The Icon Analyst

In-Depth Coverage of the Icon Programming Language and Applications

December 2000
Number 63

In this issue

Constant Square-Root Palindromes

In the last issue of the Analyst, we described an application, which we have named qirplore, for exploring the space of square-root palindromes — the palindromic parts of the repeats in continued-fraction sequences for square roots [1].

In this article we'll use qirplore to gather information about constant square-root palindromes — palindromes in which all terms are the same — and then deduce some general results.

Formulas for *m*

In the previous article, we wrote formulas for m in terms of n, as in

palindrome	n	m
1	i	$2n$
1,1,1	$3i-1$	$(4n+1)/3$
1,1,1,1	$5i-2$	$2(3n+1)/5$

It also is possible to write formulas for m in terms of i by substituting the formula for n into the corresponding formula for m (or by computing them that way in the first place). For example, the examples above can be written as

1	i	$2i$
1,1,1	$3i-1$	$4i-1$
1,1,1,1	$5i-2$	$6i-2$

Note that i does not range independently in the formulas for n and m.

Writing formulas for m in terms of i does not reduce the number of distinct parameters — if m is expressed in terms of n, the quotient in the formula for m is the same as the multiplier for i in the formula for n — but formulas for m written in terms of i are easier to handle.

We revised qirplore to make i an independent variable rather than implicit in n. Figure 1 shows the new specification dialog and Figure 2 shows the new result dialog.

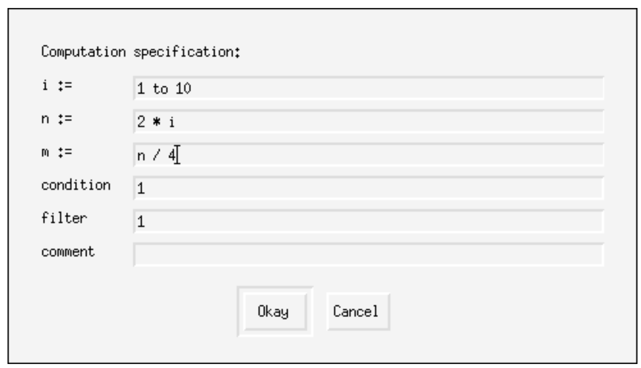


Figure 1. Specification Dialog

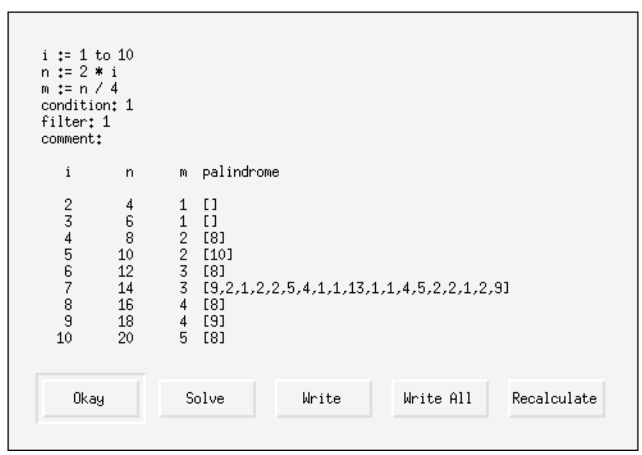


Figure 2. Result Dialog

Previous Results

The previous article had several results for constant palindromes. Here they are with m expressed in terms of i and using the notation $\overline{j}^{\,k}$ to indicate a sequence of k values equal to j.

palindrome	*n*	*m*
$\overline{1}^{1}$	i	$2i$
$\overline{1}^{3}$	$3i–1$	$4i–1$
$\overline{1}^{4}$	$5i–2$	$6i–2$
$\overline{1}^{6}$	$13i–6$	$16i–7$
$\overline{1}^{7}$	$21i–10$	$26i–12$
$\overline{1}^{9}$	$55i–27$	$68i–33$
$\overline{2}^{1}$	$i+1$	$i+1$
$\overline{2}^{2}$	$5i+1$	$4i+1$
$\overline{3}^{3}$	$3(11i–5)$	$20i–9$
$\overline{4}^{1}$	$2i$	i
$\overline{4}^{2}$	$17i+2$	$8i+1$
$\overline{4}^{3}$	$2(18i+1)$	$17i+1$
$\overline{6}^{1}$	$3(i+1)$	$i+1$
$\overline{6}^{2}$	$37i+3$	$12i+1$
$\overline{8}^{1}$	$4(i+1)$	$i+1$
$\overline{8}^{2}$	$65i+4$	$16i+1$

Two possible approaches are to investigate $\overline{j}^{k}$ with k constant and j varying or with j constant and k varying. We'll start with k constant and j varying.

Constant *k* and Varying *j*

k = 1

In the last article, we explored unit palindromes, that is $k=1$. The results depend on whether j is odd or even. For j odd, they are:

$$n = j\,i \qquad\qquad k = 1,\ j \text{ odd}$$
$$m = 2i$$

and for j even, they are:

$$n = (j/2)i + j/2 \qquad\qquad k = 1,\ j \text{ even}$$
$$m = i + 1$$

k = 2

Going on to $k = 2$, we have some values from the last article:

palindrome	*n*	*m*
$\overline{2}^{2}$	$5i+1$	$4i+1$
$\overline{4}^{2}$	$17i+2$	$8i+1$
$\overline{6}^{2}$	$37i+3$	$12i+1$
$\overline{8}^{2}$	$65i+4$	$16i+1$

We showed in the last article that $\overline{1}^{2}$ does not exist. The results above suggest that $\overline{j}^{2}$ for j odd do not exist, but that could just be because we were not looking for such palindromes.

In fact, we can show that $\overline{j}^{2}$ for j odd do not exist by solving the continued fraction for

$$x = n + \cfrac{1}{j + \cfrac{1}{j + \cfrac{1}{2n + \ldots}}}$$

We'll say more on this in a subsequent article.

We can get a general result from the formulas above by inspection. For j even,

$$n = (j^2+1)i + j/2 \qquad\qquad k = 2,\ j \text{ even}$$
$$m = 2ji + 1$$

These formula are, of course, just conjectures. However, they hold up for very large j, so we have considerable confidence in them. (We can prove these formulas by solving the continued fraction above; we'll get to this later.)

k = 3

So far, so good. What about $k = 3$? What we have from the last article, with factors multiplied out is:

palindrome	*n*	*m*
$\overline{1}^{3}$	$3i–1$	$4i–1$
$\overline{3}^{3}$	$33i–15$	$20i–9$
$\overline{4}^{3}$	$36i+2$	$17i+1$

In this case, the value for $j = 2$ is missing simply because we didn't happen to look for it last time. Certainly we don't have enough data points to find formulas, so getting results for other values of j is the next order of business. Before going on, however, note that the constant terms in the formulas have different signs for j odd and j even. We may therefore expect different formulas for the two cases.

The filter field of qirplore can be used to limit results in various ways. For example, to get all constant palindromes of length 3, we can use the search dialog shown in Figure 3, which gives the results shown in Figure 4.

