The Icon Analyst

In-Depth Coverage of the Icon Programming Language and Applications

June 2001
Number 66

In this issue

The Last Issue of the Analyst

This is the last issue of the Icon Analyst. 11 years, 66 issues, 917 pages.

As we've said before, when we started, we had no way to know where the Analyst would go or how long it would last.

In theory, the Analyst could go on indefinitely — or at least until it ran out of readers and editors. In practice, as much fun and rewarding as it has been, the Analyst takes too much time — time we prefer to spend on other things.

We want to thank you, our readers, for your interest and loyalty to the Analyst. Many of you were charter subscribers and have been with us for all 11 years — you are a special group.

A few of you have subscriptions that run beyond this last issue. If you are one of these persons, you'll find enclosed information on how to get a refund or otherwise use your balance.

Spectra Sequences

Given an irrational number α, the integer sequence $\lfloor\alpha\rfloor, \lfloor 2\alpha\rfloor, \lfloor 3\alpha\rfloor, \ldots$, where $\lfloor x\rfloor$ is the floor x, is called the *spectrum sequence* of α. For example, the spectrum sequence of π is 3, 6, 9, 12, 15, 18, 21, 25, 28, 31, 34 … and the spectrum sequence of e is 2, 5, 8, 10, 13, 16, 19, 21, 24, 27, 29, … .

We'll denote the spectrum sequence of x by $\mathcal{S}(x)$.

Generating spectra sequences is easy:

```
procedure spectseq(r)
  suspend integer(seq() * r)
end
```

Beatty Sequences

A very interesting case occurs for two positive irrational numbers α and β such that

$$1/\alpha + 1/\beta = 1$$

Then $\mathcal{S}(\alpha)$ and $\mathcal{S}(\beta)$ together contain all the positive integers without repetition. These are called *Beatty sequences* after Samuel Beatty, who discovered their remarkable property.

Note: This formulation is by Weisstein [1]. Superficially this gives the impression that α and β are independent. However, given α, $\beta = \alpha/(\alpha - 1)$. Similarly, given β, $\alpha = \beta/(\beta - 1)$. It would seem more straightforward to say that, given a positive irrational number α, $\mathcal{S}(\alpha)$ and $\mathcal{S}(\alpha/(\alpha - 1))$ are Beatty sequences that together contain all the positive integers without repetition. The catch is that α must be greater than 1; otherwise β is negative.

Here are procedures for generating the Beatty sequences of the first (α) and second ($\alpha/(\alpha - 1)$) kind:

```
procedure beatty1seq(r)
  if r < 1.0 then fail
  suspend integer(seq() * r)
end
procedure beatty2seq(r)
  if r < 1.0 then fail
  suspend integer(seq() * (r / (r – 1)))
end
```

Figure 1 shows grid plots for some Beatty sequence pairs.

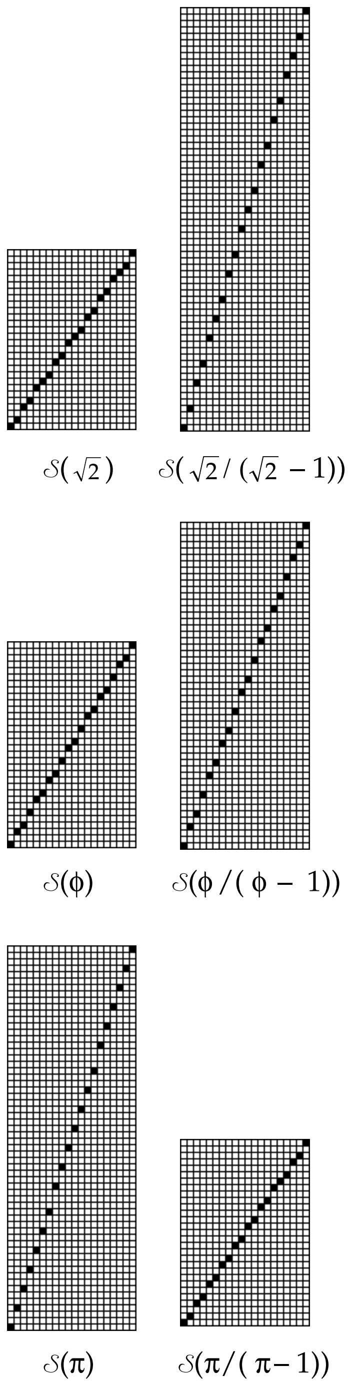


Figure 1. Beatty Sequences

Two observations about spectra sequences:

(1) Because $\sqrt{2}/(\sqrt{2}-1)$ simplifies to $2+\sqrt{2}$, the complementary sequence $\mathcal{S}(\sqrt{2})$ is just the same sequence, spread out over bigger gaps. You can see how the clusters of two and three follow the same pattern in both sequences.

(2) The complement of $\mathcal{S}(\phi)$ is $\mathcal{S}(\phi^2)$ after simplification, due to the special properties of ϕ.

Collating the pairs of Beatty sequences in Figure 1 gives the results shown in Figure 2.

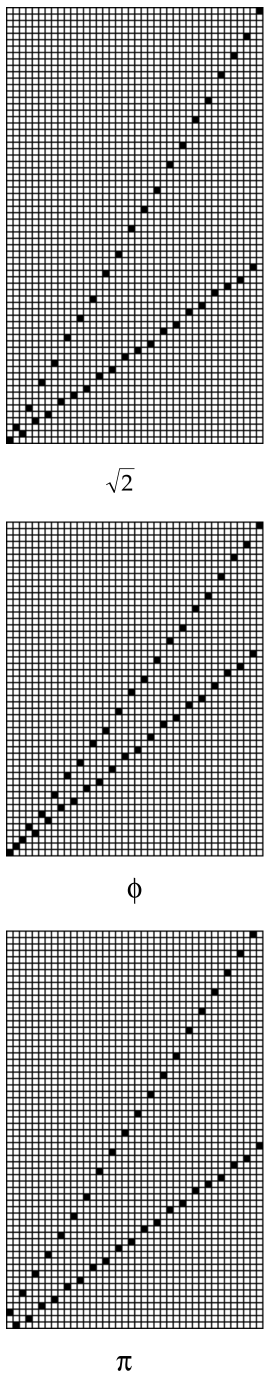


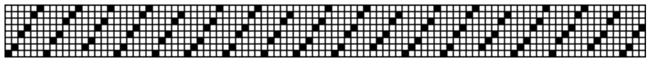
Figure 2. Collated Beatty Sequences

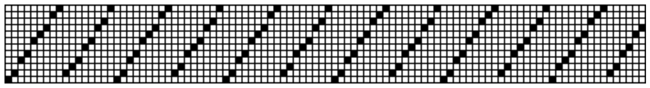
Spectra T-Sequences

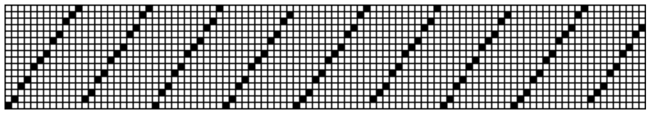
As we've mentioned several times in earlier articles, almost all integer sequences with any structure can be used as the basis for interesting weave patterns. Spectra sequences are no exception.

