*extension*
$S\,,T$	*concatenation*
$\ulcorner\overline{S}$	*open palindrome*
$\ulcorner\overline{S}^{\,i}$	*closed palindrome*
$\overset{\leftrightarrow}{S}$	*horizontal reflection*
$\updownarrow S$	*vertical reflection*

In this article, we'll look at a few T-sequences that are more complicated and add operations for describing and constructing them.

Before we do that, however, there is a generalization of one of the operations above that we want to introduce.

Connected Runs

In many T-sequences, there are connected runs between inflection points at which the runs change from up to down or vice versa. Figure 1 shows an example.

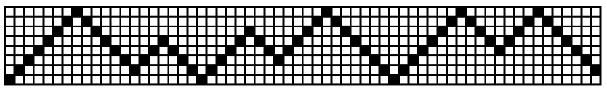


Figure 1. A Connected Run

The sequence in this figure can be described as the concatenation of runs or, in places, palindromes. Here is a description of this sequence in terms of concatenated runs.

$$1 \rightarrow 8, 7 \rightarrow 2, 3 \rightarrow 5, 4 \rightarrow 1, 2 \rightarrow 6, 5 \rightarrow 3,$$
$$4 \rightarrow 8, 7 \rightarrow 1, 2 \rightarrow 8, 7 \rightarrow 4, 5 \rightarrow 8, 7 \rightarrow 2$$

The problem with using concatenated runs is that it obscures the structure of the sequence and requires arbitrary decisions as to whether inflection points belong to upward runs or downward runs. For example, the two runs at the beginning could be

$$1 \rightarrow 8, 7 \rightarrow 2$$

or

$$1 \rightarrow 7, 8 \rightarrow 2$$

A better method, which captures the structure of the sequence, is to use the inflection points in a connected run operation

$$1 \rightarrow 8 \rightarrow 2$$

Although this form is descriptive and easily understood, there are technical problems with a binary operator used in this way. For example, if the operators group as $(1 \rightarrow 8) \rightarrow 2$, $(1 \rightarrow 8)$ produces a sequence. How can this be the first operand for the second operator?

An n-ary operation is more appropriate, as in

$$\rightarrow (1, 8, 2)$$

This form has the advantage of emphasizing the fact that the argument is a sequence. We will continue to use the notation $i \rightarrow j$, although it means the same thing as $\rightarrow (i, j)$

The sequence in Figure 1 then can be described by

$$\rightarrow (1, 8, 2, 5, 1, 6, 3, 8, 1, 8, 4, 8, 2)$$

Here's another example from the previous article:

$$\overline{\ulcorner(8 \rightarrow 5)\urcorner}, 7 \rightarrow 2, \overline{\ulcorner(1 \rightarrow 4)\urcorner}$$

Using connected runs, this sequence can be described by

$$\rightarrow (8, 5, 8, 1, 4, 1)$$

We think connected runs better capture the essence of the sequence than the concatenation of palindromes and runs. Furthermore, connected runs clearly show palindromes.

We have not made provisions for an increment other than 1 for connected runs. We have never seen an example, and the complications of specifying an increment seem out of proportion to their potential value.

More Examples

Figure 2 shows six T-sequences with structures that cannot be described adequately by the operations developed so far. Some of these sequences are too long to allow a full repeat to be shown. Ellipses indicate this.

Motif Along a Path

It is evident that the sequence in Figure 2a consists of the concatenation of instances of a short sequence offset by different amounts. We'll call the short sequence a motif, although in general, motifs need not be short.