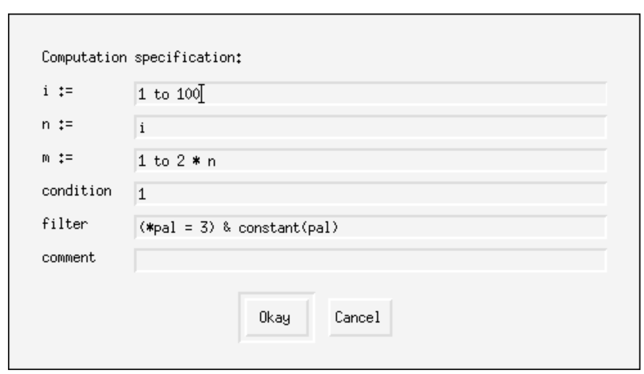


Figure 3. Search Dialog for $\overline{j}^{3}$

```
i := 1 to 100
n := i
m := 1 to 2 * n
condition: 1
filter: (*pal = 3) & constant(pal)
comment:

   i       n      m  palindrome

   2       2      3  [1,1,1]
   5       5      7  [1,1,1]
   7       7      6  [2,2,2]
   8       8     11  [1,1,1]
  11      11     15  [1,1,1]
  13      13     11  [2,2,2]
  14      14     19  [1,1,1]
  17      17     23  [1,1,1]
  18      18     11  [3,3,3]
  19      19     16  [2,2,2]
  20      20     27  [1,1,1]
  23      23     31  [1,1,1]
  25      25     21  [2,2,2]
  26      26     35  [1,1,1]
  29      29     39  [1,1,1]
  31      31     26  [2,2,2]
  32      32     43  [1,1,1]
  35      35     47  [1,1,1]
  37      37     31  [2,2,2]
  38      38     18  [4,4,4]
  38      38     51  [1,1,1]
          ...

 Okay    Solve    Write    Write All    Recalculate
```

Figure 4. Result Dialog for $\overline{j}^{\,3}$

These results show, among other things, the expected palindromes for $k = 3$. But the results for different values of k are mixed up. By using the fact that if constant() succeeds, it returns the constant value, we can confine the search to $k = 3$. See Figures 5, 6, and 7.

```
Computation specification:

i :=        1 to 100
n :=        i
m :=        1 to 2 * n
condition   1
filter      (*pal = 3) & (constant(pal) = 2)
comment

            Okay   Cancel
```

Figure 5. Specification Dialog for $\overline{2}^{\,3}$

Back Issues

Back issues of The Icon Analyst are available for \$5 each. This price includes shipping in the United States, Canada, and Mexico. Add \$2 per order for airmail postage to other countries.

```
i := 1 to 100
n := i
m := 1 to 2 * n
condition: 1
filter: (*pal = 3) & (constant(pal) = 2)
comment:

   i       n      m  palindrome

   7       7      6  [2,2,2]
  13      13     11  [2,2,2]
  19      19     16  [2,2,2]
  25      25     21  [2,2,2]
  31      31     26  [2,2,2]
  37      37     31  [2,2,2]
  43      43     36  [2,2,2]
  49      49     41  [2,2,2]
  55      55     46  [2,2,2]
  61      61     51  [2,2,2]
  67      67     56  [2,2,2]
  73      73     61  [2,2,2]
  79      79     66  [2,2,2]
  85      85     71  [2,2,2]
  91      91     76  [2,2,2]
  97      97     81  [2,2,2]

 Okay    Solve    Write    Write All    Recalculate
```

Figure 6. Results for $\overline{2}^{\,3}$

```
n = 6*i+1
m = (5*n+1)/6
m = (5*i+1)
palindrome: [2,2,2]

 Write   Verify   Okay
```

Figure 7. Solution for $\overline{2}^{\,3}$

Continuing in this manner, we can get additional data points, although it becomes more difficult as k gets larger to find even the first one. Fortunately, if we can find the first two, the difference between them can be used to get more.

Here are the formulas we have, separated into j odd and j even:

	palindrome	*n*	*m*
odd	$\overline{1}^{\,3}$	$3i–1$	$4i–1$
	$\overline{3}^{\,3}$	$33i–15$	$20i–9$
	$\overline{5}^{\,3}$	$135i–65$	$52i–25$
	$\overline{7}^{\,3}$	$357i–175$	$100i–49$
even	$\overline{2}^{\,3}$	$6i+1$	$5i+1$
	$\overline{4}^{\,3}$	$36i+2$	$17i+1$
	$\overline{6}^{\,3}$	$114i+3$	$37i+1$
	$\overline{8}^{\,3}$	$264i+4$	$65i+1$

We'll start with j even, since the formulas are simpler. If we express the general formulas as

$$n = ai + b$$
$$m = ci + d$$

it's easy to see that $b = j/2$, $c = j^2 + 1$, and $d = 1$. A formula for a is not obvious. The method of differences does not yield a solution, and the sequence 6, 36, 114, 264 is not in the *On-Line Encyclopedia of*

Integer Sequences (OLEIS) [2].

All the terms in this sequence, however, are divisible by 6, giving 1, 6, 19, 44, which *is* in OLEIS (**A5900**), which tells us these are the first four octahedral numbers: $(2t^3 + t)/3$. Multiplying by 6 and substituting $j/2$ for t gives us $a = j^3/2 + j$.

Consequently, the general formulas for j even are

$$n = (j^3/2 + j)i + j/2 \qquad k = 3, j \text{ even}$$
$$m = (j^2 + 1)i + 1$$

Again, testing these formulas for large values of j gives us confidence of their correctness.

The case for j odd is a bit more difficult, but using a combination of methods, including OLEIS (**A5917**) to find that b is obtained from the rhombic dodecahedral numbers

$$t^4 - (t-1)^4$$

we eventually derived these formulas for j odd:

$$n = (j^3 + 2j)i - j(j^2 + 1) \qquad k = 3, j \text{ odd}$$
$$m = (2j^2 + 2)i - j^2$$

k = 4

The going gets progressively tougher. For $k = 4$, we had to resort to OLEIS's Superseeker, which tries all kinds of methods to attempt to find formulas for sequences that are not in OLEIS. We also used *Mathematica*, polyseq() and llrcseq() from the Icon program library module genrfncs, and some unorthodox techniques.

We'll spare you the details and just give the results. Given the general forms

$$n = ai + b$$
$$m = ci + d$$

then

$$n = (j^4 + 3j^2 + 1)i - (16j^3 - 83j^2 + 175j - 104) \qquad k = 4, j \text{ odd}$$
$$m = 2(j^3 + 2j)i - (j^3 + 2j - 1)$$

$$n = (j^4 + 3j^2 + 1)i + j/2 \qquad k = 4, j \text{ even}$$
$$m = 2(j^3+2j)i + 1$$

Note that the formulas for a and c are the same for j odd and j even.

Going On

As gratifying as it was to puzzle out these formulas and then verify them for large values of j, their form does not bode well for larger value of k. (While the formulas can be written in other ways and somewhat simplified, the powers of j remain.) In fact, we stalled at $k = 5$.

The problem is in getting enough initial terms to derive a formula. Since the powers of j in the formulas increase as k does (and we expect that to continue), it requires more initial terms to derive formulas. As we mentioned in an earlier article [3], all polynomials of degree t can be represented by a single recurrence of order $t+1$, with the initial terms for the recurrence depending on the coefficients in the polynomial. For example, to specify a polynomial of degree 3, it takes 4 initial terms.

As k increases, it becomes very difficult to get even one term for $j > 2$. For example, the first term for $k = 5$ and $j = 4$ is for $n = 648$. If you don't have a basis for making a good guess, it becomes hopeless to find such terms.

Patterns

Finding general formulas that work for all k seems unlikely, at least using the empirical approach we've used here, especially for j odd.

Based on what we have found, we'd conjecture that all the formulas can be represented by polynomials in j (as, say, opposed to recurrence relations that do not have polynomial solutions). It also seems that the highest power of j in n is k and is $k–1$ in m.

For j even, the general situation is more promising. Given the forms we've used:

$$n = ai + b$$
$$m = ci + d$$

we can write a table of coefficients:

j even

k	a	b	c	d
1	$j/2$	$j/2$	1	1
2	$j^2 + 1$	$j/2$	$2j$	1
3	$j^3/2 + j$	$j/2$	$j^2 + 1$	1
4	$j^4 + 3j^2 + 1$	$j/2$	$2j^3 + 4j$	1

Just in terms of pattern-matching, it seems very likely that

$$b = j/2$$
$$d = 1$$

for all even j. In fact, this is the case for a large number of individual results we've found.

