The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1999
Number 57

In this issue

Shaft Arithmetic 1
Periodic Sequences 5
Finding Repeats 7
Name Drafting 11
Variations on Versum Sequences 15
Answers to Last Quiz 18
From the Library 19
What's Coming Up 20

Shaft Arithmetic

Editors' Note: This article was adapted from one designed as a tutorial for weavers without a technical background. We have added program material only near the end.

Shafts and treadles of looms are numbered for identification [1]. The numbers of the shafts through which successive warp threads pass form a sequence, as do the numbers of the treadles for successive picks. Consider the draft shown in Figure 1, in which the arrows indicate the orientation:

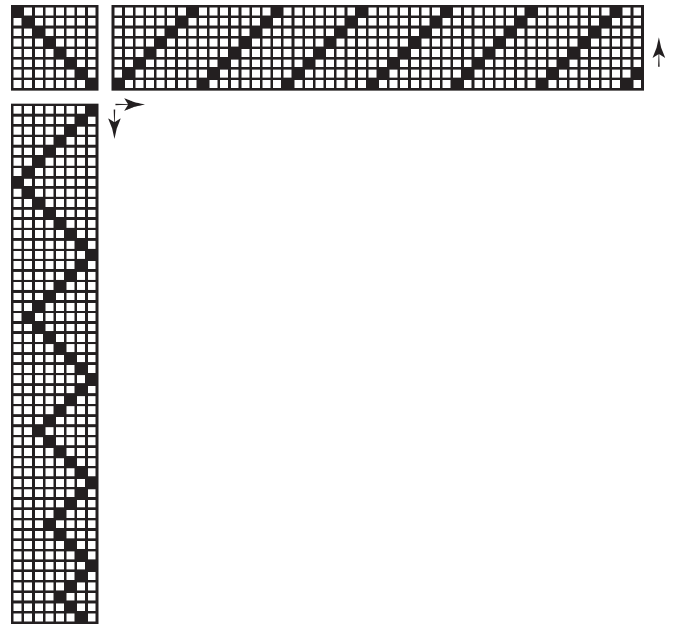


Figure 1. Example Draft

The threading is an upward straight draw. The sequence is:

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

The treadling sequence is more complicated:

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

These two sequences, in combination with the tie-up, define the structure of the weave.

Threading and treadling sequences often have distinctive patterns, as in the repeat for the threading sequence above. In the case of a repeat, it's only necessary to know the basic unit, which we'll indicate by an overbar:

$\overline{1,2,3,4,5,6,7,8}$

Modular Arithmetic

Since looms have a fixed number of shafts and treadles, the sequences are most easily understood

in terms of modular arithmetic, sometimes called clock or wheel arithmetic, in which numbers go around a circle clockwise, starting with 0. If there are 8 shafts, there are 8 equally spaced points on the circle 0 to 7, as shown in Figure 2:

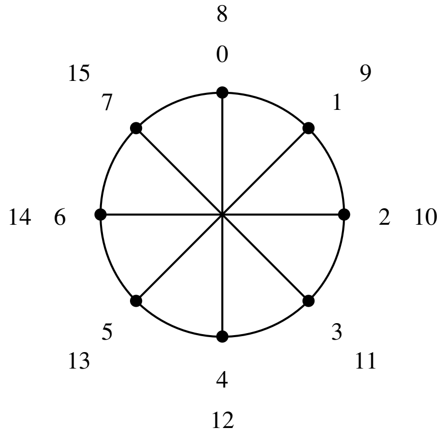


Figure 2. Arithmetic Modulo 8

The numbers on the inner circle are those that exist in the modular arithmetic. If we continue beyond 7, as shown in the outer ring, the numbers wrap around the wheel. Numbers on the same spoke are equivalent. For example, 0 and 8 are equivalent, 1 and 9 are equivalent, 2 and 10 are equivalent, and so on. Another way to look at it is that when 9 is introduced into modular arithmetic with 8 shafts, it *becomes* 1, and so on.

Shaft Arithmetic

Although modular arithmetic uses the number 0 as a starting point, most persons count from 1. Shafts and treadles are numbered this way. This 1-based numbering system is easily accommodated by rotating the wheel counterclockwise by one position, as shown in Figure 3:

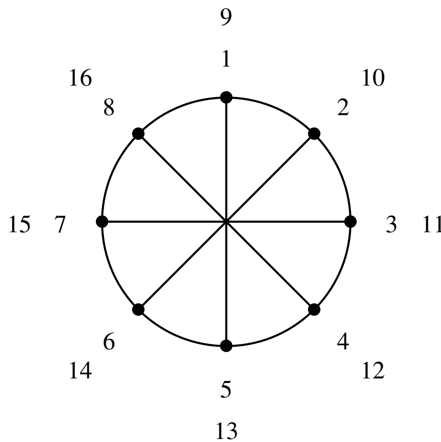


Figure 3. Shaft Arithmetic Modulo 8

Notice that 1 and 9 are still equivalent, as are 2 and 10, and so on. 0 has gone away, but it will be back.

For sequences, shafts and treadles are handled the same way, so we'll call this *shaft arithmetic*, with the understanding that it applies to treadles also. Of course, most facts about shaft arithmetic hold for ordinary modular arithmetic.

In shaft arithmetic, an upward straight draw for 8 shafts is described by the positive integers in sequence:

1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, …

and wrapped around the shaft circle to produce

1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, …

The point is that an upward straight draw comes from the most fundamental of all integer sequences, the positive integers in increasing order. (We'll discuss downward straight draws later.)

Drafting with Sequences

The idea behind drafting with sequences is that many sequences have interesting patterns, which often become more interesting in shaft arithmetic. In fact, many sequences show repeats when cast in shaft arithmetic. For example, the shaft sequence for an upward straight draw for 8 and 10 shafts are represented by

$$\overline{1,2,3,4,5,6,7,8}$$

and

$$\overline{1,2,3,4,5,6,7,8,9,10}$$

respectively.

Patterns in Sequences

Sequences may produce interesting woven patterns when they are used for threading and treadling.

There are a great many well-documented integer sequences. The Fibonacci sequence, which has many connections in nature, design, and mathematics, is one of the best known and most thoroughly studied of all integer sequences. The Fibonacci sequence starts with 1 and 1. Then each successive number (*term*) is the sum of the preceding two:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

As the sequence continues, the numbers get

very large. For example, the 50th term in the Fibonacci sequence is more than 12 billion. Shaft arithmetic brings this sequence under control. For 8 shafts, the result is

1, 1, 2, 3, 5, 8, 5, 5, 2, 7, 1, 8, 1, 1, 2, 3, 5, 8, 5, 5, 2, 7, 1, 8, 1, 1, 2, 3, 5, 8, 5, 5, 2, 7, 1, 8, …

As you can see, there is a repeat, so the entire sequence can be represented by

$$\overline{1,1,2,3,5,8,5,5,2,7,1,8}$$

Patterns in sequences are more easily seen if they are plotted, as in the grids used in weaving drafts. For 8 and 12 shafts, the Fibonacci sequence are shown in Figures 4 and 5:

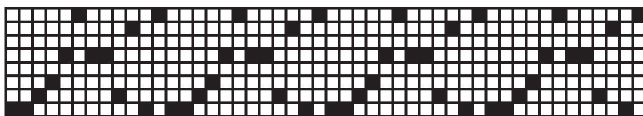


Figure 4. Fibonacci Sequence for 8 Shafts

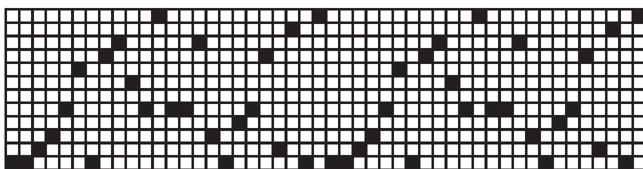


Figure 5. Fibonacci Sequence for 12 Shafts

Here are some other simple sequences and what they look like for various numbers of shafts.