The motif in Figure 2a can be described in several ways, including

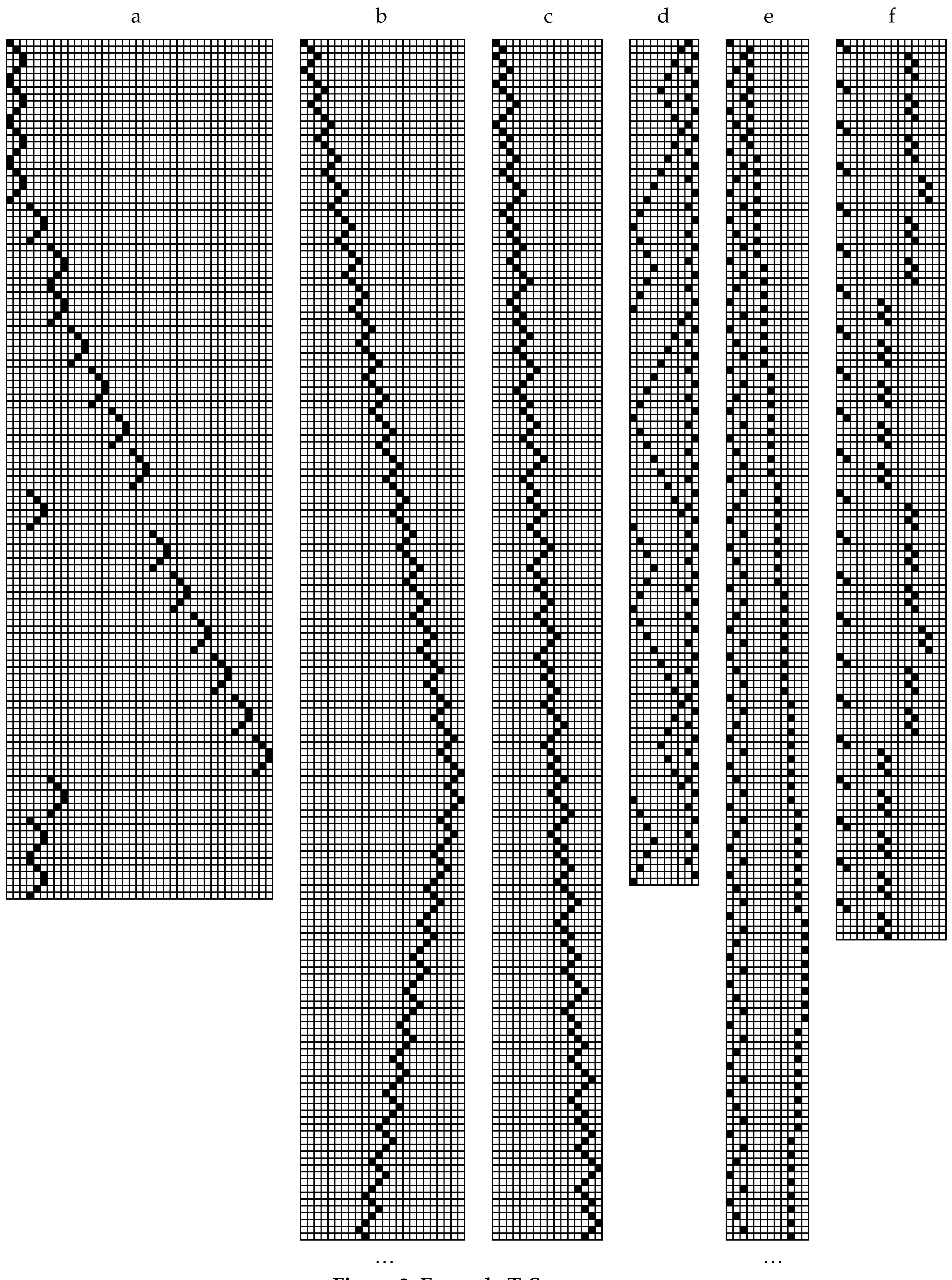


Figure 2. Example T-Sequences

$M = 1 \rightarrow 3, 3 \rightarrow 1$

To describe concatenations of a motif at various offsets, we'll use the notation

$M @ P$

where P is a path, a sequence of offsets at which M is placed.

The path for Figure 2a is

$$P = \overline{1}^4, 4, \overline{7}^2, 10 \overset{3}{\rightarrow} 19, 4, 22 \overset{3}{\rightarrow} 37, 7, \overline{4}^2$$

This path is shown in Figure 3.

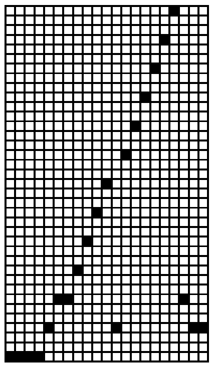


Figure 3. Path for the Motif in Figure 2a

Notice that the differences between successive different positions P are multiples of 3, the bound on M. The basic path can be written as

$$P' = \overline{1}^4, 2, \overline{3}^2, 4 \rightarrow 7, 2, 8 \rightarrow 13, 3, \overline{2}^2$$

When P' is *scaled* by $\beta(M)$ the result is P.

Scaling for T-sequences is not simply a matter of multiplying the terms by the scaling factor, since T-sequences are 1-based, not 0-based as is ordinary arithmetic. A term i is scaled by k as follows:

$i' = k(i - 1) + 1$

We'll use the notation

$S \times i$

for scaling a sequence.

Scaling has the effect of simplifying the structure of sequences such as the one in Figure 2a, which may be written as

$M @ (P' \times \beta(M))$

The T-sequence in Figure 2b also is a motif along a path. It's a bit more difficult to identify its motif than it is for the one in Figure 2a, although it is simpler than the one in Figure 2a:

$M = \rightarrow (1, 3, 1)$

Aside: We are relying here on visual analysis to determine the structure of T-sequences. Our ultimate goal is to determine T-sequence structure automatically. We have that planned for the final issue of the Analyst.

The path in Figure 2b is a simple run reflected to form a palindrome (the entire sequence is too long to show in Figure 2b):

$\rightarrow (1, 25, 1)$

The entire sequence can be given by

$S = \overline{(\rightarrow(1, 3, 1)) @ (1 \rightarrow 25)}$

Notice that this is different from

$T = (\rightarrow(1, 3, 1)) @ (\rightarrow (1, 25, 1))$

because of the handling of the term at the point of reflection in a palindrome.

Figure 2c also is a motif along a path, although the motif is more difficult to identify than in the preceding cases. It is

$M = \rightarrow (1, 2, 1, 3, 1, 4, 1)$

The path is again a simple run,

$P = 1 \rightarrow 13$

so the entire sequence is simply

$S = \rightarrow(1, 2, 1, 3, 1, 4, 1) @ (1 \rightarrow 13)$

Collation

The argument of the connected run in the motif in the preceding example,

1, 2, 1, 3, 1, 4, 1

also can be viewed as the *collation* or interleaving of the terms of two sequences,

$\overline{1}^3$

and

$2 \rightarrow 4$

We'll use the notation

$S \sim T$

for the collation of S and T. In the result, the first term comes from S, the second from T, the third from S, and so on cyclically, stopping when one of the sequences runs out. When terms are taken one by one from sequences in order, the collation is *simple*. We'll take up more complex collations in the next article on T-sequences.

Therefore the sequence above can be written as

$$\overline{1}^3 \sim (2 \rightarrow 4)$$

The simple collation of several sequences is given by

$$\sim(S_1, S_2, S_3, \ldots, S_n)$$

The first term of the result comes from S_1, the second from S_2, and so, continuing cyclically after S_n with S_1, and so on. The process stops when any sequence runs out.