Knowing two of the coefficients *a priori* considerably simplifies finding results for larger k. There also are patterns in the coefficients of j in a and c. But the results in the next section make this unimportant.

Constant *j* and Varying *k*

j = 1

It's relatively easy to get formulas for $\overline{1}^k$ using qirplore. Here are the first few:

palindrome	*n*	*m*
$\overline{1}^1$	$1i–0$	$2i–0$
$\overline{1}^3$	$3i–1$	$4i–1$
$\overline{1}^4$	$5i–2$	$6i–2$
$\overline{1}^6$	$13i–6$	$16i–7$
$\overline{1}^7$	$21i–10$	$26i–12$
$\overline{1}^9$	$55i–27$	$68i–33$
$\overline{1}^{10}$	$89i–44$	$110i–54$
$\overline{1}^{12}$	$233i–116$	$288i–143$
$\overline{1}^{13}$	$377i–188$	$466i–232$

Note that there are no palindromes for $k = 2 \bmod 3$. We showed earlier using an algebraic solution that there was no palindrome for $k = 2$. For larger values of k, it is a conjecture. It simplifies the problem further to separate the cases $k = 0 \bmod 3$ and $k = 1 \bmod 3$.

Using

$$n = ai - b$$

$$m = ci - d$$

the sequences for a, b, c, and d are:

$a = 3, 13, 55, 233, \ldots$ $\qquad k = 0 \bmod 3$

$b = 1, 6, 27, 116, \ldots$

$c = 4, 16, 68, 288, \ldots$

$d = 1, 7, 33, 143, \ldots$

$a = 1, 5, 21, 89, 377, \ldots$ $\qquad k = 1 \bmod 3$

$b = 0, 2, 10, 44, 118, \ldots$

$c = 2, 6, 26, 110, 466, \ldots$

$d = 0, 2, 12, 54, 232, \ldots$

For both cases, the coefficients a, b, and c are given by simple recurrences (which do not have polynomial solutions). For a and c, the recurrence is

$$t_n = 4t_{n-1} + t_{n-2}$$

while for b, it is

$$t_n = 5t_{n-1} - 3t_{n-2} - t_{n-3}$$

The coefficient d is not given by a simple recurrence, but instead $d_m = b_m + b_{m-1}$, where $m > 1$ is the position in the two sets of sequences, and b_1 is 1, and 0, respectively. (Finding this relation is a matter of pattern matching combined with elementary arithmetic; it is crucial in the solution.)

A procedure that writes out the formulas for a given k is:

```
link genrfncs

procedure p1k(k)
   local i, a, b, c, x

   case k % 3 of {
      0  :  {
         i := k / 3                        # limit
         every a := lrrcseq([3, 13], [4, 1]) \ i
         every b := lrrcseq([1, 6, 27], [5, –3, –1]) \ i
         every x := lrrcseq([1, 6, 27], [5, –3, –1]) \ (i – 1)
         /x := 0                           # first term
         every c := lrrcseq([4,16], [4, 1]) \ i
         }
      1  :  {
         i := k / 3 + 1                    # limit
         every a := lrrcseq([1, 5], [4, 1]) \ i
         every b := lrrcseq([0, 2, 10], [5, –3, –1]) \ i
         every x := lrrcseq([0, 2, 10], [5, –3, –1]) \ (i – 1)
         /x := 0                           # first term
         every c := lrrcseq([2, 6], [4, 1]) \ i
         }
      2  :  fail
      }

   write("n=", a, "*i–", b, "\tm=", c, "*i–", b + x)

   return

end
```

j = 2

As usual, the situation for j even is simpler than for j odd Here's the initial data:

palindrome	*n*	*m*
$\overline{2}^1$	$i+1$	$1i+1$
$\overline{2}^2$	$5i+1$	$4i+1$
$\overline{2}^3$	$6i+1$	$5i+1$
$\overline{2}^4$	$29i+1$	$24i+1$
$\overline{2}^5$	$35i+1$	$29i+1$
$\overline{2}^6$	$169i+1$	$140i+1$

As noted earlier, $c = 1$ and $d = j/2 = 1$, so there are only two sequences to solve. These both are given by the same recurrence:

$t_n = 6t_{n-2} - t_{n-4}$

A procedure that writes out the formulas for a given k is:

```
link genrfncs

procedure p2k(k)
   local a, c

   every a := lrrcseq([1, 5, 6, 29], [0, 6, 0, –1]) \ k

   every c := lrrcseq([1, 4, 5, 24], [0, 6, 0, –1]) \ k

   write("n=", a, "∗i+1\tm=", c, "∗i+1")

   return

end
```

$k > 1$, j Odd

We came up absolutely empty for $j > 1$, j odd. We have been unable to find any palindromes for $k > 4$ (palindromes for $k \leq 4$ were given previously). We even wonder if such palindromes exist. It seems they should exist but probably only for $k = 0$ mod 3 and $k = 1$ mod 3.

$k > 2$, j Even

We expected results for $j > 2$, j even. We did indeed find them — in fact, more than we expected — but also discovered a surprising relationship.

First we start with a fact we turned up using OLEIS for $j = 2$. The sequence for a,

$a = 1, 5, 6, 29, 35, 169, \ldots$

gives the convergents to the continued fraction for $\sqrt{8}$ (**A41011**). At the time we just thought this was a curious coincidence.

Going to $j = 4$, our initial values were

$a = 2, 17, 36, 305, 646, \ldots$
$c = 1, 8, 17, 144, 305, \ldots$

with $b = j/2 = 2$ and $d = 1$ as expected.

Following our usual plan, we tried first to find a recurrence for a — and failed. Resorting again to OLEIS, we found this sequence gives the convergents for the continued fraction for $\sqrt{20}$ (**A41031**).

At this point we observed that $c_m = 4a_{m-1}$ for m even and $c_m = a_{m-1}$ for m even. See Figure 8.

$$\begin{tikzcd}[column sep=small] & 1 & 2 & 3 & 4 & 5 & \\ a = & 2, \arrow[dr, "\times 4"] & 17, \arrow[dr] & 36, \arrow[dr, "\times 4"] & 305, \arrow[dr] & 646, \arrow[dr, "\times 4"] & \ldots, \\ c = & 1, & 8, & 17, & 144, & 305, & \ldots, \end{tikzcd}$$

Figure 8. Relationship between *a* and *c*

Consequently, a completely characterizes the palindromes for $j = 4$. Or, if you prefer, $\sqrt{20}$ is a complete characterization of the palindromes $\overline{4}^{k}$.

The relationship between c and a also holds for $j = 2$, although we did not formulate the solution of $j = 2$ in that way. Alternatively, $\sqrt{8}$ is a complete characterization of the palindromes $\overline{2}^{k}$.

Could this be a coincidence? If not, what is the relationship between the values of j and the square roots?

Going on to $j = 6$, 8, and 10, we find the continued-fraction convergents again, this time for $\sqrt{40}$, $\sqrt{68}$, and $\sqrt{104}$, respectively.

And what is the relationship between 8, 20, 40, 68, and 104? These numbers all are divisible by 4, so the underlying sequence of interest is 2, 5, 10, 17, 26. That's easy: $i^2 + 1$, i = 1, 2, 3, 4, 5. In terms of j, the sequence is $4(j/2)^2 + 1$.

So our final result (conjecture) is that the palindromes $\overline{j}^{k}$, j even, can be computed from the denominators of the convergents to the continued fractions for

$$\sqrt{4(j/2)^2 + 1}$$

Granted, the convergents have to be computed, but we've shown how to do that [4].