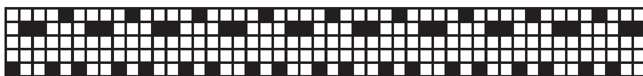


Figure 6. The Squares for 5 Shafts

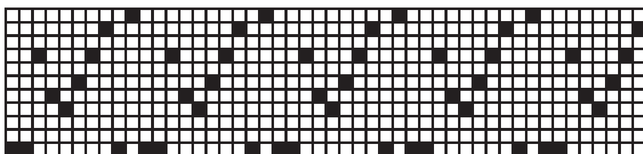


Figure 7. Fibonacci Cubes for 11 Shafts

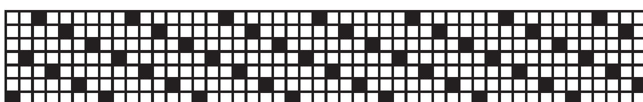


Figure 8. Every Third Positive Integer for 7 Shafts

The patterns such sequences produce in weaves depend on many factors. To keep things simple to begin with, we'll use direct tie-ups and treadling as drawn in (that is, the same sequence for the threading and the treadling) [2]. Even in this very limited framework, interesting woven patterns abound.

Figure 9 shows a drawdown for a few repeats of the Fibonacci sequence for 4 shafts:

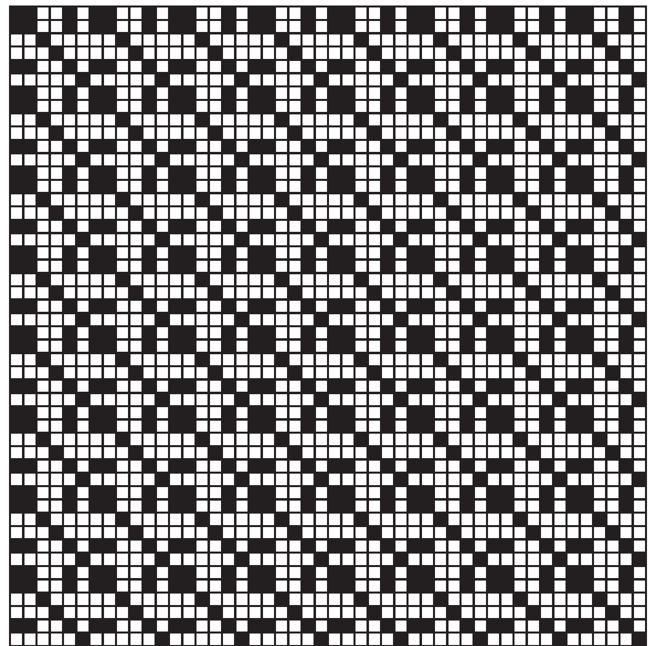


Figure 9. Fibonacci Drawdown for 4 Shafts

The pattern is noticeably different for 8 shafts, as shown in Figure 10. If you compare the two, however, you'll see commonalities:

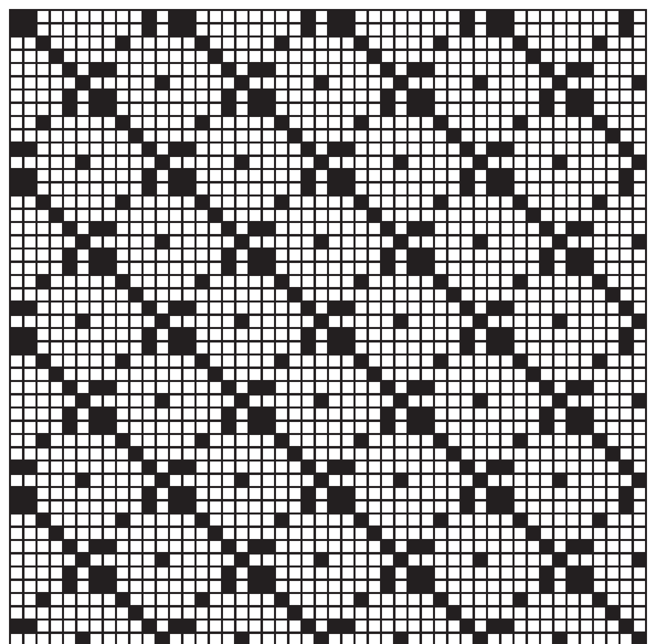


Figure 10. Fibonacci Drawdown for 8 Shafts

A simple sequence that produces interesting patterns is the "multi" sequence, which starts with a single 1 and is followed by 2 copies of 2, 3 copies of 3, and so on:

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

Note that there are no repeats in shaft arithmetic for this sequence, since the "width" of the repeated integer blocks constantly increases.

The drawdown for the multi sequence for 4 shafts is shown in Figure 11.

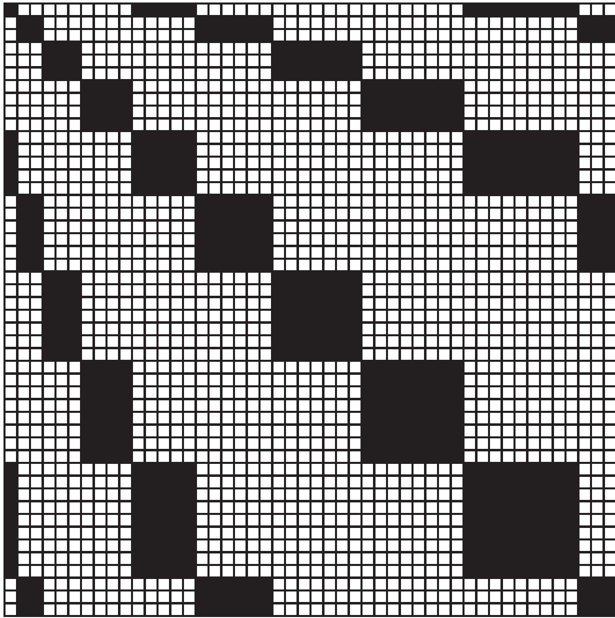


Figure 11. 4-Shaft Multi Sequence Drawdown

One way to produce interesting sequences is to combine other sequences, such as interleaving the terms of two sequences. For example, interleaving the positive integers and the Fibonacci sequence produces

1, 1, 2, 1, 3, 2, 4, 3, 5, 5, 6, 8, 7, 5, 8, 5, 1, 2, 2, 7, 3, 1, 4, 8, 5, 1, 6, 1, 7, 2, 8, 3, 1, 5, 2, 8, 3, 5, 4, 5, 5, 2, 6, 7, 7, 1, 8, 8 …

A drawdown for 8 shafts is shown in Figure 12.

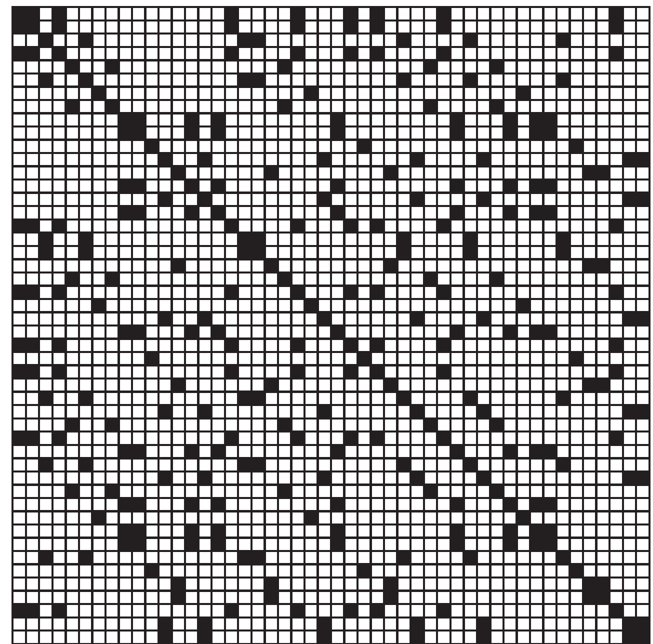


Figure 12. Interleaved Integer and Fibonacci Sequences for 8 Shafts

Other tie-ups, as well as threading sequences and treadling sequences that are different, produce all kinds of interesting results.