Collation plays an important part in the structure of many T-sequences and is the main subject of the remaining examples.

In Figure 2d, it is easy to see that there are two distinct sequences, a lower one, L, on shafts 1 through 8 and upper one, U, on shafts 9 and 10. It's also clear that terms from these two sequences alternate. The entire sequence is

$$S = U \sim L$$

It remains to determine L and U with L consisting of the odd-numbered terms and U the even. U is easy:

$$U = \overline{9, 10}^{38}$$

There is not a problem if there are too many repeats, since the collation will terminate when L does and give the correct length.

L is considerably more complicated. It is shown in Figure 4.

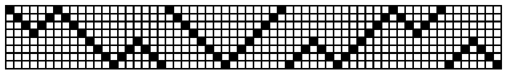


Figure 4. The Lower Sequence in Figure 2d

This sequence can be described by

$$T = \rightarrow(8, 5, 8, 1, 4, 1)$$
$$V = \rightarrow(8, 1)$$
$$W = \rightarrow(1, 4, 1)$$
$$L = \ulcorner\overline{T, V}\urcorner, \mathrm{W}$$

The sequence in Figure 2e also is a simple collation. The upper sequence, U, can be considered to be a motif along a path:

$$U = \overline{1}^8 \mathbin{@} (\rightarrow (4, 12, 4))$$

(the complete palindrome is too long to show in Figure 2e).

The lower sequence is a repeat,

$$L = \overline{\overline{\ulcorner 1, (3 \rightarrow 1), 3}}$$

and the entire sequence is

$$S = L \sim U$$

The sequence U can be considered in another way as 8 replications of each term in $\rightarrow (4, 12, 4)$.

We'll use the notation

$$\underline{S}_i$$

to indicate the sequence resulting from i replication of each term in S.

In this view,

$$U = \underline{(\rightarrow (4, 12, 4))}_8$$

Note that although adjacent duplicate terms may cause structural problems in weaving, in this case the duplicate terms in U subsequently are separated by collation.

Figure 2f can be viewed in several ways. One way is as a collation of a sequence on shafts 1 and 2 with another sequence on the remaining shafts. (Note that some shafts are unused. This is a common occurrence when design is done for a loom that has more shafts than are needed.)

Another view of the sequence is as motif

$$M = 1, 2$$

along the path illustrated in Figure 5.

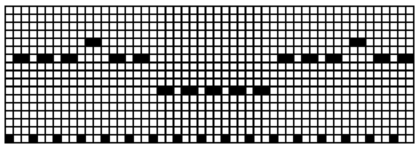


Figure 5. The Path in Figure 2f

The path is a repeat of a collation

$$R = \sim(L, U, U)$$

where

$$L = \overline{1}^{17}$$

and

$$U = \overline{11}^3, 13, \overline{11}^2, \overline{7}^5$$

This in turn can be viewed as

$$U = (\overline{6}^3, \overline{7}, \overline{6}^2, \overline{4}^5) \times \beta(M)$$

So the entire sequence can be composed from

$M = 1, 2$
$R = \sim(L, U, U)$
$S = M @ \overline{R}^2$

Summary

In this article we've introduced the following operations on T-Sequences:

$\rightarrow S$	*connected runs*
$M @ P$	*motif along a path*
$S \times i$	*scaling*
$\sim(S_1, S_2, S_3, \ldots, S_n)$	*collation*
$\underline{S}_i$	*term repetition*

Implementing the Operations

Here are procedures that implement these operations. They make use of some of the procedures in Reference 1.

```
procedure srun(args[ ])
  local lseq, i, j

  lseq := []

  i := get(args) | return lseq

  while j := get(args) do {
    lseq |||:= sruns(i, j)
    pull(lseq)
    i := j
    }

  put(lseq, i)

  return lseq

end

procedure sruns(i, j, k)
  local lseq

  /k := 1
  if j < i then k:= −k

  lseq := []

  every put(lseq, i to j by k)

  return lseq

end
```

Note that srun() also handles the case of just two arguments provided the increment is 1.

This procedure implements simple collation:

```
procedure scollate(args[ ])
  local lseq, i

  args := copyargs ! args

  lseq := []

  repeat
    every i := 1 to *args do
      put(lseq, get(args[i])) | break break

  return lseq

end
```

The procedure copyargs() copies a list while promoting its elements to lists if necessary:

```
procedure copyargs(args[])
  local new_args

  new_args := []

  every put(new_args, copy(spromote(!args)))

  return new_args

end
```

Here's a procedure to place a motif along a path:

```
procedure splace(x1, x2)
  local lseq, i

  x1 := copy(spromote(x1))

  x2 := spromote(x2)

  lseq := []

  every i := !x2 do
    every put(lseq, !x1 + i – 1)

  return lseq

end
```

The following procedure performs scaling:

```
procedure sscale(x, i)
  local lseq, j

  lseq := copy(spromote(x))

  every j := 1 to *lseq do
    lseq[j] := (lseq[j] – 1) * i + 1

  return lseq

end
```

Finally, this procedure implements term repetition:

```
procedure srepl(x, i)
  local lseq, j

  lseq := []
```

```
   every j := !spromote(x) do
      every 1 to i do
         put(lseq, j)

   return lseq

end
```