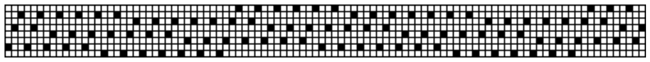
As usual, it's necessary to bring such sequences within the bounds of the number of shafts or treadles used. And, as usual, we'll do this by taking residues in shaft arithmetic [2].

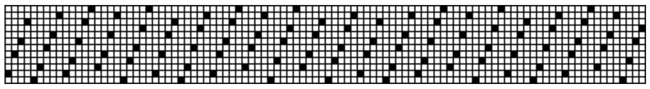
Figure 3 shows grid plots for some Beatty T-sequences and different numbers of shafts.

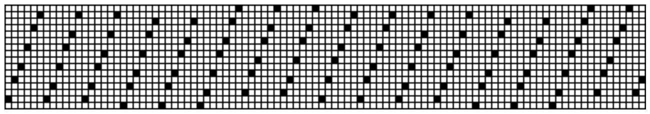

$\mathcal{S}(\sqrt{2})$, 8 shafts

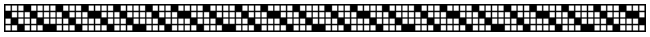

$\mathcal{S}(\sqrt{2})$, 12 shafts


$\mathcal{S}(\sqrt{2})$, 16 shafts


$\mathcal{S}(\phi/(\phi - 1))$, 8 shafts


$\mathcal{S}(\phi/(\phi - 1))$, 12 shafts


$\mathcal{S}(\phi/(\phi - 1))$, 16 shafts


$\mathcal{S}(\pi)$, 4 shafts

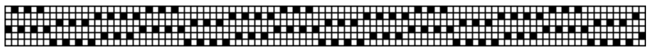
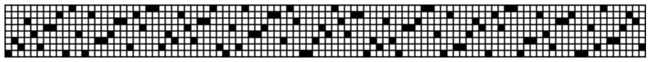
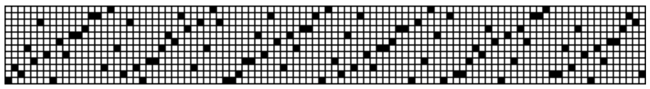
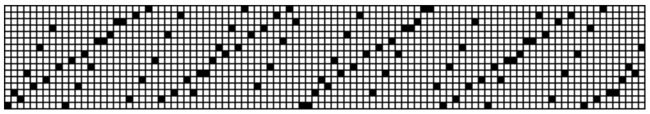

$\mathcal{S}(\pi)$, 6 shafts

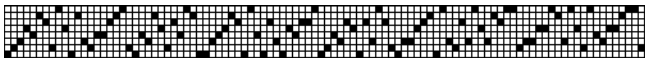
Figure 3. Beatty T-Sequences

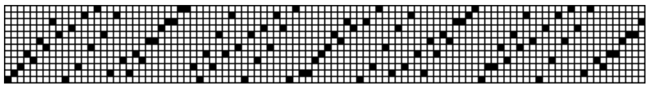
Figure 4 shows collated Beatty T-sequences corresponding to the sequences in Figure 3.

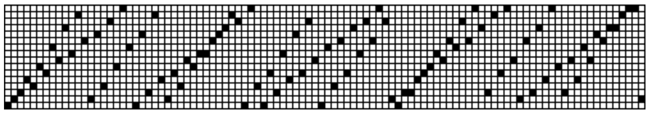

$\sqrt{2}$, 8 shafts

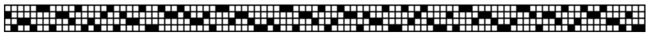

$\sqrt{2}$, 12 shafts


$\sqrt{2}$, 16 shafts


ϕ, 8 shafts


ϕ, 12 shafts


ϕ, 16 shafts


π, 4 shafts

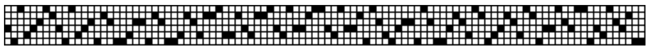

π, 6 shafts

Figure 4. Collated Beatty T-Sequences

Figures 5 through 7 show drawdowns for collated Beatty T-sequences with 2/2 twill tie-ups and 8 shafts and treadles, treadled as drawn in.

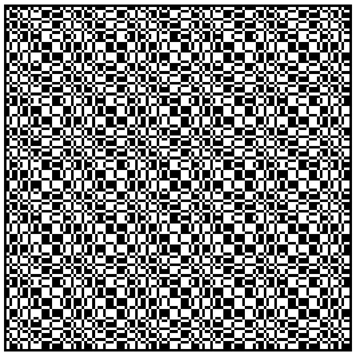


Figure 5. Beatty $\sqrt{2}$ Drawdown

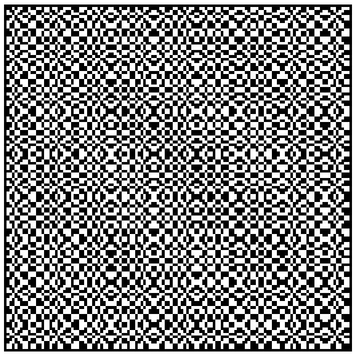


Figure 6. Beatty ϕ Drawdown

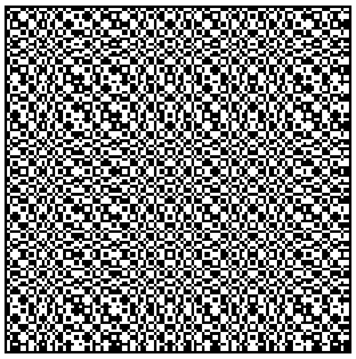


Figure 7. Beatty π Drawdown

References

1. *CRC Concise Encyclopedia of Mathematics*, Eric Weisstein, Chapman & Hall/CRC, 1999, p. 104.

2. “Shaft Arithmetic”, Icon Analyst 57, pp. 1-5.

Modular Expansion

In an earlier article, we described how modular reduction can be used to bring integer sequences within the range of a fixed number of shafts [1].

Modular reduction effectively wraps the sequence around a modular wheel whose modulus, *m*, is the number of shafts. Values not in the range $1 \le i \le m$ are replaced by their residues. See Figure 1.