Zero and Negative Integers

There's one more matter to be dealt with — zero and negative numbers. Weavers drafting on the basis of sequence usually just drop such numbers or take the absolute values of negative numbers. The proper way to deal with these is indicated by looking at what happens when you have negative integers in increasing sequence as they cross over to the positive integers:

…, –7, –6, –5, –4, –3, –2, –1, 0, 1, 2, 3, 4, 5, 6, 7, 8, …

Now think of the modular wheel and what happens if you wrap this sequence of numbers around it. See in Figure 13.

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia57/

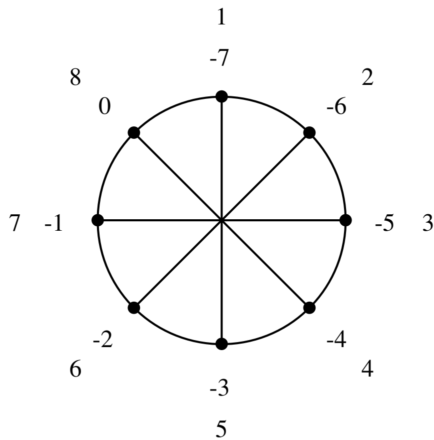


Figure 13. Negative Shaft Arithmetic Modulo 8

In other words, –1 becomes 7, –2 becomes 6, and so on. Note that 0, which we've been hiding, becomes 8.

Perhaps you now see the integer sequence that produces a downward straight draw:

0, –1, –2, –3, –4, –5, –6, –7, –8, –9, –10, …

All that's needed to convert a non-positive remainder to a shaft number is to add it to the number of shafts. For –1, for example,

8 + (–1) = 7

The Programming View

Icon's remaindering operation, i % j, produces the remainder of i divided by j. The sign of the result is the sign of i. Therefore, –7 / 3 produces –1. But the *common residue* (usually just residue) [3] in modular arithmetic is defined to be the remainder of i divided by j but given between 0 and j – 1. This is what the wheel shows.

A procedure to produce the residue is

```
procedure residue(i, j)
  i := i % j
  if i < 0 then i := j + i
  return i
end
```

This procedure can be modified to give results with indexing based on a number other than 0:

```
procedure residue(i, j, k)
  /k := 0
  i := i % j
  if i < k then i := j + i
  return i
end
```

Since k defaults to 0, if the third argument is omitted the usual residue is produced, but if k is 1, we get the *shaft residue*.

Incidentally, the underlying sequence for an upward straight draw is given by seq(1), while the sequence for a downward straight draw is given by –seq(0).

References

1. "A Weaving Language", Icon Analyst 51, pp. 5-11.

2. "Dobby Looms and Liftplans", Icon Analyst 55, pp. 17-20.

3. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1998, p. 281.

Periodic Sequences

A periodic sequence is an infinite sequence in which a finite subsequence repeats indefinitely. The digits of the mantissa (see the side-bar on the next page) of the decimal expansion of 1/7 provide an example:

1, 4, 2, 8, 5, 7, 1, 4, 2, 8, 5, 7, …

A periodic sequence may have a pre-periodic part before the repeat, as in the digits of the decimal expansion of 1/12:

0, 8, 3, 3, 3, 3 …

Sequences with pre-periodic parts are called *quasi-periodic*; those without pre-periodic parts are call *strictly periodic*.

There are many mathematical sources of periodic sequences. The main ones are:

- residues of terms of non-periodic sequences
- decimal (and other base) expansions of fractions
- denominators of continued fractions for quadratic irrationals
- samples of periodic functions like sin(x)

There probably are others we haven't thought of, and there's always the miscellaneous category.

Mantissa

If you look in a dictionary, the definition you'll most likely find for mantissa is that it's the decimal part of a logarithm.

In mathematics, the term has a more general meaning as the fractional part of a real number [1]:

$$\text{mantissa}(x) = x - \lfloor x \rfloor$$

where $\lfloor x \rfloor$ is the floor of x, the largest integer less than or equal to x.

The first use of the term mantissa in this way is attributed to Gauss.

A procedure to produce the mantissa of a real (floating-point) number would be trivial except for the possibility that the string representation may be in scientific notation, such as "2.45e–2".

This is just a messy detail of the kind that infests programming. Here's a procedure:

```
link numbers

procedure mantissa(r)
   local fpart

   r := real(r)

   fpart := r – floor(r)       # from numbers module

   fpart ?:= {
      tab(upto('.') + 1)
      tab(0)
      }

   fpart ? {
      if fpart := tab(upto('Ee')) then {
         move(1)
         if = "+" then fpart := "0"
         else {
            move(1)
            fpart := repl("0", tab(0) – 1) || fpart
            }
         }
      }

   return "." || fpart

end
```

Reference

1. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1998, p. 136.

There are many things of interest about periodic sequences: their periods, the values they contain, the patterns of values, and so on. But before we explore these areas, we need to discuss notation and the representation of periodic sequences in data and programs.

Notation and Representation

Sequences usually are written with terms separated by commas as shown in preceding examples. Sometimes other separators, such as blanks, are used, but commas make the separation of terms easier to see and we'll use commas here.

For periodic sequences, it's conventional to use a bar over the repeat, as in

$$\overline{1,4,2,8,5,7}$$

and

$$0,8,\overline{3}$$

Like many forms of mathematical notation, bars over text are typographically difficult. Word processors and page layout systems generally do not support them, since they cannot be composed from characters, unlike underscores, which come with font families. (Recall that we used underscores to indicate repeated digit patterns in versum numbers [1].) We've had to go to a program specifically designed for laying out mathematical expressions to provide the examples here.

When representing sequences as strings for processing by programs, neither overbars nor underscores are available. The string representation we chose is to enclose repeats in brackets, as in

```
"[1,4,2,8,5,7]"
```

and

```
"0,8,[3]"
```

Strings are awkward and inefficient to process in a program. For finite sequences we usually use lists, which are sequences by definition and might have been so named. Since the repeat in a periodic sequence is finite, we can represent periodic sequences by a pair of lists: a pre-periodic part (possibly empty) and repeat. A record brings these together in a single (defined) type:

```
record perseq(pre, rep)
```

Examples are

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

and

```
one_twelveth := perseq([0, 8], [3])
```

Note that a finite sequence can be represented in this way also by using an empty repeat, as in

```
one_eighth := perseq([1, 2, 5], [])
```

It's worth mentioning that for sequences consisting of single digits, strings could be used in place of lists, as in

```
one_twelveth_s := perseq("01", "3")
```

Several operations apply to both strings and lists. For example,

```
!one_twelveth
```

and

```
!one_twelveth_s
```

generate equivalent results, although the first produces integers and the latter one-character strings, which in numerical contexts are converted to integers automatically.

Although strings require less memory than lists, there are potential pitfalls and we'll generally avoid this "shortcut".

There is another possibility for representing the periodic part of a sequence — as a list whose last element points to the list itself. Thus,

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

can be visualized as shown in Figure 1.

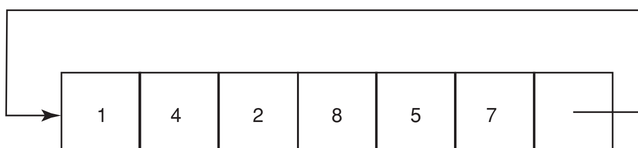


Figure 1. A Looping Structure

Programs that process such a representation need to take it into account, as in this procedure, which generates the elements of a repeat:

```
procedure genelem(rep)
  local x

  repeat {
    every x := !rep do {
      if type(x) == "list" then {
        rep := x
        break next                # go to repeat loop
        }
      else suspend x
    break                         # exit if finite
    }
end
```

There is no need for this extra complexity in our consideration of periodic sequences, but the representation is useful in more general contexts, which we'll discuss in a later article on "packet sequences".

Next Time

In the next article on periodic sequences, we'll explore the role of modular arithmetic in the creation of periodic sequences. See the article **Shaft Arithmetic**, which begins on page 1, for a hint of what's in store.

Reference

1. "Versum Factors", Icon Analyst 40, p. 9-14.

Finding Repeats

Given a finite portion of a sequence that is known to be periodic or that might be, how do you find the repeat?