Procedures for Constructing the Example Sequences

The following procedures construct the T-sequences shown in Figure 2.

```
procedure fig2a()
   local M, P

   M := sconcat(srun(1, 3), srun(3, 1))

   P := sconcat(
      srepeat(1, 4),
      4,
      srepeat(7, 2),
      sruns(10, 19, 3),
      4,
      sruns(22, 37, 3),
      7,
      srepeat(4, 2)
      )

   return splace(M, P)

end

procedure fig2b()
   local M, P

   M := srun(1, 3, 1)

   P := srun(1, 25)

   return scpal(splace(M, P))

end

procedure fig2c()
   local M, P

   M := srun(1, 2, 1, 3, 1, 4, 1)

   P := srun(1, 13)

   return splace(M, P)

end

procedure fig2d()
   local  U, T, V, W, L

   U := srepeat(srun(9, 10), 38)

   T := srun(8, 5, 8, 1, 4, 1)

   V := srun(8, 1)

   W := srun(1, 4, 1)

   L := sconcat(
      scpal(sconcat(T, V)),
      W
      )

   return scollate(U, L)

end

procedure fig2e()
   local L, U

   U := sdupl(srun(4, 12, 4), 8)

   L := srepeat(sopal(sconcat(1, srun(3,1), 3)), 25)

   return scollate(L, U)

end

procedure fig2f()
   local R, L, U, M

   L := srepeat(1, 11)

   M := sconcat(1, 2)

   U := sscale(
      sconcat(
         srepeat(6, 3),
         7,
         srepeat(6, 2),
         srepeat(4, 5)
         ),
      sbound ! M
      )

   R := scollate(L, U, U)

   return splace(M, srepeat(R, 2))

end
```

Next Time

We are not finished with collation — not by a long shot. In the next article on T-sequences, we'll generalize collation to allow the specification of how terms are taken from different sequences.

We'll also make use of the fact that many collations, such as the ones in this article, are on disjoint sets of shafts.

In addition, we will explore the generalization of integer operands to sequences.

Reference

1. "Understanding T-Sequences", Icon Analyst 63, pp. 1-7.

Graphics Corner — Image Permutations

Editors' Note: This article was inspired by an article by Bill Jones for the Complex Weavers Mathematics and Textiles Study Group [1].

Permutations

A permutation is a rearrangement in the order of *n* distinct objects. The objects may be anything, but for notational and programming convenience, it is easiest to deal with the positive integers from 1 to *n*. These integers can be considered as labels for the actual objects.

Note: In mathematical parlance, a *transposition* is the exchange in position of two objects. All permutations can be achieved by a sequence of transpositions. In view of this difference in terminology, transposition ciphers [2] might better be called permutation ciphers.

Several different notations are used for describing permutations. A commonly used one has two lines, the first to label the positions and the second to indicate what objects go into those positions, as in

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 2 & 6 & 5 & 4 \end{pmatrix}$$

Thus object 3 goes to position 1, object 1 to position 2, object 2 to position 3, and so on.

The positions are implicit in the permutation and we can omit the positions and just give this permutation as 3 1 2 6 5 4.

The *n*! permutations of length *n* form the *symmetric group* S_n. The *identity* permutation for this group is one that does not change the order of the objects to which it applies. In our notation, it is just 1 2 3 … *n*.

The *period* of a permutation is number of successive applications of it to a sequence to get back to where the original sequence. For the example above, if we start with the objects in numerical order, the permutation applied successively produces

```
1 2 3 4 5 6     original
3 1 2 6 5 4     1
2 3 1 4 5 6     2
1 2 3 6 5 4     3
3 1 2 4 5 6     4
2 3 1 6 5 4     5
1 2 3 4 5 6     6
```

So the period of this permutation is 6. It doesn't matter what the original order is — the period is always 6. Here, for example, are the results starting with the permutation itself.

```
3 1 2 6 5 4     original
2 3 1 4 5 6     1
1 2 3 6 5 4     2
3 1 2 4 5 6     3
2 3 1 6 5 4     4
1 2 3 4 5 6     5
3 1 2 6 5 4     6
```

Permutations can be factored into *cycles* of objects that permute together independently of other objects. For example, the permutation above has cycles 3 1 2, 6 4, and 5. This is written

(3 1 2) (6 4) (5)

The period of a permutation is the least common multiple of the periods of its cycles. The cycle (3 1 2) has period 3, the cycle (6 4) has period 2, and the cycle (5) has length 1, so the period of the permutation is *lcm*(1, 2, 3) = 6.

The cyclic form of the identity permutation is (1) (2) (3) … (*n*) and, of course, it has period 1.

Every permutation has an *inverse* permutation that undoes its result. The inverse of 3 1 2 6 5 4 is 2 3 1 6 5 4. Note that 6 4 is its own inverse. More generally, any cycle with period 2 is its own inverse.

Implementation

The natural data structure for representing permutations is the list. Here's a procedure that produces the identity permutation:

```
procedure pident(i)
  local p

  p := []

  every put(p, 1 to i)

  return p

end
```

Note that the result is the same as for srun(1, i) from T-sequences [3].

The following procedure applies a permutation:

```
procedure permute(objects, p)
  local result
```

```
   result := []

   every put(result, objects[!p])

   return result

end
```

The inverse of a permutation can be obtained as follows:

```
procedure pinvert(p)
   local inverse, i

   inverse := list(*p)

   every i := 1 to *p do
      inverse[p[i]] := i

   return inverse

end
```

The cycles of a permutation can be computed by starting with any value in the permutation and collecting the values it leads to until the original value is found. These values are removed along the way and the process continues until there is nothing left.

```
procedure cycles(p)
   local indices, cycle, cycles, i

   cycles := []              # list of cycles

   indices := set()

   every insert(indices, 1 to *p)

   repeat {
      i := !indices | break
      delete(indices, i)
      cycle := [i]
      repeat {
         i := integer(p[i])
         delete(indices, i)
         if i = !cycle then break    # done with cycle
         else put(cycle, i)          # new member of cycle
         }
      put(cycles, cycle)
      }

   return cycles

end
```

Notice that a packet sequence is returned [4].