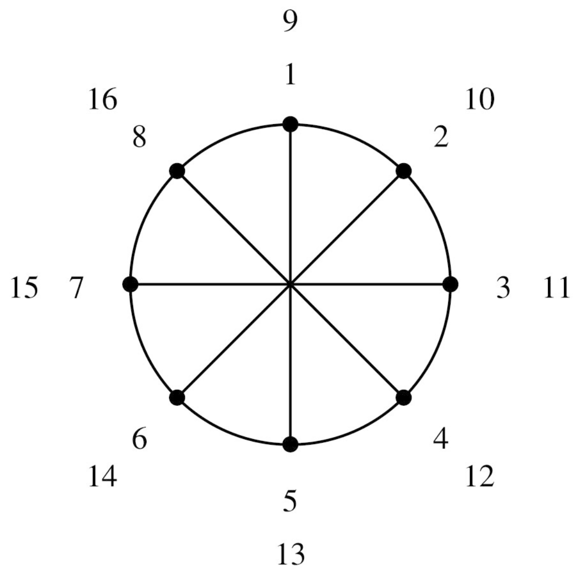


Figure 1. Arithmetic Shaft Modulo 8

The converse operation to modular reduction, which we call modular expansion, can be used to convert a T-sequence on *m* shafts to a T-sequence on *n* shafts, $n \ge m$, in which there is no wrap-around. The result is a sequence whose residues, shaft modulo *m*, produce the original sequence.

Figures 2 and 3 show an example of modular expansion.

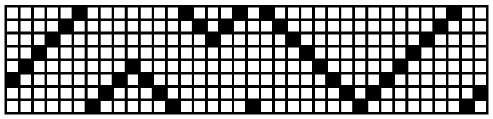


Figure 2. A T-Sequence with Wrap-Around

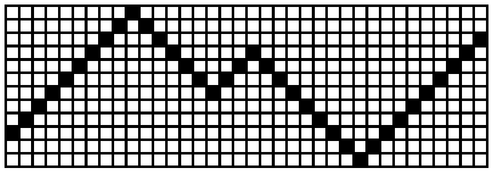


Figure 3. Wrap-Around Removed by Modular Expansion

The process of modular expansion is simple and relies on the fact that 1 and *m* are adjacent on the modular wheel.

Starting with $i = 1$, if term $t_i = m$ and $t_{i+1} = 1$, add *m* to t_{i+1} and all the remaining terms (shifting them upward by *m*). Similarly, if $t_i = 1$ and $t_{i-1} = m$, subtract *m* from t_{i-1} and all the remaining terms (shifting them downward by *m*). Note that adding or subtracting a multiple of *m* does not affect the residues.

When the process is done, add enough multiples of *m* to bring the smallest value in the range 1 to *m*. (The smallest value can be less than 1 but it cannot be greater than *m*, since t_1 is not greater than *m* and is not changed by the process.)

We’ll use the notation

$$\not\equiv S$$

to denote the modular expansion of *S*. Then

$$((\not\equiv S) \equiv m) = S$$

Of course, $\not\equiv S$ is not the only sequence whose residues shaft modulo *m* produces *S*.

Here’s a procedure that performs modular expansion:

```
procedure sunmod(x)
  local base, bound, i, lseq, k

  x := copy(spromote(x))

  base := 0

  bound := sbound ! x
```

```
   lseq := [get(x)] | return []
   while i := get(x) do {
      if (i = 1) & (lseq[–1] = base + bound) then
         base +:= bound
      else if (i = bound) & (lseq[–1] = base + 1) then
         base –:= bound
      put(lseq, base + i)
      }
   k := (smin ! lseq)
   if k > 0 then return lseq
   k := bound ∗ (–k / bound + 1)
   every !lseq +:= k
   return lseq
end
```

The motivation for modular expansion is to make runs evident (see Figures 2 and 3) and easy to deal with using ordinary arithmetic.

Comment: Although there are occasional references in the weaving literature to modular arithmetic, they usually are cast informally in terms of modular wheel diagrams. All the ones we've seen have been in reference to specific problems such as how to add incidentals to get alternating parity. Nonetheless, weavers who design original weaves must have at least an intuitive awareness of the logical adjacency between the top and bottom shafts.

Reference

1. "Shaft Arithmetic", Icon Analyst 57, pp. 1-5.

T-Sequence Analysis

In previous articles on T-sequences [1-5], we analyzed T-sequences manually, finding patterns and casting them in terms of operations on sequences that produce them.

As noted earlier [4], there may be many ways to create the same T-sequence. Any particular characterization is just one way. Without knowing how a T-sequence was constructed, analysis is at best an educated guess. Independent of how a T-sequence was constructed, some characterizations in terms of patterns and operations are better than others. Brevity and compactness (objective) and naturalness (subjective) are desirable.

Automatic (algorithmic) analysis of T-sequences that gives acceptable results is more difficult than it may seem and, in fact, is not possible for all T-sequences found in actual drafts.

T-sequences that are not good candidates for algorithmic analysis include ones:

- derived from mathematical sources, such as Fibonacci residues [6]
- that contain elements of randomness
- obtained by digitizing curves

In any event, the best way to do T-sequence analysis probably is to combine algorithmic methods with user interaction. In this article, we'll stick to algorithmic analysis.

Core Analysis Operations

In past articles we have presented many operations that can be used to construct and describe T-sequences. Some, such as repeats, are basic. Others, such as vertical reflection, occur rarely. Some operations, such as duplicate removal, leave no trace of their work and are not useful in analysis.

In the following sections, we'll describe the operations that are the most important to the analysis of T-sequences found in actual weaving drafts.

Repeats

Repeats are found in almost all T-sequences. If a repeat is found, it reduces the size of the T-sequence remaining to be analyzed, often by a substantial amount.

Collations

Collations result from the interleaving of two or more sequences. Since the component sequences may, and usually do, have different patterns, finding collations has high priority.

The result of finding a collation is three or more sequences, all of which are shorter than the original sequence. One of these is the index sequence, which usually is quite simple.

Collation analysis is important because of the frequency with which collations occur in T-sequences but it also is vexing.

The underlying problem is that any non-trivial T-sequence can be considered as a collation, often in many different ways. For example, $(1 \rightarrow 4)^5$ can be cast as the collation $\sim(1^5, 2^5, 3^5, 4^5)$. (We've dropped the overbar for repeats.)

General collation analysis is hopeless. We'll limit our attempts to simple collation [3] of sequences with repeats on disjoint shaft sets.

Such collations are periodic. Specifically, the collation of k sequences with periods $p_1, p_2, \ldots, p_k$ is periodic with period $m \times \mathrm{lcm}(p_1, p_2, \ldots, p_k)$. (The factor of m is a result of each component sequence being spread out m places.)

Concatenations

Concatenation is a consideration when a T-sequence consists of dissimilar segments. This may occur when borders bound a central pattern. See Figure 1.

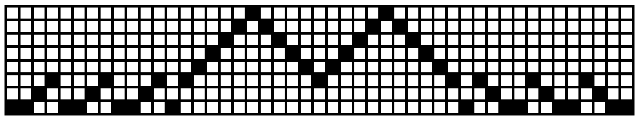


Figure 1. Concatenation of Dissimilar Segments

Dissimilar segments need to be analyzed separately and the results concatenated.