In the first place, the problem is not well defined. For example, given the terms

0, 1, 2, 3, 4, 5, 6, 7

it might seem obvious that the next term is 8. However, if this sequence is the initial portion of the nonnegative integers mod 8, the next term is 0. The next term could, of course, be anything.

Even though the problem is not well defined, it's still possible to make useful guesses.

The method for finding a possible repeat is not conceptually difficult. You just try initial subsequences until one, when repeated, matches the rest of the sequence. If there is none, you remove the initial term and add it to a sequence for a pre-periodic part (initially empty) and start over. Eventually this process terminates, either with a possible repeat or with all the terms in the pre-periodic part. Here's a procedure:

```
link lists

record perseq(pre, rep)

procedure repeater(seq, ratio, limit)
  local init, i, prefix, results, segment, span

  /ratio := 2
```

```
/limit := 0.75

results := copy(seq)

prefix := []

repeat {
   span := *results / ratio
   every i := 1 to span do {
      segment := results[1+:i] | next
    if lequiv(lextend(segment, *results), results) then
         return perseq(prefix, segment)
      }
   put(prefix, get(results)) |     # first term to prefix
      return perseq(prefix, results)
  if *prefix > limit * *seq then return perseq(seq, [ ])
   }
end
```

The argument sequence is copied, so that it is not modified. The list prefix holds the potential pre-periodic part.

The variable ratio determines how long the repeat can be as a fraction of the length of the sequence and is designed to allow a reasonable determination of a repeat. The default, 2, ensures that the original sequence has at least two full repeats.

The variable limit prevents a very long a pre-periodic part with a short repeat at the end, which usually is erroneous.

In the repeat loop, initial subsequences from 1 to the allowed maximum are tried. For each, the subsequence is extended by repeating to the length of the sequence using lextend() from the lists module of the Icon program library.

If the two lists are equivalent, using lequiv(), also from the lists module, a possible repeat has been found and the procedure return with a record containing the pre-periodic part and the repeat.

If the two lists are not equal, the initial term of the current sequence is removed, appended to the pre-periodic part, and the loop is repeated. If the sequence is exhausted without finding a repeat, the procedure returns a record with all of the original sequence in the pre-periodic part and an empty repeat.

The procedure lextend() is a list version of the weaving procedure Extend() [1]:

```
procedure lextend(L, i)
   local result

   result := copy(L)

   until *result >= i do
      result |||:= L

   result := result[1+:i]

   return result

end
```

We'll come back to lequiv() later.

Figure 1 shows output from an instrumented version of repeater().

```
pre-periodic part:  []
remaining terms:  [1,10,3,5,1,1,3,5,3,1,1,10,1,1,3,5,3,1,1,10,1]

searching for repeat

trial segment:     [1]
extension:         [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]                    no match
trial segment:     [1,10]
extension:         [1,10,1,10,1,10,1,10,1,10,1,10,1,10,1,10,1,10,1,10,1]          no match
trial segment:     [1,10,3]
extension:         [1,10,3,1,10,3,1,10,3,1,10,3,1,10,3,1,10,3,1,10,3]             no match
                   …
trial segment:     [1,10,3,5,1,1,3,5,3]
extension:         [1,10,3,5,1,1,3,5,3,1,10,3,5,1,1,3,5,3,1,10,3]                 no match
trial segment:     [1,10,3,5,1,1,3,5,3,1]
extension:         [1,10,3,5,1,1,3,5,3,1,1,10,3,5,1,1,3,5,3,1,1]                  no match

attempt to find repeat failed
```

Figure 1. Finding a Repeat

```
moving initial term to pre-periodic part

pre-periodic part:  [1]
remaining terms:    [10,3,5,1,1,3,5,3,1,1,10,1,1,3,5,3,1,1,10,1]

searching for repeat

trial segment:      [10]
extension:          [10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10]   no match
trial segment:      [10,3]
extension:          [10,3,10,3,10,3,10,3,10,3,10,3,10,3,10,3,10,3,10,3]          no match
trial segment:      [10,3,5]
extension:          [10,3,5,10,3,5,10,3,5,10,3,5,10,3,5,10,3,5,10,3]             no match
                    …
trial segment:      [10,3,5,1,1,3,5,3,1]
extension:          [10,3,5,1,1,3,5,3,1,10,3,5,1,1,3,5,3,1,10,3]                 no match
trial segment:      [10,3,5,1,1,3,5,3,1,1]
extension:          [10,3,5,1,1,3,5,3,1,1,10,3,5,1,1,3,5,3,1,1]                  no match

attempt to find repeat failed

moving initial term to pre-periodic part

pre-periodic part:  [1,10]
remaining terms:    [3,5,1,1,3,5,3,1,1,10,1,1,3,5,3,1,1,10,1]

searching for repeat

trial segment:      [3]
extension:          [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3]                      no match
trial segment:      [3,5]
extension:          [3,5,3,5,3,5,3,5,3,5,3,5,3,5,3,5,3,5,3]                      no match
trial segment:      [3,5,1]
extension:          [3,5,1,3,5,1,3,5,1,3,5,1,3,5,1,3,5,1,3]                      no match
           …
trial segment:      [3,5,1,1,3,5,3,1]
extension:          [3,5,1,1,3,5,3,1,3,5,1,1,3,5,3,1,3,5,1]                      no match
trial segment:      [3,5,1,1,3,5,3,1,1]
extension:          [3,5,1,1,3,5,3,1,1,3,5,1,1,3,5,3,1,1,3]                      no match

attempt to find repeat failed

moving initial term to pre-periodic part

pre-periodic part:  [1,10,3]
remaining terms:    [5,1,1,3,5,3,1,1,10,1,1,3,5,3,1,1,10,1]

searching for repeat

trial segment:      [5]
extension:          [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]                        no match
trial segment:      [5,1]
extension:          [5,1,5,1,5,1,5,1,5,1,5,1,5,1,5,1,5,1]                        no match
trial segment:      [5,1,1]
extension:          [5,1,1,5,1,1,5,1,1,5,1,1,5,1,1,5,1,1]                        no match
                    …
trial segment:      [5,1,1,3,5,3,1,1]
extension:          [5,1,1,3,5,3,1,1,5,1,1,3,5,3,1,1,5,1]                        no match
trial segment:      [5,1,1,3,5,3,1,1,10]
extension:          [5,1,1,3,5,3,1,1,10,5,1,1,3,5,3,1,1,10]                      no match

attempt to find repeat failed
```

Figure 1 (continued). Finding a Repeat

```
moving initial term to pre-periodic part

pre-periodic part:  [1,10,3]
remaining terms:    [5,1,1,3,5,3,1,1,10,1,1,3,5,3,1,1,10,1]

searching for repeat

trial segment:      [5]
extension:          [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]             no match
trial segment:      [5,1]
extension:          [5,1,5,1,5,1,5,1,5,1,5,1,5,1,5,1,5,1]             no match
trial segment:      [5,1,1]
extension:          [5,1,1,5,1,1,5,1,1,5,1,1,5,1,1,5,1,1]             no match
                    …
trial segment:      [5,1,1,3,5,3,1,1]
extension:          [5,1,1,3,5,3,1,1,5,1,1,3,5,3,1,1,5,1]             no match
trial segment:      [5,1,1,3,5,3,1,1,10]
extension:          [5,1,1,3,5,3,1,1,10,5,1,1,3,5,3,1,1,10]           no match

attempt to find repeat failed

moving initial term to pre-periodic part

pre-periodic part:  [1,10,3,5]
remaining terms:    [1,1,3,5,3,1,1,10,1,1,3,5,3,1,1,10,1]

searching for repeat

trial segment:      [1]
extension:          [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]               no match
trial segment:      [1,1]
extension:          [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]               no match
trial segment:      [1,1,3]
extension:          [1,1,3,1,1,3,1,1,3,1,1,3,1,1,3,1,1]               no match
trial segment:      [1,1,3,5]
extension:          [1,1,3,5,1,1,3,5,1,1,3,5,1,1,3,5,1]               no match
trial segment:      [1,1,3,5,3]
extension:          [1,1,3,5,3,1,1,3,5,3,1,1,3,5,3,1,1]               no match
trial segment:      [1,1,3,5,3,1]
extension:          [1,1,3,5,3,1,1,1,3,5,3,1,1,1,3,5,3]               no match
trial segment:      [1,1,3,5,3,1,1]
extension:          [1,1,3,5,3,1,1,1,1,3,5,3,1,1,1,1,3]               no match
trial segment:      [1,1,3,5,3,1,1,10]
extension:          [1,1,3,5,3,1,1,10,1,1,3,5,3,1,1,10,1]             match

found repeat

pre-periodic part:  [1,10,3,5]
repeat:             [1,1,3,5,3,1,1,10]

done
```

Figure 1 (concluded). Finding a Repeat

As we mentioned earlier, the problem of finding a repeat is not well defined. The procedure may fail to find a repeat because there are not enough terms. More serious, perhaps, is a "false positive" in which a potential repeat is found but it is not a repeat in a longer portion of the entire sequence.