From this, it is easy to compute the period of a permutation:

```
procedure permperiod(p)
   local lengths

   lengths := []

   every put(lengths, *!cycles(p))

   return lcml ! lengths

end
```

The procedure lcml() is from the Icon program library module numbers:

```
procedure lcml(L[])
   local i, j

   i := get(L) | fail

   while j := get(L) do
      i := lcm(i, j)

   return i

end
```

where lcm() is from the same module:

```
procedure lcm(i, j)

   if (i =  0) | (j = 0) then return 0

   return abs(i * j) / gcd(i, j)

end
```

The following procedures generate all the permutations of *n* objects in lexicographical order:

```
procedure permutations(i)

   suspend permutations_(pident(i))

end

procedure permutations_(p)
   local i

   if *p = 0 then return []

   suspend [p[i := 1 to *p]] |||
      permutataions_(p[1:i] ||| p[i+1:0])

end
```

MultiReduce

The permutation that inspired this article is called MultiReduce. MultiReduce uses permutations of the columns and rows of an image to "quarter" it. First, all the even-numbered columns are moved to the left half, while all the odd-numbered columns are moved to the right half. For an image 20 columns wide, the permutation is

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

Then the same thing is done to the rows to move the even-numbered row to the top half and the odd-numbered rows to the bottom half.

Figure 1 shows an example image and the result of applying MultiReduce to it:

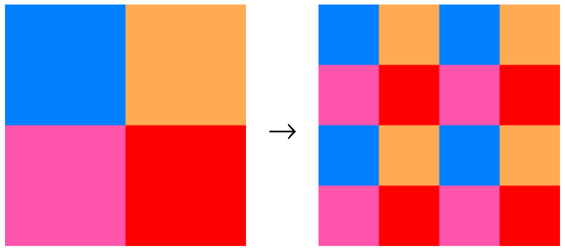


Figure 1. MultiReduce Applied to an Image

The process then is repeated, again and again. As the permutation is applied repeatedly, the image gets sliced and diced to the point that the pattern becomes very fine-grained.

Figure 2 shows the images after 19 and 20 applications of MultiReduce.

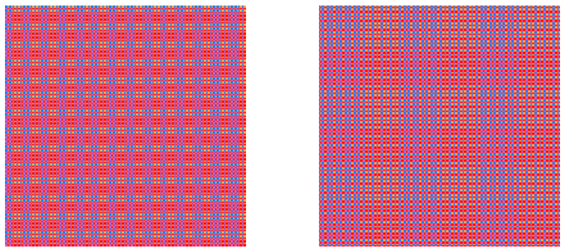


Figure 2. Further Applications of MultiReduce

No pixels are discarded, however; they are just rearranged. Since all permutations have a period, repeated application must eventually produce the original image. The image above is square and hence the period for the combined horizontal and vertical permutations is the same as the period for the horizontal/vertical permutation by itself. In this case, the period is 40. Figure 3 shows the last two permutations before the original image reappears.

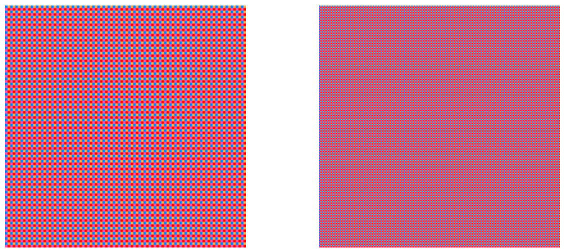


Figure 3. The Last Two Permutations

It seems remarkable that the next image should be the same as the original.

Choosing Images for MultiReduce

Simple images generally produce more interesting results than complex ones, although there are exceptions.

Generally speaking, photo-realistic images with many different colors do not produce interesting results. What seems to matter most is the degree of organization in the image — a concept that eludes precise definition but nonetheless is understandable. For example, images composed completely at random, which have little aesthetic appeal, produce more of their kind. (If they don't, the original image wasn't really random.) On the other hand, the application of MultiReduce to symmetric images often produces attractive results. This is due, at least in part, to the fact that MultiReduce preserves symmetry to some extent. Figure 4 shows an example from an article on program visualization [5]:

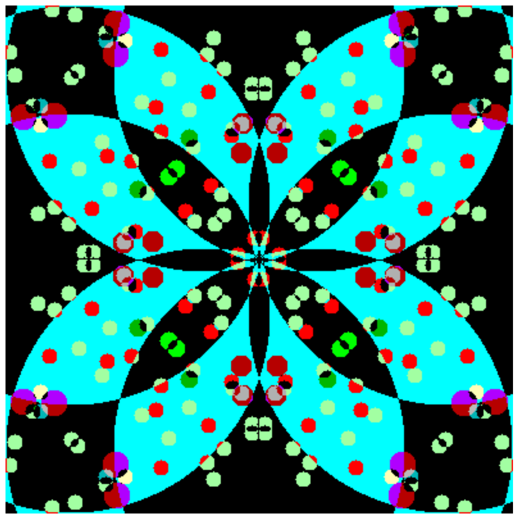


Figure 4. A Symmetric Image

Almost all the 156 MultiReduce permutations of this image are attractive—for example, they would make interesting decorative tiles.