The problem is that "dissimilar" is not well defined. Still, a human being usually can easily find the boundaries between dissimilar segments of a sequence.

We've tried several ways of dealing with concatenation in an algorithmic way, including discrete Fourier transforms [7] and breaking up sequences into segments of runs and disconnected values. So far, no method has produced useful results and all methods have interfered with other kinds of analyses.

Runs

Simple, disconnected runs consist of successive values between starting and ending points. See Figure 2.

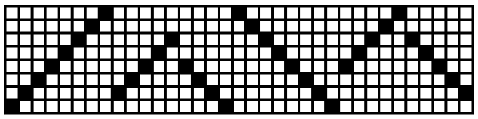


Figure 2. Simple Runs

A connected run consists of successive values between a beginning point, inflection points at which direction changes, and an ending point. See Figure 3. A connected run is, of course, a succession of simple runs, but its characterization is simpler.

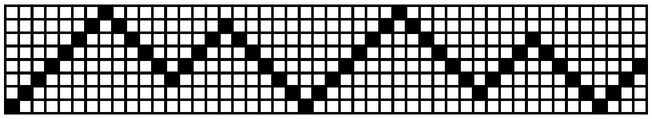


Figure 3. A Connected Runs

Both kinds of runs occur frequently in T-sequences.

Motifs Along Paths

Motifs along paths deserve consideration because of the frequency with which they occur in practice. Motifs along paths also subsume repeats: A repeat is just a motif along a constant path, $\overline{1}$, although we will not use this fact in analysis.

The Role of Modular Expansion

As described in the article **Modular Expansion** that begins on page 4 of this issue of the *Analyst*, modular expansion is effective in revealing runs that have been broken by modular reduction, as shown in Figures 2 and 3 in that article.

An extreme but common case is shown in Figures 4 and 5.

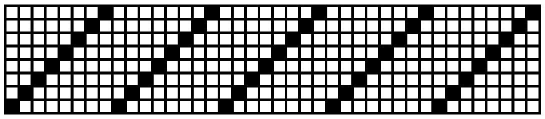


Figure 4. An Ascending Straight Draw

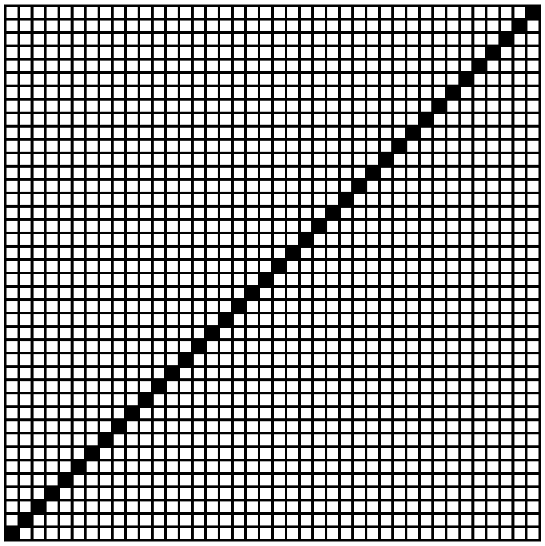


Figure 5. Modular Expansion of an Ascending Straight Draw

The ascending straight draw can be cast in terms of T-sequence operations as both $(1 \rightarrow 8)^5$ and $(1 \rightarrow 40) \equiv 8$.

Which view is more natural? A weaver probably would view the familiar repeat as the "right" characterization and the modular reduction of a very long draw as an aberration. On the other hand, a mathematically literate nonweaver probably would view the modularly-reduced long draw as better representing the structure of the sequence.

One thing this example makes clear is that modular expansion can alter the period of a repeat or even eliminate it.

For this reason, we've used modular expansion only within the procedures for run analysis.

The Question of Order

The most difficult (in fact, impossible) task in straightforward algorithmic analysis without pre-analysis or evaluation of intermediate results and backtracking is deciding on the order in which the various kinds of analyses should be performed.

It might seem that repeat analysis should be done first to extract a basic unit. If that is done and the resulting repeat is later analyzed for collations, there may not be enough information left to do collation analysis. Similarly, motif-along-a-path analysis will succeed and produce bizarre results for T-sequences that are most naturally viewed as connected runs. If the order is reversed, however, some T-sequences that are most naturally viewed as motifs along paths will instead be analyzed as long, unnatural runs.

Trying to circumvent these problems by making different kinds of analyses more "clever" leads to complexity and other kinds of problems. The difficulty is fundamental.

We therefore know, *a priori*, that simple algorithmic analysis can only have limited success. It is, however, a step toward more sophisticated kinds of analyses and needs to be done, if only to discover lurking problems.

Termination

At some point in analysis, there either is nothing left to try or the T-sequence is too short to benefit from analysis and the result may be more complicated than the sequence itself. For example, is $(1, 4)^2$ a better characterization than just 1, 4, 1, 4? Is $\rightarrow(1, 3, 1)$ better than 1, 2, 3, 2, 1?

As with so many other things in T-sequence analysis, there is no "right" answer. We have tried various values for preventing further attempted analysis. At present we do not attempt to analyze a sequence whose length is less than 5.

The Core Analysis Procedures

The procedures that follow succeed if the analysis is successful but fail otherwise. In all cases, to be successful, the analysis must apply to the entire sequence.

Some procedures described in earlier articles on T-sequences are used in what follows. In a few cases the procedures have been changed slightly. The full set is on the Web site for this issue of the *Analyst*.

The procedure that controls the analysis is

```
$define MinLength 5

procedure get_analysis(seq)

   if *seq < MinLength then return simageb(seq)

   return (
      get_scollate(seq) |
      get_srepeat(seq) |
      remod(seq, get_srun) |
      remod(seq, get_sruns) |
      get_splace(seq) |
      simageb(seq)
      )

end
```

Modular expansion for run analysis is handled by the procedure `remod(seq, p)`:

```
procedure remod(seq, p)
   local nseq, bound

   nseq := sunmod(seq)

   if (sbound ! nseq) > (bound := sbound ! seq) then
      return "smod(" || p(nseq) || ", " || bound || ")"
   else return p(copy(seq))

end
```

The simplest analysis procedure, although not the first one tried, is

```
procedure get_srepeat(seq)
   local i

   i := speriod(seq) | fail
   return "srepeat(" || get_analysis(seq[1+:i]) ||
      ", " || (*seq / i) || ")"

end
```

A separate procedure is used to get the period because it is needed in other analysis procedures.

We'll make an important simplifying assumption that the repeat comes out "even" — that the T-sequence does not end in a partial repeat. This allows us to examine only those segments whose lengths are divisors of the length of the entire sequence.

```
procedure speriod(seq)
   local i, segment
```

```
    every i := 1 | divisors(*seq) do {
      segment := seq[1+:i]
      if sequiv(sextend(segment, *seq), seq) then
        return i
      }

    fail

  end
```

The procedure divisors(i), which is from the Icon program library module factors, generates the proper divisors of i in increasing order. The improper divisor 1 is added, since in some situations, the sequence is constant. Note that the smallest repeat is found.