Neither of these problems can be avoided altogether, so it is well to treat the results with reservations.

Performance Issues

A brute-force approach like this can be very slow, especially for long sequences in which no repeat is found. As mentioned earlier, this may occur even when there is a repeat if the sequence given does not have enough terms. This is a caution

to the user to provide an adequate numbers of terms. The downside of this is that if there is no repeat, the procedure takes even longer.

If you look at Figure 1, you no doubt will see ways to improve the performance of the procedure. Suggestions are welcome. Send e-mail to

icon-analyst@cs.arizona.edu

Hidden in the library code is a source of inefficiency that has nothing to do with repeater(). The procedure lequiv() is designed to handle lists in their most general form, in which list elements can be of any type, included structures:

```
procedure lequiv(x,y)
  local i

  if x === y then return y
  if type(x) == type(y) == "list" then {
    if *x ~= *y then fail
    every i := 1 to *x do
      if not lequiv(x[i], y[i]) then fail
    return y
   }

end
```

For lists of numbers, this generality is not needed and the following somewhat faster procedure will do:

```
procedure seqequiv(seq1, seq2)
  local i

    every i := 1 to *seq1 do
      if seq1[i] ~= seq2[i] then fail

    return seq2

end
```

This improvement is, of course, minor compared to the combinatorial nature of the problem.

Reference

1. "A Weaving Language", *Icon Analyst* 51, pp. 5-11.

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

Name Drafting

Many handweavers simply weave from the large number of drafts that are available in books and magazines about weaving. These weavers may make minor modifications, but the designs they weave are the creations of others.

The measure of "real" handweavers is the desire and ability to create their own designs.

Weavers who have woven only from the drafts of others often come to the point where they want to design their own drafts — to become "real" handweavers. But how to start?

A type of weaving known as *name drafting* often is recommended for this situation. (Name drafting also is known as name code drafting, code drafting, commemorative drafting, and personalized design.)

Although name drafting is naive in concept, as you'll see, it does provide an easy bridge between copying the work of others and creating new designs.

Mapping Strings into Draft Sequences

The basic idea is simple: A string — a word, or more often, a phrase or sentence — is coded to make shaft and treadling sequences. Such drafts usually are treadled as draw in, with the same sequence used for both the threading and treadling, so we'll just refer to threading sequences here.

The coding assigns a shaft number to each character of the selected string. Although any method of associating shafts with characters could be used, only a few appear in the literature [1-6] and weavers generally are instructed to use one of these. Three codings that commonly are used for four shafts are:

ABCDEFG shaft 1
HIJKLMN shaft 2
OPQRSTU shaft 3
VWXYZ shaft 4

ABCDEF shaft 1
GHIJKL shaft 2
MNOPQR shaft 3
STUVWXYZ shaft 4

AEIMQUY shaft 1
BFJNRVZ shaft 2
CGKOSW shaft 3
DHLPTX shaft 4

Using a specified coding formula is an example of the dominating role of rote among unsophisticated weavers. It also is telling that only letters are considered and that upper- and lowercase letters always are taken to be equivalent. This is akin to the problem of a person who is not familiar with computing and has trouble with the fact that a blank is just a much character as X. Surprisingly, to this day this problem exists with beginning computer science students.

One problem in choosing a mapping between characters and shaft numbers is whether some shafts will be underutilized or not used at all. There are strong statistical patterns in the frequency in which characters appear in written text (usually considered only in terms of letters). Average frequencies vary with the subject and the language. It's well known that in English, e is the most commonly used letter and q and z are the least.

Letter frequency is an important aspect of some kinds of cryptography and we'll discuss it in more detail in that context in an upcoming article.

The mapping can be chosen to try to balance shaft usage, but any predefined mapping can be defeated by a particular string — not to mention the fact that the string chosen may not contain as many different characters as there are shafts to be used. In practice, strings are chosen to work around such problems.

Modifying Sequences for Weaving

Name drafts usually employ a kind of weaving called *overshot* [7-8] in which a pattern is woven over a background texture. A technical requirement of overshot is that the shaft numbers alternate between odd and even. This problem is solved by adding "incidentals" where necessary to break odd and even pairs that arise from the coding.

The result usually is better if this is done in a systematic way. OddEvenPDCO{} in Icon Analyst 55 [9] works nicely (and corresponds to what name drafters usually do, although the directions for doing it are often are given on a case-by-case basis and fail to reveal a general method). The idea is simple: When a prohibited pair occurs, insert a shaft number one greater than the first member of the pair, wrapping around where necessary using shaft arithmetic (see the article that starts on page 1).

Thus, for four shafts, the sequence

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

becomes

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

In practice, weavers often make other modifications to produce more attractive weaves after small trial weaves (called samples). We won't get into that here, since there is no system to it.

Implementing Name Drafting

To implement name drafting, we generalized the conventional interpretation of characters to

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

© 1999 by Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend

include all characters, not just letters. Making upper- and lowercase letters equivalent or disregarding some characters is done by applying an appropriate function to the chosen string. For example,

```
string := map(string)
```

maps uppercase letters to lowercase ones, leaving all other characters unchanged. There are many other relevant uses of map(), including transpositions [10].

Other functions may be useful, such as

```
string := cset(string)
```

which removes duplicate characters and puts the results in lexical order.

The Icon program library module strings contains several procedures that may be useful in this context:

compress(s, c) compresses runs of characters in c that occur in s to a single character.

csort(s) sorts the characters of s but does not remove duplicates.

deletec(s, c) deletes characters in c from s.

fchars(s) orders the characters in s according to decreasing frequency of occurrence.

ochars(s) places the unique characters of s in the order in which they first occur.

Procedures can be written to produce various other effects, including adapting the mapping to the string chosen to balance shaft usage.

The next step is to assign a positive integer to each distinct character of the string. Here's a procedure. Note that it assigns integers in the order in which characters occur.

```
procedure shaftmap(s)
   local j, map_table

   map_table := table()

   j := 0

   every /map_table[!s] := (j +:= 1)

   return map_table

end
```

The table returned then can be used for the actual mapping. Notice that at this point, the result is independent of the number of shafts. When the draft is created, shaft arithmetic is applied to bring the values in range.

The mapping table then can be applied to any string. This procedure generates the shaft numbers:

```
procedure genshafts(s, tbl)

   suspend tbl[!s]

end
```

The two processes can be combined:

```
procedure genmapshafts(s1, s2)

   suspend genshafts(s1, shaftmap(s2))

end
```

Other Aspects of Name Drafting

Name drafts usually are reflected about their centers to add symmetry and increase the visual appeal of the resulting weaves.

As mentioned earlier, name drafting usually is done using an overshot weave. In overshot weaves, the tie-up usually is a twill, which, in its simplest form, produces a diagonal surface effect as shown in Figure 1.

Figure 1. A Twill

The particular twill tie-up used may have a dramatic effect on a weave produced by a name draft. We don't have space here to explore twills, but we'll get to them in a later article.