Large images tend to produce more interesting results under MultiReduce than small images. This is due in large part to the fact that large images disintegrate into pixel dust less quickly than small images.

By the nature of MultiReduce, its application to images consisting of vertical or horizontal stripes produces vertical or horizontal stripes, respectively, usually with many interesting variations on the original images. (Permuting the columns of horizontal stripes has no effect, nor does permuting the rows of vertical stripes.) Checks, plaids, and so forth also are good candidates.

The Significance of Image Size

The size of an image strongly affects the period — the number of successive applications of a permutation necessary to produce the original image. If the period is very large, it may be impractical to get all the images, and sometimes the most interesting images occur near the end of the period. Figure 5 shows an image from an earlier Analyst **Graphics Corner** [6] and the results of the penultimate application of MultiReduce.

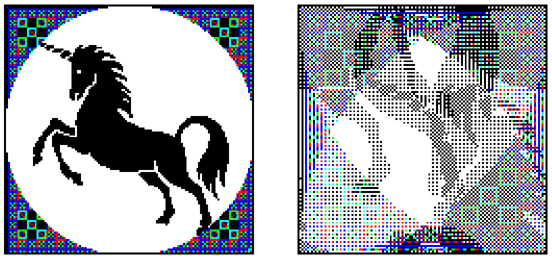


Figure 5. Unicorn Fantasies

If the period is small, on the other hand, there may not be enough images to be worthwhile.

For square images, the period for MultiReduce increases, on average, with the size, but by no means in a regular fashion. For example, the period for a 100 × 100 pixel image is 30, but for a 128 × 128 pixel image, it is 7. Figure 6 shows a histogram of MultiReduce periods for square images of sizes 1 to 500:

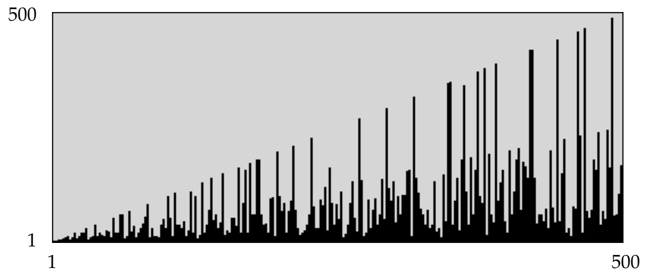


Figure 6. MultiReduce Periods

For images that are not square, the period, which is the least common multiple of the vertical and horizontal periods, can be very large. For example, the period for an 80 × 84 image is 3,198.

Variations on MultiReduce

The basic MultiReduce permutation is 2 × 2; that is, it divides the image in half, horizontally and vertically on each application. Other divisions, such as 1 × 2, 2 × 3, 4 × 4, and so on, produce different images that are no less interesting.

Figures 7 through 10 show the first permutation of the image in Figure 4 for different variations on MultiReduce.