Motif-along-a-path analysis is similar to repeat analysis:

```
  procedure get_splace(seq)
    local i, j, motif, seq2, path

    every i := divisors(*seq) do {
      motif := seq[1+:i]
      every j := i + 1 to *seq by i do
        if not sequiv(motif, sground(seq[j+:i], seq[1]))
          then break next
      path := []
      every put(path, seq[1 to *seq by i])
      return "splace(" || get_analysis(motif) ||
        ", " || get_analysis(path) || ")"
      }

    fail

  end
```

Motifs of length 1 are not considered, since all T-sequences are unit sequences along a path.

The procedure sground() brings the next potential motif segment to the specified level, which in the procedure call above is the first value in the current motif candidate:

```
  procedure sground(seq, i)
    local j

    j := smin ! seq

    every !seq –:= (j – i)

    return seq

  end
```

The two procedures for run analysis are similar:

```
  procedure get_srun(seq)        # connected runs
    local i, j, new_seq, dir

    seq := copy(seq)

    i := get(seq)
    j := get(seq)

    if j = i – 1 then dir := –1        # down going
    else if j = i + 1 then dir := 1    # up going
    else fail

    new_seq := [i]

    while i := get(seq) do {
      if i = j + 1 then {
        if dir = –1 then put(new_seq, j)
        dir := 1
        }
      else if i =  j – 1 then {
        if dir = 1 then put(new_seq, j)
        dir := –1
        }
      else {
        put(new_seq, j)
        push(seq, i)                   # put back
        break
        }
      j := i
      }

    if *seq ~= 0 then fail             # remaining terms?

    put(new_seq, j)

    return "srun(" || get_analysis(new_seq) || ")"

  end

  procedure get_sruns(seq)       # disconnected runs
    local i, j, seq1, seq2, dir

    seq1 := []
    seq2 := []

    repeat {
      i := get(seq) | {
        put(seq2, j)
        break                          # end of road
        }
      j := get(seq) | fail             # isolated end point
      if j = i – 1 then dir := –1      # down going
      else if j = i + 1 then dir := 1# up going
      else fail
      put(seq1, i)                     # beginning point
      while i := get(seq) do {
        if i = j + dir then {
          j := i
          next
          }
        else {
            push(seq, i)                # put back
            put(seq2, j)
            break
            }
         }
      }

   return "sruns(" || get_analysis(seq1) || ", " ||
      get_analysis(seq2) || ")"

end
```

Collation analysis is the most complicated of all analyses. It involves finding the periods of individual shafts and collecting together those with the same periods:

```
procedure get_scollate(seq)
   local bound, deltas, i, j, poses, positions, oper
   local results, result, k, count, oseq, m, nonperiod
   local seqs, facts, period

   speriod(seq) | fail     # only do periodic case

   bound := (sbound ! seq)

   deltas := table()
   positions := table()

   every  i := 1 to bound do {
      poses := spositions(seq, i)
      positions[i] := poses
      j := sconstant(sdelta(poses))
      /deltas[j] := []
      put(deltas[j], i)
      }

   if *deltas < 2 then fail

   oseq := list(*deltas)            # decollation order

   count := 0

   every k := key(deltas) do {
      count +:= 1
      every j := !deltas[k] do
         every m := !positions[j] do
            oseq[m] := count
      }

   seqs := sdecollate(oseq, seq) | fail

   oper := "scollate(" ||
      (simageb(oseq[1+:speriod(oseq)]) |
         get_analysis(oseq))

   every oper ||:= ", " || get_analysis(!seqs)

   return oper || ")"

end
```

The procedure spositions() produces a list of positions at which shafts appear, from which the shaft periods can be determined:

```
procedure spositions(seq, i)
   local lseq, count

   seq := copy(seq)

   lseq := []

   count := 0

   while i := get(seq) do {
      count +:= 1
      if member(seq, i) then
         put(lseq, count)
      }

   return lseq

end
```

The procedure sdecollate() does the actual decollation:

```
procedure sdecollate(order, seq)
   local lseq, i, j

   order := copy(order)

   lseq := list(sbound ! order)    # sequences to return

   every !lseq := []               # all initially empty

   every j := !seq do {
      i := get(order)
      put(order, i)
      put(lseq[i], j)
      }

   return lseq

end
```

If all else fails, the analysis produces the concatenation of the remaining values:

```
procedure concat_image(seq)

   if *seq = 1 then return seq[1]

   return "sconcat(" || simage(seq) || ")"

end
```

Evaluation

Analyzing T-sequences using the core procedures described above produces correct results for all the T-sequences we have tried — correct in the sense that the expressions resulting from analysis produce the original sequences.

The quality of analysis is another matter. In the case of the concatenation of dissimilar sequences, the analysis often produces just the concatenation of the individual values, although in some cases it may produce runs.

In some cases, the results of analysis are not as expected. On occasion, an unexpected result, upon examination, may prove to be better than the expression used to produce the sequence.

And, gratifyingly, the results of analysis sometimes are identical to the expressions that produce the sequences. Figure 6 shows examples for which the analysis is "perfect" in this sense.

The analysis described here is inefficient in the sense that some some analyses, notably repeat analysis, are attempted when it could be determined in advance that they will fail. The speed of analysis is not a problem, however, and efficiency is low on our list of concerns.

The most valuable result of developing even this flawed algorithmic analysis has been what we've learned about relationships between different kinds of patterns.

Other Analysis Operations

There are many other possible analysis operations, including palindromes, scaling, term duplication, and so forth.

Most palindromes consist of runs, and analysis of the sequence of inflection points in the runs produces the essence of the palindromes. Therefore, palindrome analysis probably is better performed after the core analyses.

Similarly, term duplication and the need for scaling usually arise from more basic kinds of analysis.

These are topics for further investigation.

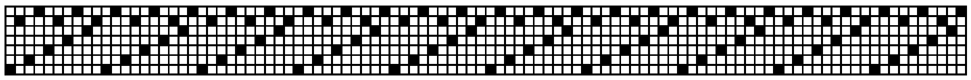
References

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

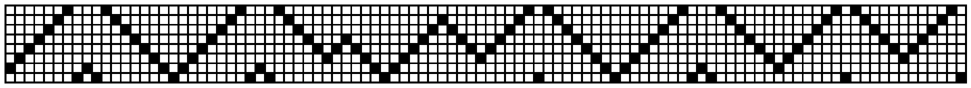
2. "Understanding T-Sequences II", Icon Analyst 64, pp. 6-12.

3. "T-Sequence Collation", Icon Analyst 65, pp. 1-2.