Name Drafting in Perspective

Certainly name drafting is an *ad hoc* mechanism for producing threading and treadling sequences. Other mechanisms are easy to imagine. In fact, one of the main subjects we'll treat in upcoming issues of the Analyst is drafting based on integer sequences. Name drafting is just one way of getting an integer sequence. See the article **Shaft Arithmetic** that begins on page 1 for examples.

To weavers, however, name drafting can serve a real purpose, which is indicated by the alternative term "commemorative drafting". The string chosen may have a meaning that is personal to the weaver, resulting in a weave embodying this mean-

ing. This aspect of name drafting is sometimes forgotten, however. A recent article on name drafting [6] described the author's attempts to find a phrase that produced an attractive weave, finally settling on "The Random House Dictionary" as the result of glancing at a nearby bookshelf. An attractive weave, yes. A special meaning? Hardly (even according to the author).

Next Time

As mentioned above, we'll explore twills in the future issue of the Analyst. In the meantime, we'll leave you with the name-drafted images in Figure 2. Don't try to figure out the strings used. To have any hope of deciphering a name draft, you need to know the tie-up used. We'll "reveal all" in the next article.

References

1. *A Handweaver's Notebook*, Helen G. Thorpe, Collier Books, 1956, pp. 153-156.

2. *Master Weaver Library*, S. A. Zielinski, Leclerc, 1979, Vol. 17, pp. 65-67.

3. *The Weaving Book: Patterns and Ideas*, Helene Bress, Scribners, 1981, pp. 227-228, 282, 307, 310.

4. "A New HGA Name Draft", Ena Marston, *Shuttle, Spindle and Dyepot*, Number 47, Summer 1981, pp. 43-45.

5. "Commemorate with a Name Draft", Norma Smayda, *Shuttle, Spindle and Dyepot*, Number 91, Summer 1992, pp. 43-45.

6. "How to Weave Name Drafts", Christina Hammel, *Handwoven*, November/December 1997, pp. 35-37.

7. *Learning to Weave*, Deborah Chandler, Interweave Press, 1995, pp. 191-198.

8. *The Complete Book of Drafting for Handweavers*, Madelyn van der Hoogt, Shuttle Craft Books, 1994, 39-51.

9. "Operations on Sequences, Icon Analyst 55, pp. 10-13.

10. *The Icon Programming Language*, 3rd edition, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer, Inc., 1996, pp. 237-245.

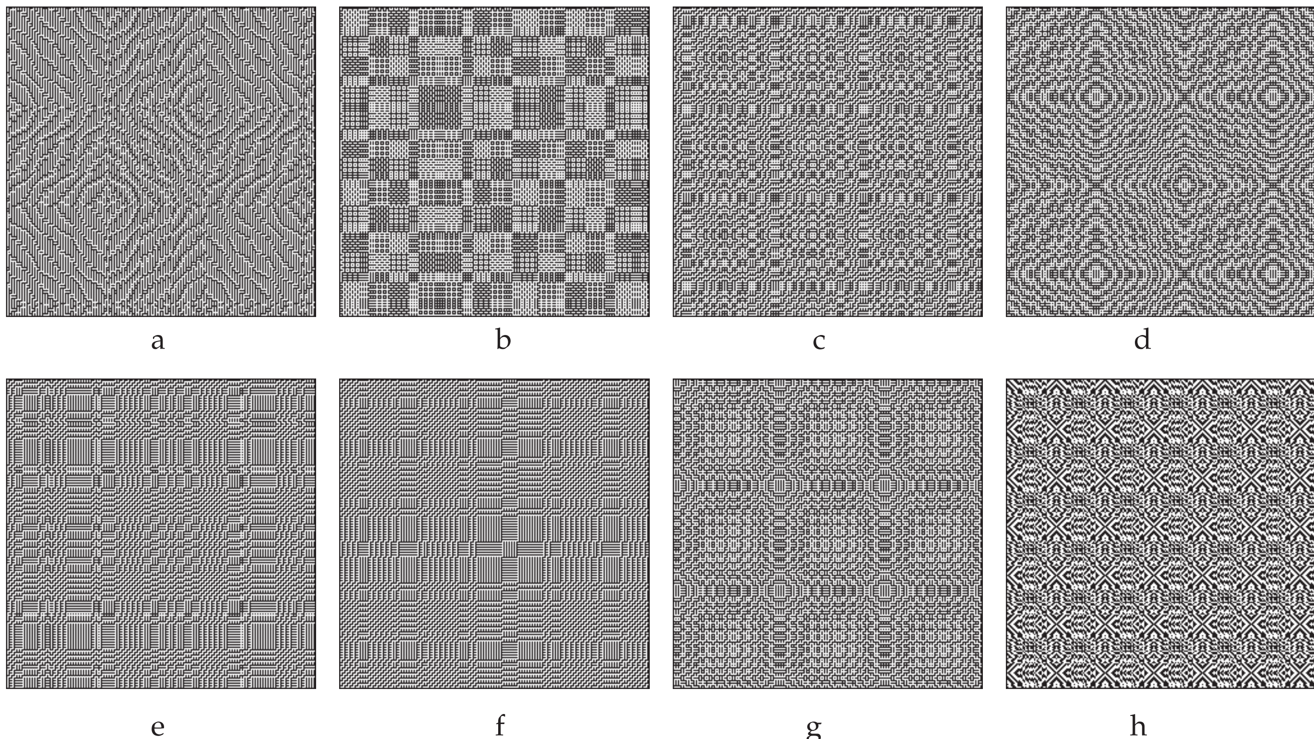


Figure 2. Weaves from Name Drafts

Variations on Versum Sequences

There have been 18 Analyst articles on versum sequences, those sequences that result from repeatedly adding the reversal of a number to itself [1-18]. It's been a year since the last article, not because we lack material but because we had other things to cover and thought a respite from versum sequences was in order.

This article doesn't include anything from our older unpublished versum material but rather introduces some variations on the reverse-addition process.

One variation is characterized by "reverse, add, and then add *j*" where *j* is a fixed integer. For example, with the seed 196, adding 7 produces this sequence:

894	1279101088
1399	10080120816
11337	71882228824
84655	114764457648
140310	961518925066
153358	1622048740242
1006716	4042527142510
7182724	4194944394921
11455548	5489878889842
96010966	7979767679694
162912042	12949535359498
403131310	102444888954426
416262621	726904777398634
542525242	1163798554808268
785050494	9791883113781886
	…

The procedure versumseq(i) from the genrfncs module of the Icon program library was originally designed to generate the ordinary versum sequence. It can easily be generalized to take j as a second argument:

```
procedure versumseq(i, j)

   /i := 196

   /j := 0

   repeat {
      i +:= reverse(i) + j
      suspend i
      }

end
```

Note that i defaults to the infamous 196, while j defaults to 0, so that if the second argument in a call of versumseq() is omitted, the ordinary versum sequence is generated.

There are seveal questions we might ask about this generalization to versum sequences:

- How do such sequences depend on the value of *j*?
- What happens if *j* is negative?
- Do such sequences contain palindromes in the fashion of regular versum sequences?

One of the featured sequences in *The Encyclopedia of Integer Sequences* [19] <1> is characterized by "reverse, add, then sort" (RATS). For example, starting with the seed 1, the sequence is:

2	12333445
4	66666677
8	133333444
16	556667777
77	1233334444
145	5566667777
668	12333334444
1345	55666667777
6677	123333334444
13444	556666667777
55778	1233333334444
133345	5566666667777
666677	12333333334444
1333444	55666666667777
5567777	123333333334444
	…

A procedure to generate such sequences is simple:

```
procedure ratsseq(i)

   /i := 196

   repeat {
      i +:= reverse(i)
      i := integer(csort(i))
      suspend i
      }

end
```

The procedure csort(), from the strings module of the Icon program library, sorts the characters of a string.

RATS sequences raise all kinds of questions, such as:

- Do they ever contain repdigit terms (terms consisting entirely of one digit)?
- Are terms ever pandigital (containing at least one of every digit except, in this case, 0)?