Summary

This article illustrates how a variety of tools — qirplore, various procedures in the Icon program library, OLEIS, and Superseeker — can be used in combination with deduction, intelligent guessing, and a lot of work to produce solutions to a problem.

Sure, the problem is recreational and highly specialized. But it's interesting to work on. It reminds us of the remark by a professor: "Once a doctoral student selects a topic and starts to study it intensively, that student soon knows more about the topic than anyone else." This, of course, also is a telling comment about specialization.

Next Time

We've about reached the limit of what we can do with the empirical approach used in this article.

In the next article on square-root palindromes, we'll look at what can be learned from algebraic solutions of square-root continued fractions. This will lead to Diophantine equations — equations for which the solutions must be in the integers.

References

1. "Square Root Palindromes", Icon Analyst 62, pp. 1-5.

2. http://www.research.att.com/~njas/sequences/

3. "Recurrence Relations", Icon Analyst 59, pp. 18-20.

4. "Continued Fractions, Icon Analyst 60, pp. 1-5.

Packet Sequences

A packet sequence is a sequence in which terms may be sequences.

Packet sequences can be used as a notational device for grouping terms in a sequence into sub-sequences to help reveal structures [1].

Consider, for example,

1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6 ...

Arranging terms in groups of three, we have

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

which makes the pattern easier to see.

Here is another example:

1, 2, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, ...

Clearly there are groups of successive integers, which we can group as

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

Now look at the lengths of the packets:

1, 1, 2, 3, 5, 8, ...

Ah, the ever-present Fibonacci sequence.

Packets can contain packets, as in

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

Data Representation

If lists are used to represent sequences, then packets are lists within lists. The first example above then would be

```
seq1a := [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6 ... ]
```

and after arranging by packets

```
seq1b := [[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6], ... ]
```

where the ellipses indicate terms that are not shown but are limited in number. Then

```
(!seq1b)[1]
```

produces 1, 2, 3, 4, ...

The second example would be

```
seq2a := [1, 2, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6,
  6, 6, 6, 6, ...]

seq2b := [[1], [2], [3, 3], [4, 4, 4], [5, 5, 5, 5, 5],
  [6, 6, 6, 6, 6, 6, 6, 6], ...]
```

and

```
*!seq2b
```

produces 1, 1, 2, 3, 5, 8, ...

Packet Sequence Procedures

The following procedures illustrate methods of creating and manipulating packet sequences.

Here's a procedure that puts successive duplicate values into packets.

```
procedure packdup(s)
  local result, packet, i, j

  s := copy(s)

  result := []

  j := get(s) | return result
  packet := [j]

  while i := get(s) do {
    if i = j then
      put(packet, i)
    else {
      put(result, packet)
      packet := [i]
      j := i
      }
```

```
    }
  put(result, packet)

  return result

end
```

For example, packdup(seq2a) produces seq2b.

This programmer-defined control operation does the same thing, but for generators:

```
procedure packdupPDCO(L)
  local packet, i, j

  j := @L[1] | fail
  packet := [j]

  while i := @L[1] do {
    if i = j then
      put(packet, i)
    else {
      suspend packet
      packet := [i]
      j := i
      }
    }

  return packet

end
```

Here's a procedure that produces packets of a specified length:

```
procedure packlen(s, i)
  local result, packet

  s := copy(s)

  result := [ ]

  while *s > 0 do {
   packet := [ ]
    every 1 to i do
      put(packet, get(s)) | break
    put(result, packet)
    }

  return result

end
```

For example, packlen(seq1a, 3) produces seq1b.

This procedure puts sequences of increasing values into separate packets:

```
procedure packup(s)
  local result, packet, i, j

  s := copy(s)

  result := []

  j := get(s) | return result
  packet := [j]

  while i := get(s) do {
    if i > j then {
      put(packet, i)
      }
    else {
      put(result, packet)
      packet := [i]
      }
    j := i
    }

  put(result, packet)

  return result

end
```

For example, packup(seq1a) produces seq1b, but for entirely different reasons than packlen().

The following procedure is a generalization of packlen() in which a second sequence determines the lengths of successive packets:

```
procedure packlenv(s1, s2)
  local result, packet, i

  result := []

  s1 := copy(s1)
  s2 := copy(s2)

  while i := get(s2) do {
    put(s2, i)                  # cyclic shift
    packet := []
    every 1 to i do
      put(packet, get(s1)) | {
        put(result, packet)     # short packet
        break break
        }
      put(result, packet)
    }

  return result

end
```

Note that s2 is cyclically rotated, so the process continues until s1 is exhausted. There may be a short packet when s2 runs out.

The following procedure produces a "flat" sequence with no packets from one that may have packets:

```
procedure flatpack(s)
   local result, x

   result := []

   every x := !s do
      if type(x) == "list" then
         result |||:= flatpack(x)
      else put(result, x)

   return result

end
```

For example, flatpack(seq1b) produces seq1a.

Here's a programmer-defined control operation for flattening generators:

```
procedure flatpackPDCO(L)
   local x

   while x := @L[1] do
      if type(x) == "list" then
         suspend !flatpack(x)
      else suspend x

end
```

Finally, this procedure produces string images of packet sequences:

```
procedure imagepack(s)
   local result, x

   result := ""

   every x := !s do {
      if integer(x) then result ||:= x else
         result ||:= pimage(x)
      result ||:= ","
      }

   return "[" || result[1:–1] || "]"

end
```

The result is a string in the fashion of limage() in the Icon program library module lists, but using recursion to show packets, packets within packets, and so on.

Loops in List Structures

Because of Icon's pointer semantics [2], it is possible to construct loops in list structures. For example,

```
one_seventh := [1, 4, 2, 8, 5, 7]
put(one_seventh, one_seventh)
```

provides a finite representation of a purely periodic sequence.

The result can be visualized as shown in Figure 1.

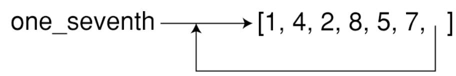


Figure 1. Packet Sequence Loop

Similarly,

```
one_third := [3]
put(one_third, one_third)
five_sixths := [8, one_third]
```

can be used to represent a periodic sequence with a preperiodic part.

It also is possible to construct sequences that have unreachable portions if loops are traversed from left to right. An example is

```
rep := [1, 2, 3]
put(rep, rep)
struct := [3, 4, rep, 5]
```

The loop in rep prevents the value 5 in struct from being reached if the pointer to rep is followed.

Lists with loops cause problems when used with procedures like imagepack() and flatpack() that expect a finite number of terms. In both cases, a sequence with (reachable) loops results in stack overflow because of uncontrolled recursion.

We simply will exclude list structures that have loops from the packet sequence mechanism.

More to Come

We have one more article on ways of representing data in sequences — template sequences.

Template sequences consist of expressions that produce sequences. As you'd expect, a good way to do this in Icon is to use co-expressions.

Incidentally, the ideas for packet sequences and template sequences both come from Hofstader [1].

References

1. "To Seek Whence Cometh a Sequence" in *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*, Douglas Hofstadter, Basic Books, 1995, pp. 13-86.

2. "Pointer Semantics", The Icon Analyst 6, pp. 2-8.

Understanding T-Sequences

In the *Analyst* article on tie-ups and T-sequences [1], we showed some simple and frequently used T-sequences.

As shown in that article, tie-ups play a major role in the drawdown patterns that result for particular threading and treadling sequences.

On the other hand, some kinds of T-sequences work best with specific kinds of tie-ups and many T-sequences are designed first and appropriate tie-ups selected later.