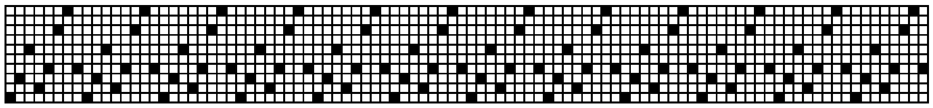
4. "Constructing T-Sequences", Icon Analyst 65, pp. 3-9.



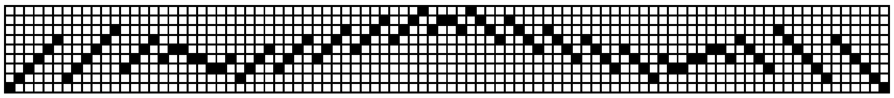
$\sim((1 \rightarrow 5)^{10}, (6, 7)^{25})$



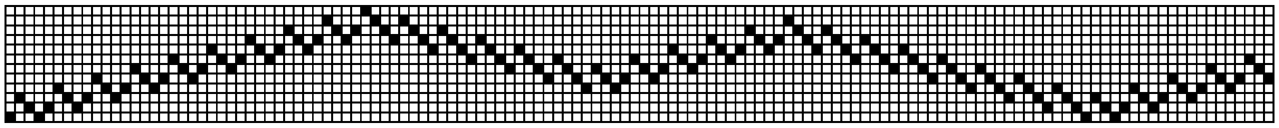
$(\rightarrow(2, 10, 1, 10, 3, 5, 1, 7, 3, 9, 1, 10, 2, 9, 3, 9)) \equiv 8$



$(1, 3, 6, 2, 4, 8, 10, 4)^{12}$



$(\rightarrow(1, 5, 2, 9, 3, 7, 6) \rightarrow (\rightarrow(6, 7, 3, 9, 2, 5, 1))$



(1, 3, 2, 1) @ $(\rightarrow$ (1, 10, 4, 9, 1, 5)

Figure 6. Sequences "Perfectly" Analyzed

5. "Generalizing T-Sequence Operands", The Iron Analyst 65, pp. 7-9.

6. "Residue Sequences", The Iron Analyst 58, pp. 4-6.

7. *Numerical Recipes: The Art of Scientific Computation*, William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, Cambridge University Press, 1986, pp. 387-390.

Tricky Business

When we first started to explore weaving, we came across a monograph entitled *Algebraic Expressions in Handwoven Textiles* [1].

The author was a weaver who had been a high-school mathematics teacher. In looking for inspirations for designing weaves, she turned to mathematics. What she did seems to us to be naive and quirky, but it was sufficiently successful to gain attention and acclaim [2-3].

Although she described her mathematical basis as algebraic expressions, her method was more limited than that — powers of linear multivariate polynomials, such as

$$(a + b + c + d)^3$$

which, when expanded, becomes

$$a^3 + 3a^2b + 3ab^2 + b^3 + 3a^2c + 6abc + 3b^2c + 3ac^2 + 3bc^2 + c^3 + 3a^2d + 6abd + 3b^2d + 6acd + 6bcd + 3c^2d + 3ad^2 + 3bd^2 + 3cd^2 + d^3$$

Each variable corresponds to a shaft (or treadle), so the expression above is for four shafts. The power used represents the "degree of interaction among the variables", higher powers leading to more complex and longer sequences.

The expanded polynomials are interpreted as sequences in the following way. Powers are written out as products. For example, a^2b becomes *aab*, which in turn corresponds to the sequence 1, 1, 2.

The numerical coefficient of a term causes the term to be replicated accordingly. For example, $3a^2b$ becomes *aabaabaab* and the sequence is 1, 1, 2, 1, 1, 2, 1, 1, 2.

Finally the results for successive terms are concatenated.

The result for the polynomial above is

aaaaabaabaababbabbabbbbbaacaacaacabcabcab
cabcabcabcbbcbbcbbcaccaccaccbccbccbcccccaa
daadaadabdabdabdabdabdabdbbdbbdbbdacdac
dacdacdacdacdbcdbcdbcdbcdbcdbcdccdccdccd
addaddaddbddbddbddcddcddcddddd

The sequence produced depends, of course, on the ordering of the variables. The one given above is that standard lexicographic ordering for multivariate polynomials.

The corresponding sequence is

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

This sequence can be represented more compactly using T-sequence operations:

$(1)^5, (2, 1, 1)^2, (2, 1, 2)^3, (2)^4, (1, 1, 3)^3, (1, 2, 3)^5,$
$(2, 2, 3)^3, (1, 3, 3)^3, (2, 3, 3)^3, (3)^3, (1, 1, 4)^3,$
$(1, 2, 4)^6, (2, 2, 3)^3, (1, 3, 4)^3, (2, 3, 4)^6, (3, 3, 4)^3,$
$(1, 4, 4)^3, (2, 4, 4)^3, (3, 3, 4)^3, (4)^5$

We have omitted the overbar for grouped terms, something we wish we'd done earlier.

A grid plot for this sequence is shown in Figure 1.

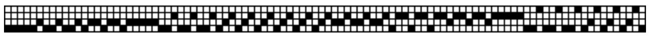


Figure 1. Sequence for $(a + b + c + d)^3$

Figure 2 shows a drawdown for this sequence, treadled as drawn in and with a direct tie-up.

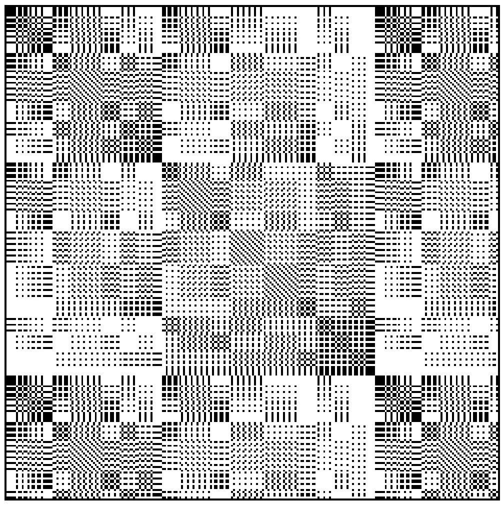


Figure 2. ($a + b + c + d$)^3 Drawdown

T-Sequences of this kind, with many successive duplicates, are better used for profile drafting [4] than for thread-by-thread drafting.

In practice, sequences produced from powers of multivariate polynomials are rearranged in various ways to produce more attractive results. The process, therefore, is not entirely algorithmic.

All kinds of extensions and variations on multivariate polynomials are possible, but as far as we know, these have not been pursued.

We are not interested here in designing weaves by the method described above. We are, however, interested in how such T-sequences can be obtained from multivariate polynomials.

The problem is the usual one: Interesting expressions are too complicated to work out by hand, not to mention the likelihood of errors.