This procedure can be generalized to allow an optional unary operation to be specified:

```
procedure versumopseq(i, p)

  /i := 196

  /p := csort

  repeat {
    i +:= reverse(i)
    i := integer(p(i))
    suspend i
    }

end
```

A further generalization allows for operations with more than one argument:

```
procedure versumopseq(i, p, args[])

  /i := 196

  /p := ochars

  push(args)          # make room for first argument

  repeat {
    i +:= reverse(i)
    args[1] := args   # make i first argument
    i := integer(p ! args)
    suspend i
    }

end
```

For example, versumopseq(1, rotate, 1) rotates the reversal left one digit and produces the following sequence:

2	219991
4	199034
8	300256
61	522599
77	5178241
541	6069566
866	27291721
5341	109934
7766	498355
44431	522491
78755	167167
345421	289289
699644	2722711
1466401	8949833
5130422	23393311
3707377	47326433
14444501	7888078
49889422	65969651
23883167	16666078
37327391	…

Reduction in the numbers of digits occurs when zeros are shifted into leading positions.

Here we might ask if there is a limit to the size of terms.

Incidentally, procedure versumopseq() subsumes ratsseq(), since versumopseq(i, csort) peforms the required operation.

Note that versumopseq(), as written, does not check that p is a valid operation.

There are many other possible variations on the reverse-addition process, such as adding the number of digits to the result or adding the term number to the result.

But what is the point of all this? There are infinitely many variations. Ordinary versum sequences are of interest because of palindromes. What about the others?

If you look at recent Analyst articles on weaving, you'll see the emergence of sequences as an important tool in drafting interesting weaves. Do versum sequences produce interesting weaves? Do variations on versum sequences produce interesting weaves? In a related question, do the residues of versum sequences yield periodic sequences?

We'll address these question in future articles. For now, we'll leave you with some weaves based on versum sequences, both ordinary and with variations, as shown in Figure 1.

References

1. "The Versum Problem", Icon Analyst 30, pp. 1-4.

2. "The Versum Problem", Icon Analyst 31, pp. 5-12.

3. "Equivalent Versum Sequences", Icon Analyst 32, p. 1-6.

4. "Versum Sequence Mergers", Icon Analyst 33, pp. 6-12.

5. "Versum Base Seeds", Icon Analyst 34, p. 6.

6. "Versum Palindromes", Icon Analyst 34, pp. 6-9.

7. "Versum Numbers", Icon Analyst 35, pp. 5-11.

8. "Versum Predecessors", Icon Analyst 37, pp. 11-15.

9. "Versum Bimorphs", Icon Analyst 39, pp. 10-13.

10. "Versum Factors", Icon Analyst 40, pp. 9-14.

11. "Factors of Versum Numbers", Icon Analyst 43, pp. 9-14.

12. "Versum Numbers as Factors", Icon Analyst 45, pp. 12-16.

13. "Versum Primes", Icon Analyst 46, pp. 12-16.

14. "Assault on Mount Versum", Icon Analyst 47, pp. 1-5.

15. "Assault on Mount Versum", Icon Analyst 48, pp. 7-9.

16. "Versum Deltas", Icon Analyst 49, pp. 6-11.

17. "Versum Deltas", Icon Analyst 50, pp. 7-11.

18. "Generating Versum Numbers", Icon Analyst 51, pp. 16-20.

19. *The Encyclopedia of Integer Sequences*, N. J. A. Sloane and Simon Plouffe, Academic Press, 1995.

Link

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

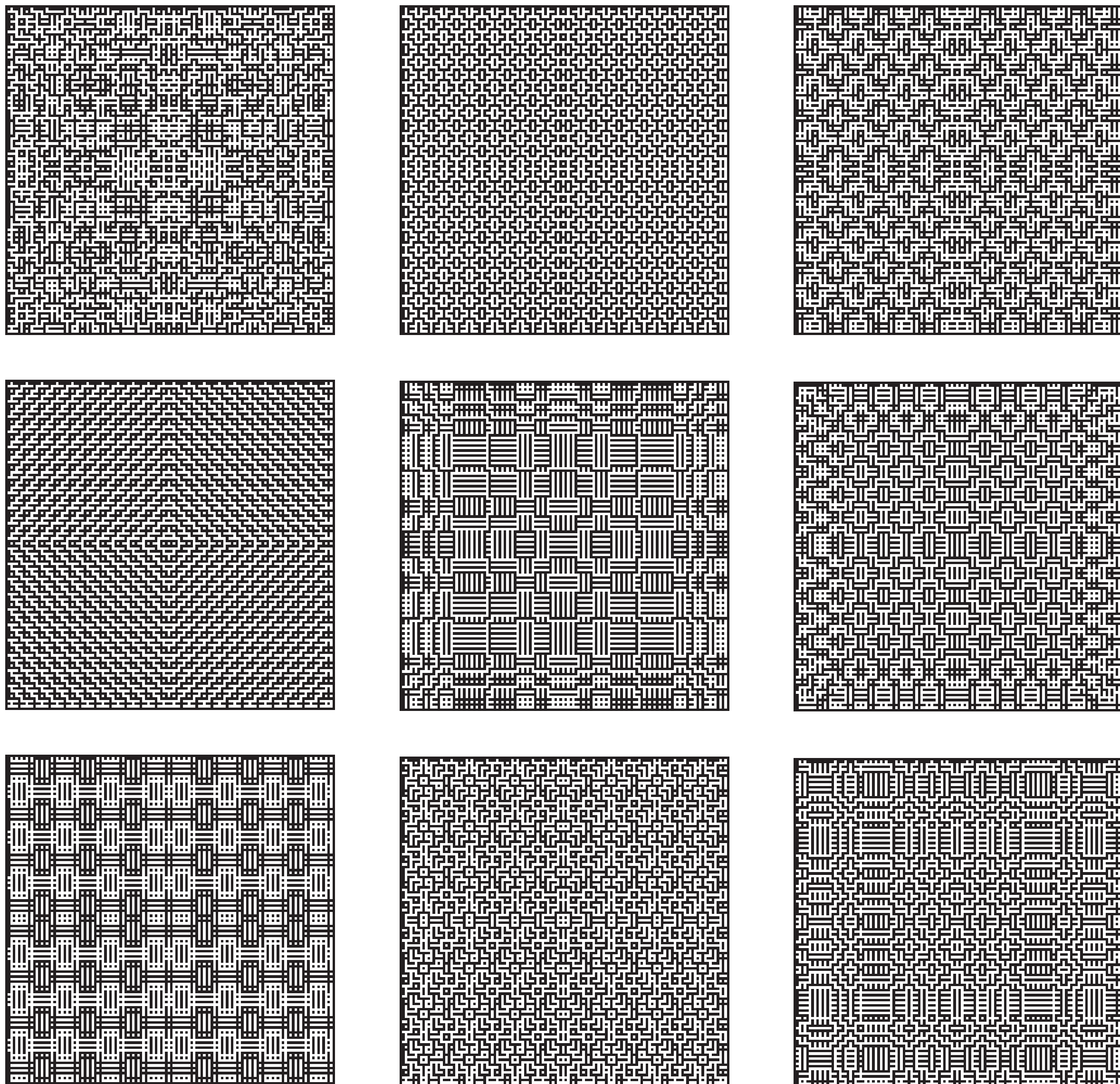


Figure 1. Versum Weaves

Answers to Quiz on Pointer Semantics

See Icon Analyst 56, page 17, for the questions.

1.

(a)

```
L :=[]
put(L, L)
```

(b)

```
L1 := []
put(L1, L1)
L2 := [L1]
```

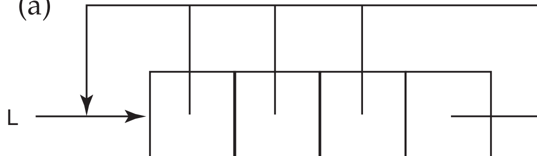
(c) Same as (b) — just drawn differently

(d)

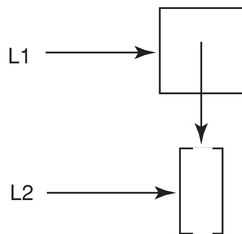
```
L2 := []
L1 := [L2]
put(L2, L2, L1)
```

2.