In this and subsequent articles, we'll explore various aspects of T-sequences, analyzing ones used in actual weaves in an attempt to develop a set of operations that can be used both for describing T-sequences and for constructing them.

It's worth noting that while many T-sequences used in weaving are constructed using formulas and variations on themes, others are constructed by means whose results cannot be usefully described by patterns. Examples are some forms of name drafting, T-sequences produced by digitizing curves, and T-sequences derived from complex integer sequences [2]. And some T-sequences are created by whim and have no evident pattern.

Examples

Conventions

The plots of T-sequences that are too long to show horizontally are displayed vertically. A vertical plot begins at the upper-left corner. Values increase to the right and downward. Plots show at least one repeat.

Uppercase italic letters are used to designate sequences, as in *S*, *T*, and *U*. Lowercase letters are used to designate integers, as in *i*, *k*, and *j*.

Examples

Figure 1 shows five T-sequences that have patterns commonly found in weaving drafts.

Runs

The T-sequence in Figure 1a consists of runs of consecutive integers, such as 1, 2, 3, 4 and 4, 5, 6, 7, 8.

For runs, also the basis for straight draws and wave draws, we'll use the notation

$i \rightarrow j$ *run*

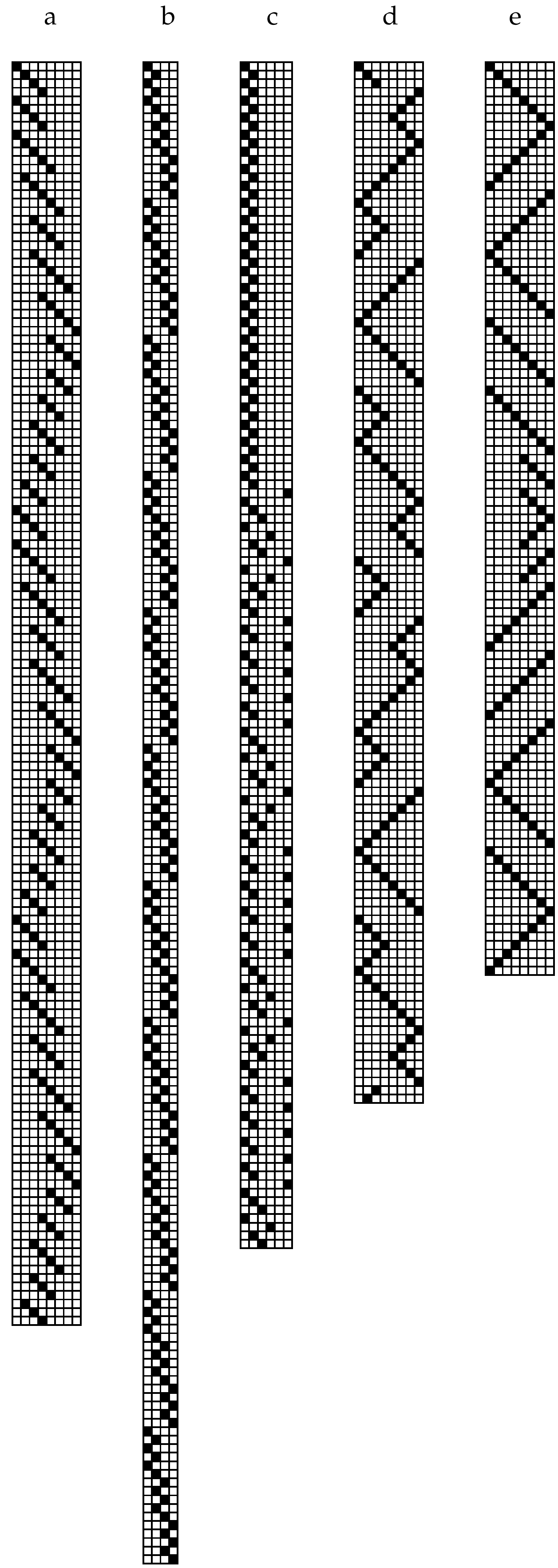


Figure 1. Example T-Sequences

If $j > i$, the run is ascending, while if $j < i$, it is descending. Examples are $1 \rightarrow 8$ and $6 \rightarrow 2$, which produce 1, 2, 3, 4, 5, 6, 7, 8 and 6, 5, 4, 3, 2, respectively. See Figures 2a and 2b.

An obvious generalization is the specification of an increment other than 1, for which we'll use the notation

$$i \xrightarrow{k} j \qquad \textit{run}$$

where k is the increment. For example,

$$1 \xrightarrow{2} 7$$

produces 1, 3, 5, 7 and

$$9 \xrightarrow{3} 3$$

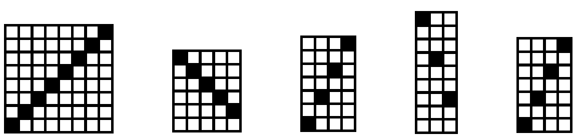
produces 9, 6, 3. See Figures 2c and 2d.

Notice the increment is expressed as a positive integer for both ascending and descending runs.

If the run does not "come out even" at the right bound, it stops at the preceding value. For example,

$$1 \xrightarrow{2} 8$$

produces 1, 3, 5, 7. See Figure 2e.



$1 \rightarrow 8$ $\quad 6 \rightarrow 2$ $\quad 1 \xrightarrow{2} 7$ $\quad 9 \xrightarrow{3} 3$ $\quad 1 \xrightarrow{2} 8$

a b c d e

Figure 2. Runs

Concatenation

Concatenation, appending one sequence to the end of another, is the most basic operation for creating sequences from other sequences. For example, the T-sequence in Figure 1a consists of the concatenation of many runs.

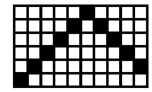
For the concatenation of two sequences, we'll use the notation

$$S\ ,\ T \qquad \textit{concatenation}$$

For example,

$$(1 \rightarrow 6)\ ,\ (5 \rightarrow 2)$$

produces 1, 2, 3, 4, 5, 6, 5, 4, 3, 2. See Figure 3.



$(1 \rightarrow 6)\ ,\ (5 \rightarrow 2)$

Figure 3. Concatenation

You may wonder about the apparent ambiguity of using commas both for the concatenation of sequences and also for separating the terms in sequences. We will address this issue later.

Repeats

Repeats are a part of almost all T-sequences. Figure 1a has repeats and repeats within repeats (can you find them?). Figure 1b consists entirely of a length-16 repeat. Figure 1c consists of the concatenation of two sequences with different repeats.

For repeats, we'll use the standard mathematical notation

$$\overline{S} \qquad \textit{repeat}$$

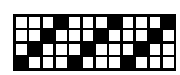
which indicates an indefinite number of repeats of S. Although actual T-sequences are finite, it sometimes is useful to specify an indefinite number of repeats during their construction. For a specific number of repeats, we'll use

$$\overline{S}^{\,i} \qquad \textit{repeat}$$

For example

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

produces 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4. See Figure 4.



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

Figure 4. Repeat

Extension

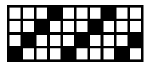
Sometimes a sequence needs to be extended by repetition to a specific length that may not be an even multiple of the length of the repeat. For this we'll use the notation

$$S \Rightarrow i \qquad \textit{extension}$$

to indicate that S is repeated as necessary to produce a sequence of length i. For example,

$$(1 \rightarrow 4) \Rightarrow 10$$

produces 1, 2, 3, 4, 1, 2, 3, 4, 1, 2. See Figure 5.



$(1 \to 4) \Rightarrow 10$

Figure 5. Extension

If the specified length is less than the length of the sequence, the sequence is truncated at the right.

It sometimes is necessary to know the length of a sequence, for which we'll use the notation

$\lambda(S)$ *length*

For example,

$S \Rightarrow \lambda(T)$

extends S to the length of T.