As usual, we use *Mathematica*, which is designed for this kind of thing, but then we are left with processing the *Mathematica* output. Here's what that output looks like for the polynomial we've been using for an example:

```
\!\(a\^3 + 3\ a\^2\ b + 3\ a\ b\^2 + b\^3 + 3\ a\^2\ c +
  6\ a\ b\ c + 3\ b\^2\ c + 3\ a\ c\^2 + 3\ b\ c\^2 +
  c\^3 + 3\ a\^2\ d + 6\ a\ b\ d + 3\ b\^2\ d +
  6\ a\ c\ d + 6\ b\ c\ d + 3\ c\^2\ d + 3\ a\ d\^2 +
   3\ b\ d\^2 + 3\ c\ d\^2 + d\^3\)
```

It's not that hard to write an *ad hoc* program to convert such input to the corresponding T-sequence:

```
link strings

procedure main()
   local expr, result, term, i, var, letters, sequence, k

   expr := ""

   # Read Mathematica expression.

   while expr ||:= read()

   #  Delete extraneous characters.

   expr := deletec(expr, '()\\! ')

   #  Analyze polynomial

   sequence := ""

   #  Parse expression.

   expr ? {
      while term := tab(upto('+–') | 0) do    # get term
         term ? {
            result := ""
            i := (tab(many(&digits)) | 1)        # multiplier
            while var := move(1) do {           # variable
               if ="^" then k := tab(many(&digits))
                  else k := 1
               result ||:= repl(var, k)
               }
            sequence ||:= repl(result, i)
            }
         move(1) | break
         }
      }

   letters := &lcase ** sequence       #variables

   #  Map variables into digits (assumes at most
   #   nine variables).

   write(map(sequence, letters,
      "123456789"[1 +: *letters]))

end
```

After writing this program, we thought there must be a better (harder), more elegant (strange), and interesting (obscure) way to solve the problem.

We were reminded of the way we dealt with *Mathematica* expressions in solving continued-fractions with variable coefficients — converting such expressions to executable Icon code that produced the desired results [5].

To do this here, we need to approach the problem in a more sophisticated (peculiar) way, since the result needs to be a string involving the variables in the expression, while the expression, as it stands, is designed to produce a numerical result for the given values of the variables.

The technique we used was to associate a unique prime number with each variable and then rely on the fundamental theorem of arithmetic to recover the variables from a numerical result.

The fundamental theorem of arithmetic states that, up to the order of terms, any positive integer can be represented uniquely as a product of powers of primes:

$$i = p_1^{c_1} \times p_2^{c_2} \times \ldots \times p_n^{c_n}$$

where p_i is the ith prime. For example, if

a = 2
b = 3
c = 5
d = 7

then a^2b = 12 and bd^2 = 147. These terms can be recovered by factoring 12 and 147, respectively.

There is one other thing to be considered: the numerical coefficients of terms. This can be handled by choosing primes for variables that are larger than the largest numerical coefficient.

As usual, it's easier to understand what's going on when one program writes another by looking at the written program first (we've done minor editing to get line lengths within printing boundaries):

```
procedure main()
   a := 7
   b := 11
   c := 13
   d := 17
   terms := (a ^ 3 || "," || 3 * a ^ 2 * b || "," || 3 * a *
   b ^ 2 || "," || b ^ 3 || "," || 3 * a ^ 2 * c || "," || 6 * a *
   b * c || "," || 3 * b ^ 2 * c || "," || 3 * a * c ^ 2 || "," ||
   3 * b * c ^ 2 || "," || c ^ 3 || "," || 3 * a ^ 2 * d || "," ||
   6 * a * b * d || "," || 3 * b ^ 2 * d || "," || 6 * a * c *
   d || "," || 6 * b * c * d || "," || 3 * c ^ 2 * d || "," || 3 *
   a * d ^ 2 || "," || 3 * b * d ^ 2 || "," || 3 * c * d ^ 2 ||
   "," || d ^ 3)
   result := ""
   terms ? {
      while term := tab(upto(',') | 0) do {
         pattern := ""
         every var := !'abcd' do {
            while term % variable(var) = 0 do {
               pattern ||:= var
               term /:= variable(var)
               }
            }
         result ||:= repl(pattern, term)
         move(1) | break
         }
      }
   letters := &lcase ** result
   every write(!map(result, letters,
      "123456789"[1 +: *letters]))
end
```

The primes used for variables start at 7 because the largest numerical coefficient is 6. The terms are concatenated so that the result is the complete string. After factoring and recovering the variables (note the use of variable() to convert a string to the corresponding variable), the variables are mapped into shaft numbers and the sequence is written out one term per line.

Here's the program that reads *Mathematica* input and produces programs such as the one above:

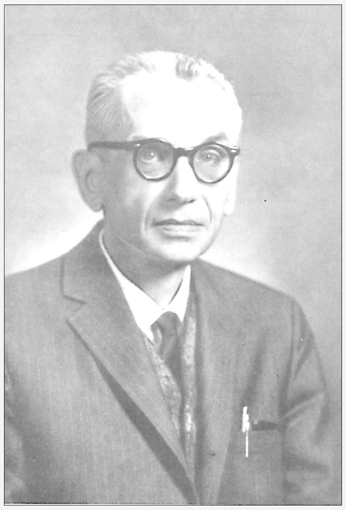
```
link options
link factors
link strings

procedure main(args)
   local exp, line, vars, limit, c, opts, name, output
   local expr1, expr2, file, var, max, primes

   opts := options(args, "l+")

   limit := \opts["l"] | 100

   output := open("dietzsol.icn", "w") |
      stop("*** cannot open file for program")

   exp := ""

   #  Input may be on more than one line.

   while exp ||:= pretrim(read())

  #  Variables are guaranteed to be lowercase letters

   vars := cset(exp) ** &letters

   #  Find the largest number in the expression.

   max := 0

   exp ? {
      while tab(upto(&digits)) do
         max <:= tab(many(&digits)) \ 1
      }

   #  Get as many primes past the largest number
   #  as there are variables.

   primes := [nxtprime(max)]                # big enough

   every 2 to *vars do
      put(primes, nxtprime(primes[–1]))

   #  Perform ad-hoc replacements to convert
   #  Mathematica syntax to a valid Icon expression.

   exp := replacem(exp,
      "\\ ", " * ",
      "\\^", " ^ ",
      "\\(", "(",
      "\\)", ")",
      "\\!", "",
      "+", "|| \",\" ||",        # concatenation and separator
      )

   write(output, "procedure main()")
   every var := !vars do
      write(output, "   ", var, " := ", get(primes))
   write(output, "   terms := ", exp)
   write(output, "   result := \"\"")
   write(output, "   terms ? {")
   write(output, "     while term := tab(upto(',') | 0) do {")
   write(output, "        pattern := \"\"")
   write(output, "        every var := !", image(vars),
      " do {")
   write(output, "           while term % variable(var)_
      = 0 do {")
   write(output, "              pattern ||:= var")
   write(output, "              term /:= variable(var)")
   write(output, "              }")
   write(output, "           }")
   write(output, "        result ||:= repl(pattern, term)")
   write(output, "        move(1) | break")
   write(output, "        }")
   write(output, "     }")
   write(output, "   write(result)")
   write(output, "end")

   close(output)

   system("icont –s dietzsol –x")
   write(output, "   every write(!map(result, letters,_
      \"123456789\"[1 +: *letters]))")

   remove("diosol.icn")            # clean up debris
   remove("diosol")

end
```

The procedure nxtprime(i), which produces the next prime after i, is from the Icon program library module factors.