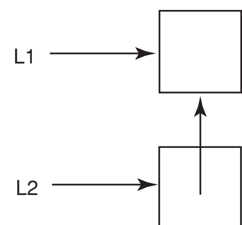
(a)

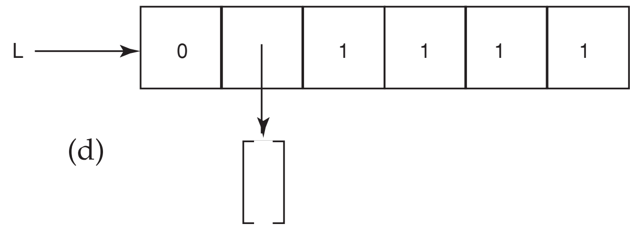


(b)



(c)





(d)

In these diagrams, [ ] indicates an empty list.

It's very easy to make mistakes when working with pointers, especially when the structures involve loops.

There's a very useful procedure in the Icon program library for situations like this: ximage(). We described it in detail in an early **From the Library** article [1]. Here's a program that shows the successive structures in question 2(d):

```
link ximage

procedure main()

   L1 := list(5, 1)
   write(ximage(L1))
   write()
   push(L1, [], L1)
   write(ximage(L1))
   write()
   L1[1] := 0
   write(ximage(L1))
   write()
   pull(L1)
   write(ximage(L1))

end
```

The output of this program is:

```
L1 := list(5,1)

L1 := list(7,1)
   L1[1] := L1
   L1[2] := L5 := list(0)

L1 := list(7,1)
   L1[1] := 0
   L1[2] := L5 := list(0)

L1 := list(6,1)
   L1[1] := 0
   L1[2] := L5 := list(0)
```

Note that the output of ximage() is in the form of executable expressions that can be used to build the structures.

Reference

1. "From the Library — Structure Images", Icon Analyst 25, pp. 1-5.

From the Library — Rational Arithmetic

A rational number is just a fraction, the ratio of two integers, p/q, where p and q are integers and $q \neq 0$.

Many numerical computations can be done using floating-point approximations to rational numbers. For example, the value of

46368.0 / 75025.0

is approximately 0.6180339887, which is quite close to 46368/75025 numerically.

However, you cannot recover 46368/75025 with any certainty from the floating-point value shown above. Several other fractions, such as 121393/196418, give the same floating point value.

For exact computations involving fractions, the Icon program library provides the module rational.

Data Representation

As in all cases like this, it is necessary to provide a standard representation of rational numbers as strings — if only for input and output. The form used for rational numbers consists of two integers separated by slashes and surrounded by parentheses, as in "(46368/75025)". The parentheses isolate rationals in strings from any surrounding string context in which they may be placed. The integers may be signed, as in "(–46368/75025)".

For computation in programs, rational numbers are represented as records:

```
record rational(numer, demon, sign)
```

The sign is 1 or –1 depending on whether the rational number as a whole is positive or negative. Using 1 and –1 allows sign computation by multiplication.

Records for rationals produced by the procedures in rational always are in a canonical form in which the numerator and denominator are positive and reduced to lowest terms (that is, with no common divisor grater than 1). For example, "(6/–14)" is converted to

```
rational(3, 7, –1)
```

The module rational contains the following procedures for converting between types:

rat2str(r)	convert rational to string
str2rat(s)	convert string to rational
rat2real(r)	convert rational to real (floating-point)
real2rat(x)	convert real (floating-point) to rational

There are five procedures for performing rational arithmetic:

addrat(r1, r2)	add rationals
divrat(r1, r2)	divide rationals
mpyrat(r1, r2)	multiply rationals
negrat(r)	form negative of rational
reciprat(r)	form reciprocal of rational

In addition, ratred(r) performs error checking and reduces a rational to its lowest terms.

Problems with Zero

Zero is not allowed as a denominator in rational numbers since division by zero is undefined. A zero denominator may come about from conversion of a string or by division (and, equivalently, forming a reciprocal). If a zero appears for a denominator, a user-defined run-time error occurs.

Zero is allowed as a numerator, but $0/n$ has the same value for all $n \neq 0$. Consequently, if a zero appears for a numerator, rational(0, 1, 1) is produced.

Problems with User-Supplied Rationals

Although the procedures in the module rational always produce values in canonical form, there

is nothing to prevent a user from creating a rational record that is not in canonical form or is erroneous. Possible examples are

```
rational(5, 50, 1)
rational(–5, –2, –1)
rational(0, 0, 1)
rational(2.5, 3.2, 1)
rational(3, 7)
rational("10x", 5, 1)
```

Handling all possible cases is messy. The details are relegated to ratred(), which is called by other procedures in rational to make sure their arguments are legal and in proper form.

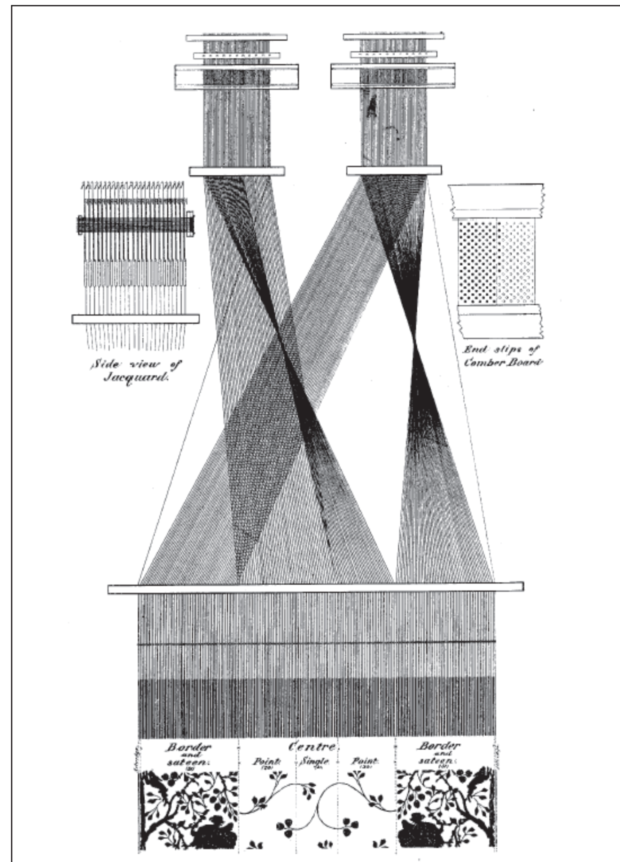
Example Procedures

Typical procedures are:

```
procedure addrat(r1, r2)
   local denom, numer, div, sign

   r1 := ratred(r1)
   r2 := ratred(r2)

   denom := r1.denom * r2.denom
   numer := r1.sign * r1.numer * r2.denom +
      r2.sign * r2.numer * r1.denom

   if numer = 0 then return rational(0, 1, 1)

   if numer * demon >= 0 then sign := 1
      else sign := –1

   numer := abs(numer)
   denom := abs(denom)

   div := gcd(numer, denom)

   return rational(numer / div, denom / div, sign)

end

procedure str2rat(s)
   local div, numer, denom, sign

   s ? {
      ="(" &
      numer := integer(tab(upto('/'))) &
      move(1) &
      denom := integer(tab(upto(')'))) &
      pos(–1)
      } | fail

   if denom = 0 then runerr(510, 0))
   if numer = 0 then return rational(0, 1, 1)

   if numer * denom >= 0 then sign := 1
      else sign := –1

   numer := abs(numer)
   denom := abs(denom)

   div := gcd(numer, denom)

   return rational(numer / div, denom / div, sign)

end
```

New Version of rational.icn

As often happens, writing articles about programs results in their revision. The current version of the module rational is on the Web site for this issue of the Analyst.



What's Coming Up

We had expected to have an article on weavable color patterns for this issue of the Analyst, but we ran out of space and time. This article has high priority for the next issue.

We'll continue the series on periodic sequences with ones that result from modular arithmetic.

We've been planning a series of articles on "classical" cryptography for some time. That's on the table for the next issue.

In **From the Library**, we'll continue the survey of the basic modules in the Icon program library.