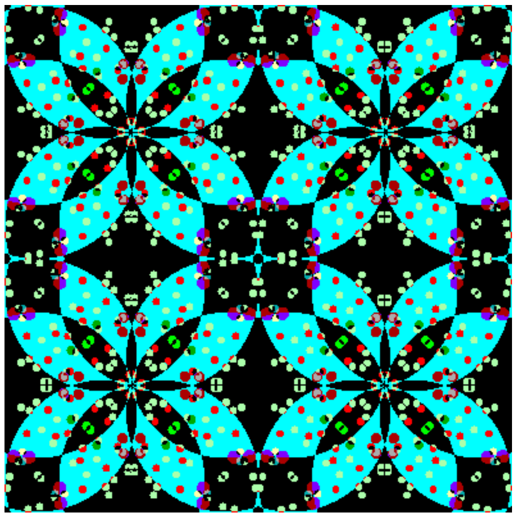


Figure 7. 2 × 2 MultiReduce

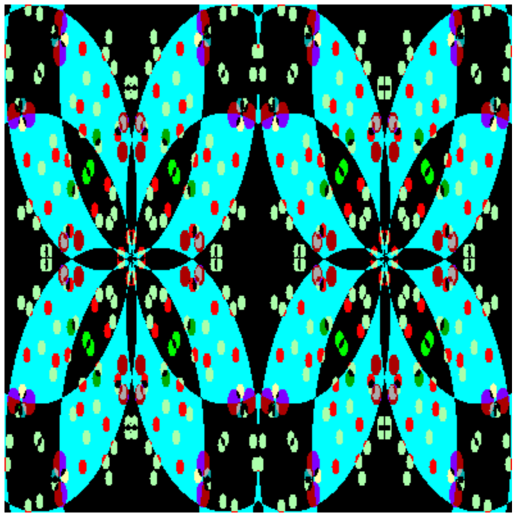


Figure 8. 1 × 2 MultiReduce

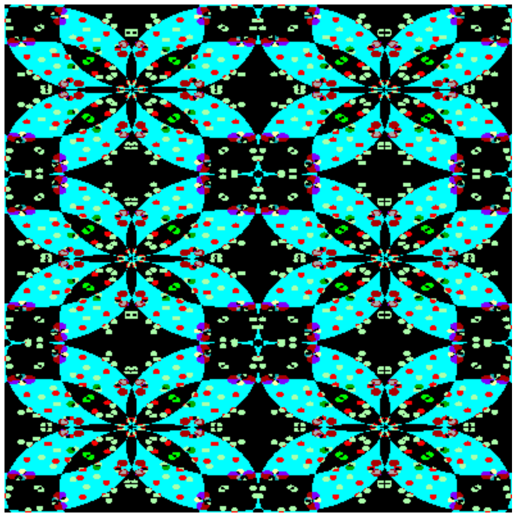


Figure 9. 2 ×3 MultiReduce

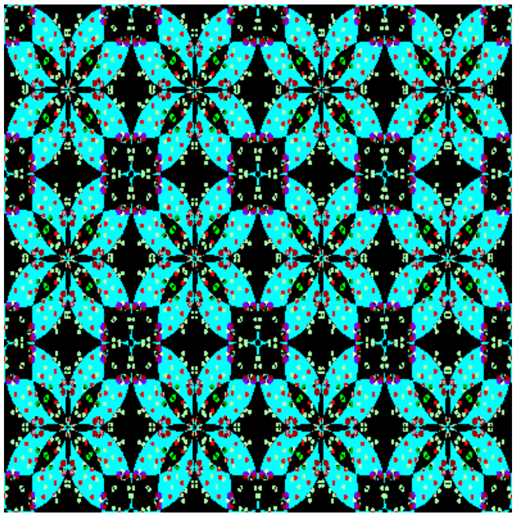


Figure 10. 4 × 3 MultiReduce

MultiReduce and Weavability

Permutations preserve weavability in the sense we described in a recent Analyst article [7]. That is, if an image is weavable, any permutation of it is also. Conversely, if an image is not weavable, then neither is any permutation of it.

A permutation of a weavable image requires the same loom resources — shafts and treadles — as the original image.

Permutations can be viewed as pattern generators and hence as tools for weave design.

Figure 11 shows some examples of MultiReduce applied to a weavable image that is used as a border for Web pages related to weaving [8]. The original image is at the upper left.

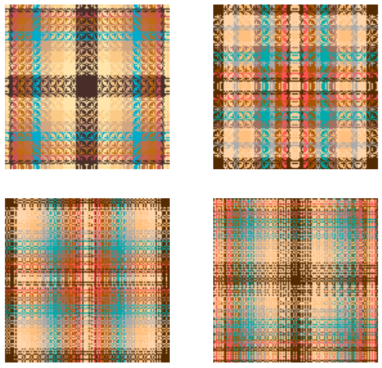


Figure 11. MultiReduce Applied to a Weavable Image

Other Permutations

The number of different permutations of n objects becomes astronomical as n gets large. To make any sense of it, it's necessary to focus on *kinds* of permutations. Some, like reversal and rotation, may be useful for some purposes but not produce much in the way of variety.

One kind of permutation we've tried permutes blocks of pixels while leaving the pixels within a block unchanged. For example, a 32-column image might be divided into 4 blocks of 8 columns each, labeled A, B, C, and D:

A	B	C	D
1 2 … 7 8	9 10 … 15 16	17 18 … 23 24	25 26 … 31 32

These blocks then might have the permutation B D C A, putting columns 9 through 16 first, followed by columns 25 through 32, and so on.

The blocks, of course, need not be of the same length and the number of blocks may vary.

Such block permutations, involving only a few "objects", have short periods. The best results for this kind of permutation occur for images that are themselves "blocky" or at least rectilinear.

Figure 12 shows some block permutations applied to the Web page image shown in the last section:

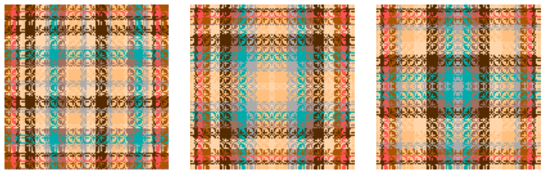


Figure 12. Block Permutations Applied to a Weavable Image

An extension of this idea would be to specify (nontrivial) permutations within the blocks.

Mutations

Permutations rearrange objects. A more general form of "mutation" allows deletion and/or duplication of objects as well. The permutation notation extends naturally to mutations. For example, suppose there are 9 columns in an image. Then the mutation

9 8 7 6 6 5 4 3 2

reverses the order of the columns, while duplicating 6 and discarding 1. The mutation

1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9

magnifies the image by 2, while

1 3 5 7 9

deletes the even-numbered columns.

Aside from using mutations to describe such image manipulations, we can't think of a use for them. Can you?

Animated Permutations on the Web

We've shown a few permutations here, but most permutations have periods that are too long to allow all the images to be shown on a printed page.

Another approach is to bundle the images for a permutation as an animation. We've done this for a few images and put the results on a Web page for this issue of the Analyst.

More Ideas

Here are some ideas for further investigations:

- Investigate the inverses of MultiReduce permutations. The first result of an inverse applied to an image would be the penultimate result of the MultiReduce permutation — which provides an easy way to get to the images near the end of the MultiReduce period.
- Apply permutations directly to threading and treadling sequences. This doesn't require as much mechanism as image manipulation. We've looked over weaving programs that we know about and have not found any such facility.
- Add an element of randomness into the creation of permutations.
- Apply different permutations in succession rather than using the same one repeatedly.
- Investigate applications of the kinds of permutations used in change ringing [9-10].
- Design a language for describing/implementing the design of permutations. Now *there's* a challenge.

Acknowledgment

We are indebted to Bill Jones for the ideas that led to these explorations and for e-mail exchanges with him about the subject.

References

1. "Iteration with Permutations: A 'Magic' Filter", William J. Jones, *Mathematics and Textiles Study Group Newsletter 3*, October 2000.

2. "Transposition Ciphers", Icon Analyst 63, pp. 17-20.

3. "Understanding T-Sequences", Icon Analyst 63, pp. 10-17.

4. "Packet Sequences", Icon Analyst 63, pp. 7-9.