Palindromes

Palindromes play an important role in T-sequences by providing an element of symmetry that is aesthetically pleasing.

There are two forms of palindromes: open and closed. In an open palindrome, the last term of the reversed sequence is omitted. This allows open palindromes to be repeated without introducing duplicate terms at the boundaries of the repeats. In a closed palindrome, the term is not omitted and the result is a true palindrome. (A palindrome also can be open at the beginning instead of the end. We'll handle this another way.)

Figure 1d shows an open palindrome, while Figure 1e shows a closed one.

We'll use the notation

$\ulcorner S$ *open palindrome*

to indicate an open palindrome and the notation

$\ulcorner S \urcorner$ *closed palindrome*

to indicate a closed palindrome. For example,

$\ulcorner (1 \to 4)$

produces 1, 2, 3, 4, 3, 2, while

$\ulcorner (1 \to 4) \urcorner$

produces 1, 2, 3, 4, 3, 2, 1. Either way, the center term (4) is not duplicated. See Figures 6a and 6b.

A sticky point remains. Open palindromes often are used in repeats. In a complete T-sequence, however, the last repeat is closed. In published drafts it's common to see a remark like this: "By the way, be sure to add a thread to the end of the repeated palindrome so the whole comes out symmetrically."

We haven't found a good way to deal with this problem, but since the pattern is so common, we'll use this notation

$\ulcorner S \urcorner^{i}$ *closed palindrome*

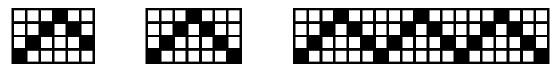
to mean i repeats of

$\ulcorner S$

and then closed. For example,

$\ulcorner (1 \to 4) \urcorner^{3}$

produces 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2,1. See Figure 6c.



$\ulcorner (1 \to 4)$ $\ulcorner (1 \to 4) \urcorner$ $\ulcorner (1 \to 4) \urcorner^{3}$

a b c

Figure 6. Palindromes

Reflections

Horizontal reflection, or reversal, is the basis for the construction of palindromes. It also is used in other ways. We'll use the notation

$\overleftrightarrow{S}$ *horizontal reflection*

For example,

$\overleftrightarrow{1 \to 8}$

produces 8, 7, 6, 5, 4, 3, 2, 1. See Figure 7a.

Horizontal reflection can be used to convert a palindrome open at one end to a palindrome open at the other.

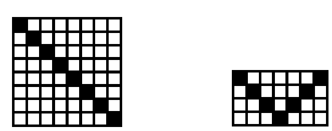
While we're at it, we'll add vertical reflection, using the notation

$\updownarrow S$ *vertical reflection*

For example,

$\updownarrow(\ulcorner (1 \to 4) \urcorner)$

produces 4, 3, 2, 1, 2, 3, 4. See Figure 7b.



$\overleftrightarrow{1 \to 8}$ $\updownarrow(\ulcorner (1 \to 4) \urcorner)$

a b

Figure 7. Reflections

Vertical reflection requires knowing the bound on the sequence, that is, its largest term (since T-sequences are finite, they are bounded). For this, we'll use the notation

$\beta(S)$	*bound*

For example, $\beta(1 \to 4)$ produces 4.

Summary

The relatively simple T-sequences in Figure 1 have led to the following operations on sequences, which also can be viewed as patterns:

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

We also have two scalar operations that produce integers from sequences.

$\lambda(S)$	*length*
$\beta(S)$	*bound*

Using the Operations

We're now in a position to see how these operations can be used to describe the T-Sequences in Figure 1.

Figure 1a

By inspecting Figure 1a, we can determine that it has two structural components, one a short run at the beginning that we'll call *S* and another, more complicated component that we'll call *T* that then is repeated. The two components are shown in Figure 8.

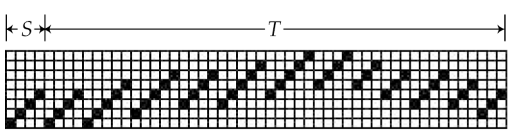


Figure 8. The Basic Components of Figure 1a

S is easy enough: $1 \to 4$. For *T*, it's more complicated but still straightforward:

$$T = 1 \to 4, 1 \to 5, 2 \to 6, 3 \to 6, 3 \to 7, 4 \to 8,$$
$$5 \to 8, 5 \to 7, 4 \to 6, 3 \to 6, 3 \to 5, 2 \to 4$$

where we assume that the concatenation operation has lower precedence than the run operation.

The entire sequence then is

$$S\ ,\ \overline{T}^{\,3}$$

There is actually more structure in *T* than is evident in the formulation above. *Hint:* Look at the end points of the runs. We'll take this up in a subsequent article.

Figure 1b

The T-sequence in Figure 1b is simpler than the one in Figure 1a. It consists entirely of the repeat shown in Figure 9.

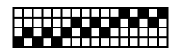


Figure 9. The Repeat in Figure 1b

There are many ways one can describe such sequences. We prefer to use runs, even when they are short, and to choose them in a way that repeats are most effective. Here's our version of the repeat:

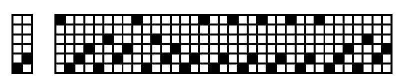
$$S = \overline{(1 \to 2)}^{\,3}, \overline{(3 \to 2)}^{\,2}, \overline{(3 \to 4)}^{\,3}$$

and the entire sequence is

$$\overline{S}^{\,11}$$

Figure 1c

The T-sequence in Figure 1c consists of the concatenation of two dissimilar sequences. Both consist of repeats. See Figure 10.



S *T*

Figure 10. The Repeats of Figure 1c

S is just $1 \to 2$. *T* is rather a mess:

$$T = U, V, 6, V, \overline{W}^{\,5}, (1 \to 3), V$$

where

$U = 6, 1 \to 3$
$V = 1, 4, 2, 3$
$W = 1, 2, 6$

Perhaps you can find a better formulation for *T*.

The entire sequence is

$$\overline{S}^{\,25}, \overline{T}^{\,3}$$

Figure 1d

The T-sequence in Figure 1d is an open palindrome. Figure 11 shows the sequence that forms the basis of the palindrome. There is a closed palindrome, *P*, within this sequence. See Figure 11.

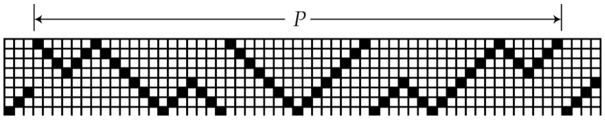


Figure 11. The Closed Palindrome in Figure 1d

The basis of this palindrome can be composed using yet more palindromes:

$$S = \overline{(8 \to 5)}, 7 \to 2, \overline{(1 \to 4)}, 8 \to 1$$

Now

$$T = 1 \to 3, \overline{S}, 1 \to 4$$

and the entire sequence is

$$\overline{T}$$

Figure 1e

The T-sequence in Figure 1e is a closed palindrome. Figure 12 shows the sequence that forms the basis of the palindrome. There is a closed palindrome, *T*, at the beginning of this sequence followed by the vertical reflection of *T*. See Figure 12.

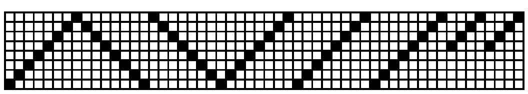


Figure 12. Basis for the Palindrome in Figure 1e

We can construct this sequence as follows:

$$S = (1 \to 8)$$
$$T = \overline{S}$$
$$U = \updownarrow T$$
$$V = T, U, \overline{S}^2, \overline{(5 \to 8)}^2$$

and the entire sequence is

$$\overline{V}$$

Implementing T-Sequence Operations

Implementing operations such as those described above has advantages beyond making it possible to carry them out quickly and accurately. Implementation often reveals ambiguities, inconsistencies, incomplete specifications, and conceptual problems. In what follows we'll have to deal with much that went unsaid (or unthought) in what has gone before.

Sequences as Data Values

In previous articles on sequences, we have represented them in programs in various ways. The most natural way to represent T-sequences is by lists or generators. Generators are more general and flexible than lists in many contexts. On the other hand, there is no *a priori* length or bound on the size of values associated with most generators. T-sequences, on the other hand, are finite, even though they may be derived from infinite sequences. T-sequences also are bounded by the number of shafts and treadles used. Of course, all finite sequences are bounded. In addition, there are many operations that can be performed on lists, such as concatenation and reversal, that cannot be performed, in general, on generators.

For these reasons, we choose to represent T-sequences by lists. We can, however, always get a list from a generator.

Data Types

We're dealing with two types of data: integers and sequences. We have implied, but not specifically stated, that the values in sequences are integers. We'll come back to this in a subsequent article, but for now, we'll take sequences to be composed of integers.

Questions now arise about the contexts in which these two types of data can occur.

There are several operations that have integer arguments. There are contexts, however, in which we have assumed without comment that integers can be interpreted as unit sequences. For example, concatenation was defined for sequences, but we've used 1, 2 as a sequence.

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia63/

In programming terms, using an integer where a sequence is expected amounts to promoting the integer to a sequence, much in the way that integers are promoted to real (floating-point) values in mixed-mode arithmetic, such as 5 + 2.3. In general, we will assume that an integer that occurs in a context where a sequence is expected is promoted to a unit sequence.

Here's a procedure to promote an integer to a sequence. If the argument, on the other hand, is a sequence, it is returned unchanged.

```
procedure spromote(x)

   if type(x) ~== "list" then x := [x]

   return x

end
```

The procedures that follow are written in a straightforward manner to illustrate what's involved. In some places more clever code could reduce their size and increase their speed. There also is no error checking.

The first operation on our list is the creation of runs:

```
procedure srun(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
```

The concatenation of sequences uses list concatenation after promoting the arguments:

```
procedure sconcat(x1, x2)

   return spromote(x1) ||| spromote(x2)

end
```

Since it is common to concatenate several sequences at a time, a more convenient form of sconcat() is:

```
procedure sconcat(x[])
   local lseq

   lseq := []

   every lseq |||:= spromote(!x)

   return lseq

end
```

A repeat is, of course, just successive concatenation:

```
procedure srepeat(x, i)
   local lseq

   x := spromote(x)

   lseq := copy(x)

   every 1 to i – 1 do
      lseq |||:= x

   return lseq

end
```

Notice that there is no default for i.

Extension is repetition with truncation:

```
procedure sextend(x, i)
   local lseq

   x := spromote(x)

   lseq := copy(x)

   until *lseq >= i do
      lseq |||:= x

   return lseq[1+:i]

end
```

Next come the procedures for creating palindromes:

```
procedure sopal(x)

   x := spromote(x)

   return x ||| sreflecth(x)[2:–1]

end

procedure scpal(x, i)
   local lseq

   /i := 1

   x := spromote(x)

   if i = 1 then return x ||| sreflecth(x)[2:0]
   else {
      lseq := srepeat(sopal(x), i)
      put(lseq, lseq[1])
      return lseq
      }

end
```

The procedures for reflection are:

```
procedure sreflecth(x)
  local lseq

  lseq := []

  every push(lseq, !spromote(x))

  return lseq

end

procedure sreflectv(x)
  local lseq, m

  x := spromote(x)

  m := sbound ! x

  lseq := []

  every put(lseq, m – !x + 1)

  return lseq

end
```

where the bound is given by

```
procedure sbound(args[])

  return sort(args)[–1]          # last is largest

end
```

Finally, we have the procedure for producing the length of a sequence:

```
procedure slength(x)

  return *spromote(x)

end
```

Using the Procedures

The following procedures construct the T-sequences shown in Figure 1.

```
procedure figure1a()
  local S

  S := sconcat(
    srun(1, 4),
    srun(1, 5),
    srun(2 ,6),
    srun(3, 6),
    srun(3, 7),
    srun(4, 8),
    srun(5, 8),
    srun(5, 7),
    srun(4, 6),
    srun(3, 6),
    srun(3, 5),
    srun(2, 4)
    )

  retrun sconcat(srun(1, 4), srepeat(S, 3))

end

procedure figure1b()
  local S

  S := sconcat(
    srepeat(srun(1, 2), 3),
    srepeat(srun(3, 2), 2),
    srepeat(srun(3, 4), 3)
    )

  return srepeat(S, 11)

end

procedure figure1c()
  local S, T, U, V, W

  S := srun(1, 2)

  U := sconcat(6, srun(1, 3))

  V := sconcat(1, 4, 2, 3)

  W := sconcat(1, 2, 6)

  T := sconcat(
    U,
    V,
    6,
    V,
    srepeat(W, 5),
    srun(1, 3),
    V
    )

  return  sconcat(srepeat(S, 25), srepeat(T, 3))

end

procedure figure1d()
  local S, T

  S := sconcat(
    scpal(srun(8, 5)),
    srun(7, 2),
    scpal(srun(1, 4)),
    srun(8, 1)
    )

  T := sconcat(
    srun(1,3),
    scpal(S),
    srun(1,4)
    )

  return sopal(T)

end

procedure figure1e()
  local S, T, U
```

```
  S := srun(1, 8)

  T := scpal(S)

  U := sreflectv(T)

  return scpal(
    sconcat(
      T,
      U,
      srepeat(S, 2),
      srepeat(srun(5, 8), 2)
      )
    )

end
```

On the Agenda

In the next article on T-sequences, we'll go on to more complicated T-Sequences, including those that combine different T-Sequences by interleaving.

References

1. "Tie-Ups and T-Sequences", *Icon Analyst* 61, pp. 5-9.

2. "Fractal Sequences", *Icon Analyst* 61, pp. 2-5.

Transposition Ciphers

Unlike substitution ciphers, transposition ciphers do not change the characters in a message but instead change their order.

A simple transposition cipher reverses the order of the characters, so that for example,

```
we attack at six
```

becomes

```
xis ta kcatta ew
```

Reversal is, of course, easy to detect.

Most transposition ciphers work on fixed-length segments of the message, using the same transposition for each segment. For example, successively reversing and concatenating two-character segments of the message above produces

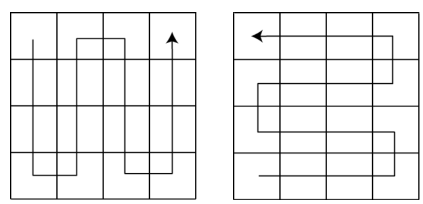
```
ewak ttca tas xi
```

Such ciphers require the length of the message to be an even multiple of the segment length. There are various ways to adjust the message length. Here is a procedure that extends a message if necessary by appending part of the reversal of the message to the end, where *i* is the segment length.

```
procedure rextend(message, i)
  local j

  j := *message % i

  if j = 0 then return message

  return message || reverse(message)[1+:(i – j)]

end
```

As is the case with most classical ciphers, the practical problems of enciphering and deciphering by hand played an important role in the methods that were used and limited the generality and sophistication of the techniques.

Most classical transposition ciphers use rectangular arrays into which characters are entered (written in) according to one rule and written out according to another. The rules constitute routes through the array. Figure 1 illustrates a pair of routes for a 4-by-4 array.



write in read out

Figure 1. A Transposition Cipher

Figure 2 shows the message above written in according to the route in Figure 1.

w	c	k	x
e	a		i
	t	a	s
a	t	t	

Figure 2. Message as Written In

When written out according to the route in Figure 1, resulting cryptogram is

```
att sat ea ixkcw
```

Routes do not have to be connected, although systematic and fairly simple routes were favored.