5. "Kaleidoscopic Visualization", Icon Analyst 43, pp. 1-9.

6. "Graphics Corner—Transparency", Icon Analyst 52, pp. 7-10.

7. "Weavable Color Patterns", Icon Analyst 58, pp. 7-10.

8. http://www.cs.arizona.edu/patterns/weaving/

9. *The Fascination of Groups*, F. J. Budden, Cambridge University Press, 1972, pp. 451-479.

10. "Oranges & Lemons Say the Bells of St. Clement's", Pauline Drake, *Handwoven*, September/October 2000, pp. 32-34.

Supplementary Material

Supplementary material for this issue of the Analyst, including images and program material, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia64/

Imagine Our Surprise

In previous articles, we have discussed functions and procedures as values [1,2]; their uses and hazards.

Most Icon programmers have encountered bugs resulting from using a variable that is the name of a function for other purposes.

We frequently use main(args) to get the command-line arguments when a program is executed. But args() is a function and using args as a parameter to main overloads its initial function value with a list of command-line arguments, making the function args() unavailable. This usually goes unnoticed, since args() is rarely used.

The typical reason for such conflicts is that in the design of Icon, function names were chosen for their mnemonic values and hence also likely to be used for other purposes by a programmer.

Most Icon programmers know the names of commonly used functions and hence avoid using these names for other purposes. It's the infrequently used functions that cause the most trouble.

There is a subtler aspect of this problem that we'll illustrate by an example.

In the kaleidoscope program [3,4], there is a slider that controls the speed of the display. The slider position is used to set the delay that is inserted between successive drawings, using WDelay().

Here's a sketch of the code

```
global delay
…
# initialization
…
  delay := 0            # start with fastest drawing
…
# drawing loop
…
  WDelay(delay)
…
# speed callback
procedure speed_cb(vidget, value)
  delay := sqrt(value)
  return
end
```

This code looks perfectly reasonable on the face of it. It does not work, however. Changing the value of delay has no effect on the speed of the display. In fact, the display runs at the maximum speed regardless of the value of delay. Mysterious.

Poring over the code reveals no problem. In fact, the code itself is perfectly correct. What is going on?

WDelay() is an Icon procedure, not a built-in function. It flushes pending output and then uses the built-in function delay() to effect the actual delay.

Eureka! The use of the variable delay in the program overloads its initial function value with a numerical value.

If this caused the program to crash, the source of the problem would be easier to locate. But a numerical value applied to an argument list selects (or attempts to select) one of the arguments by position.

Suppose the value of delay is set to 50. The call delay(i) in WDelay() is equivalent to 50(i), which fails. The failure goes unnoticed and there is no delay regardless of the value set.

Simply changing the name from delay to, say, delayval, as we did in the article on the kaleidoscope application, fixes the problem.

We've seen several instances of this specific problem. Novice Icon programmers who are unable to find the source of their program malfunction typically assume it's a bug in Icon itself.

The reason why this problem is serious is that a good portion of Icon's graphic facilities are written as Icon procedures. (If WDelay() were a built-in function, the problem described above would not occur.)

The severity of the problem can be reduced by using code such as this at the beginning of the procedure WDelay():

```
local delay
initial delay := proc("delay", 0)
```

This fixes the problem described above, but what

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

about other function names in other procedures?

It's not practical or even desirable to "protect" the names of all built-in functions that are used in all procedures.

We settled for an uncomfortable compromise based on problems that have actually occurred. At present, delay, image, and type are "protected" in graphics procedures that are part of the main repertoire and in the vidgets.

The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–analyst@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA®
TUCSON ARIZONA

and

Bright Forest Publishers
Tucson Arizona

There are lessons here, of course: about language design, about the choice of function names, about writing procedures, and about the impact of obscure problems.

The one things we wish is that the Icon translator would provide warning messages when the names of built-in functions are declared as local variables

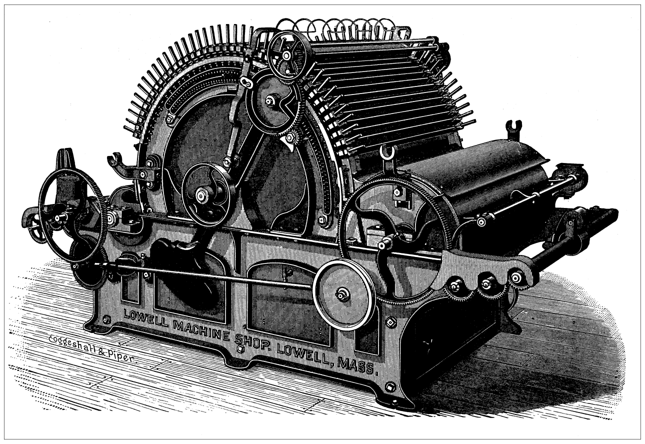
References

1. "Programming Tips", Icon Analyst 5, pp. 11-12.

2. "Procedure and Operator Values", Icon Analyst 29, pp. 1-3.

3. "The Kaleidoscope", Icon Analyst 38, pp. 8-13.

4. "The Kaleidoscope", Icon Analyst 39, pp. 5-10.



What's Coming Up

In programming, as in everything else, to be in error is to be reborn.

— Alan Perlis

With only two issues of the Analyst left, we are running out of options.

We expect to have one more article on solutions of square-root palindromes.We also have an article in the works on their terms and the distribution of terms and another on their lengths.

In addition, we have two more articles on T-sequences, one on generalizing from integer arguments to sequence arguments and another on the analysis of T-sequences.

Beyond that, it's whatever fits or tickles our fancy.