Figure 3 illustrates some other routes.

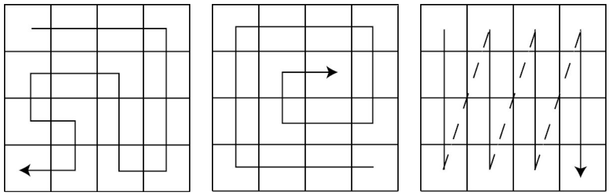


Figure 3. Transposition Routes

More elaborate routes use magic squares and knights' tours.

A pair of routes for reading in and writing out constitutes a key for a transposition cipher. The Nihilist transposition [1] adds an additional degree of security by using words that specify rearrangements of the columns and rows of the array after the message is written in, with the inverse rearrangements being applied before the message is written out. The lengths of the column and row words, after the removal of duplicate letters, must correspond to the number of columns and rows, respectively.

For our example, dumb and modem are possible words (modem being reduced to mode by deleting the second m). The letters are then converted to numbers according to their order in the alphabet. Thus, dumb becomes 2431 and mode becomes 3412. 2431 is interpreted as meaning column 2 moves to the first column, column 4 moves to the second column, column 3 to the third, and column 1 to the fourth. The interpretation of row rearrangements is the same.

Figure 4 shows the array of Figure 2 with columns and rows rearranged according to these numbers.

c	x	k	w
a	i		e
t	s	a	
t		t	a

t	s	a	
t		t	a
c	x	k	w
a	i		e

columns changed → rows changed

Figure 4. Example of a Nihilist Transposition

The result of reading out is

```
ai  ewkxct ta ast
```

Here is a procedure for converting words to rearrangement strings, where it is assumed that there are no more than nine rows or columns:

```
procedure worder(word)
   local chars

   chars := cset(word)

   return map(
      ochars(word),
      chars,
      &digits[2+:*chars]
      )

end
```

The procedure ochars() is from the Icon program library module strings:

```
# Order unique characters in order of first
#  appearance.

procedure ochars(w)
   local out, c

   out := ""

   every c := !w do
        if not find(c, out) then
           out ||:= c

   return out

end
```

This computation can be done more compactly and efficiently by tailoring the code to the problem rather than by using a library procedure. Compactness and efficiency, however, are of little importance in this case and using a library procedure saves programming and debugging time.

Transposition ciphers based on arrays are straightforward to implement using the matrix module in the Icon program library. There is, however, a more general and more interesting way.

Transposition ciphers can be characterized in terms of a pair of strings, one that labels the characters in the message by position and another that is the transposition of labels. Consider, for example, the write-in route shown in Figure 1 with a message consisting of the lowercase letters in order as shown in Figure 5.

a	h	i	p
b	g	j	o
c	f	k	n
d	e	l	m

Figure 5. Labeled Route Positions

Applying the Nihilist transposition described above, the rearranged rows and columns are shown in Figure 6.

h	p	i	a
g	o	j	b
f	n	k	c
e	m	l	d

f	n	k	c
e	m	l	d
h	p	i	a
g	o	j	b

columns changed → rows changed

Figure 6. Rearranged Routes

The result of writing out the labels according to the route in Figure 1 is

```
gojbaiphemldcknf
```

This string is a key that characterizes the entire transposition, independently of the way it was constructed.

Transpositions can be implemented by an easy if unobvious use of the function map().

The function map(s1, s2, s3) is, of course, the natural way to implement monoliteral substitutions in Icon [2]. For transpositions, s2 is the labeling of the positions in the message (such as the first 16 lowercase letters), s1 is the desired transposition of these labels, and s3 is the message. How a string can be reversed by this method is shown in Figure 7, which was taken from the third edition of *The Icon Programming Language* [3].

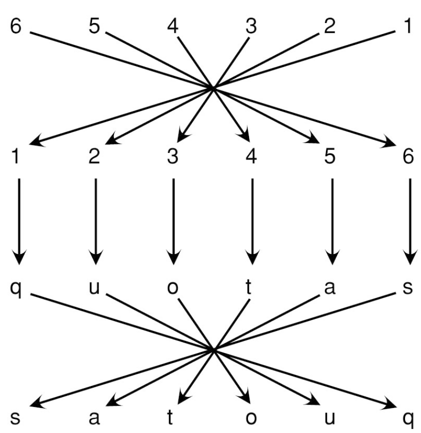


Figure 7. Transposition by Mapping

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

In this reversal, the strings are

```
s1 := "654321"                # transposition
s2 := "123456"                # labels
s3 := "quotas"                # string to reverse
```

and

```
s4 := map(s1, s2, s3)
```

produces

```
satouq
```

For our example above, we have

```
s1 := "gojbaiphemldcknf"      # transposition
s2 := "abcdefghijklmnop"      # labels
s3 := "we attack at six"      # message
```

and

```
s4 := map(s1, s2, s3)
```

produces the cryptogram

```
ai ewkxct ta ast
```

An interesting extension of this method is to add extraneous characters to the transposition string that are not in the label string, perhaps to further obscure the enciphered message. For the example above, the transposition string might be

```
s1 := "gotjbaiyphvewmldcknuf"
```

so that

```
s4 := map(s1, s2, s3)
```

produces the cryptogram

```
ait ewkyxcvtw ta asut
```

In deciphering,

```
s5 := map(s2, s1, s3)
```

the extraneous characters in s1 (t, u, v, w, and y) that do not appear in s2 do not participate in the mapping and hence drop out, producing the original message as before.

Summary

This concludes our series of articles on classical cryptography. Many ciphers can be constructed using the techniques we have described, including compound ciphers that first use one method to produce a cryptogram, then apply another method to the result, and so on. Typical compound ciphers use both substitutions and transpositions.

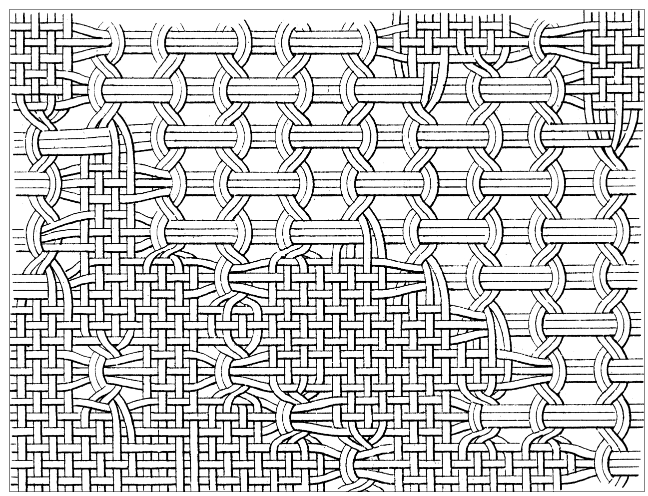
Classical cryptography now is largely in the province of puzzles, although scholars in some fields still find themselves having to deal with real cryptograms from times past.

References

1. *Cryptanalysis: A Study of Ciphers and Their Solution*, Helen Fouché Gaines, Dover, 1956, pp. 17-36.

2. "Classical Cryptography", Icon Analyst 59, pp. 7-9.

3. *The Icon Programming Language*, third edition, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, San Jose, California, 1996, p. 238.



What's Coming Up

There are two ways to write error-free programs; only the third one works.

— Alan Perlis

In the next issue of the Analyst, we plan to have another article on square-root palindromes, this time focusing on the results of solving continued fractions algebraically.

We'll continue our sequence of articles on T-sequences with more complicated examples and additional operations.

And we plan a **Graphics Corner** article on applying permutations to images.

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