Now isn't that a better (harder), more elegant (strange), and interesting (obscure) way to solve the problem than the simple *ad hoc* method?

Sarcasm aside, the use of the fundamental theorem of arithmetic to encode nonnumeric values deserves remembering. Kurt Gödel used a more sophisticated version of this idea to prove one of the most important mathematical results of all time. See the side bar.

Gödel and Undecidability


Kurt Gödel 1906 - 1978

Kurt Gödel shook the foundations of mathematics by proving that certain mathematical propositions cannot be proven to be true or false.

Put in an informal way, Gödel's incompletness theorem states that all consistent axiomatic formulations of arithmetic include undecidable propositions [1].

The method Gödel used relied on a system by which every mathematical proposition was represented by a distinct number. Such numbers are called Gödel numbers. For example, the proposition $(\exists x)(x = sy)$, which means "there exists an x such that x is the immediate successor of y" is encoded as

$$2^8\ 3^4\ 5^{13}\ 7^9\ 11^8\ 13^{13}\ 17^5\ 19^7\ 23^{16}\ 29^9$$

where the exponents correspond to the symbols in the proposition [1].

Needless to say, this is a very large number:

74880654697373651627226805069425599081
28930612274430799531084603348039652271
735661562500000000

For more information about Gödel's work and its ramifications, see Reference 2.

References

1. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1999, pp. 741-742.

2. *Gödel, Escher, Bach: an Eternal Golden Braid*, Douglas R. Hofstadter, Basic Books, 1979.

All that having been said, a saner way of approaching the problem would be to calculate the terms of multivariate polynomials using the binomial theorem.

References

1. *Algebraic Expressions in Handwoven Textiles*, Ada K. Dietz, The Little Loomhouse, Louisville, Kentucky, 1949.

2. "Two Weavers in a Trailer", *Handweaver & Craftsman*, Spring 1953, pp. 20-21, 56, 60.

3. "Algebraic Expressions: Designs for Weaving", Lana Schneider, *Handwoven*, January/February 1998, pp. 48-49.

4. "Profile Drafting", Icon Analyst 65, pp. 18-20.

5. "Solving Square-Root Palindromes II", Icon Analyst 65, pp. 9-14.

Color-and-Weave

The visual pattern of a woven fabric comes from two primary sources. One is the interlacement pattern, which is what we've shown in drawdowns. In drawdowns, the interlacement pattern is made clear by using black warp threads and white weft threads. The interlacement is call *structure*.

In a woven fabric, the structure is apparent (if not always easily seen) even if the warp and weft are the same color. This is because threads have thickness and surface properties and their interlacement produces a three-dimensional effect.

The other source of patterns in woven fabrics comes from using threads of different colors.

The effect of the combination of structure and thread colors is a complex and fascinating subject that appears not to have been studied in any systematic way.

When we first encountered the term *color-and-weave* , we thought we'd at last found a source of information about how to use color in weave design. In fact, color-and-weave has nothing to do with hue. It is concerned only with the patterns produced by threads of two contrasting colors, generally referred to as dark and light — D and L. Despite this limitation, color-and-weave is the basis for the design of most stripes, checks, and similar simple patterns in woven fabric.

For our purposes, the contrasting colors can be represented by binary digits — 0 for dark and 1 for light, which work naturally with Icon's `g2` palette.

The simplest color-and-weave effects use alternating dark and light threads for the warp and the complement for the weft: DLDLDL … warp and LDLDLD … weft or, equivalently, 010101 … and 101010 … . There are many other possibilities that produce interesting patterns. We'll call the color sequences for threading and treadling *C-sequences*. In the present context, C-sequences are limited to two contrasting colors, but the concept has more general application [7].

The tie-up and T-sequences, which define the structure of the weave, play a major role in the resulting pattern. Plain weave, with uniform over-and-under interlacing, is the simplest. Figure 1 shows the pattern, which is just vertical stripes, for the basic color-and-weave C-sequences.

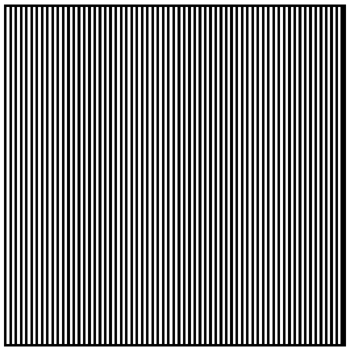


Figure 1. Color-and-Weave Stripes

Other possibilities fill books [1-6]. In this article, we'll show some possibilities that you won't find in the weaving literature. All the patterns that follow use weft C-sequences that are the complements of the corresponding warp C-sequences.

Almost all published C-sequences are periodic. Aperiodic sequences offer interesting alternatives.

Randomness usually produces unattractive patterns that just look noisy. However, with the constraints of complementary C-sequences for the warp and weft, there is enough correlation to produce interesting patterns. See Figure 2.

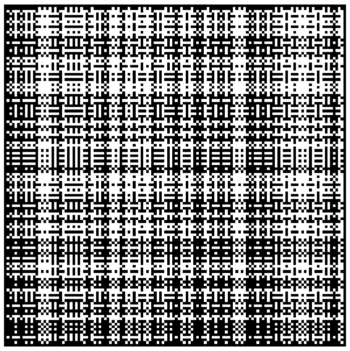


Figure 2. Random Color-and-Weave

Aperiodic fractal sequences that have strong structural properties, like the Morse-Thue sequence [8], produce interesting effects because they appear periodic at a glance, but actually are not. See Figure 3.

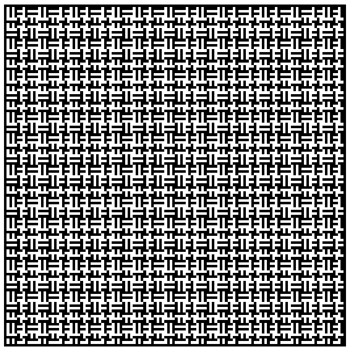


Figure 3. Morse-Thue Color-and-Weave

A sequence that has properties that are in many ways similar to the Morse-Thue sequence is the *rabbit sequence* [9], which is defined by the following L-system:

```
axiom:0
0–>1
1–>10
```

Here's a procedure that generates the rabbit sequence.

```
procedure rabbitseq()
  local memory, i

  memory := [0]

  repeat {
    i := get(memory)
    if i = 0 then put(memory, 1)
    else put(memory, 1, 0)
    suspend memory[1]
    }

end
```

Figure 4 shows a rabbit-sequence color-and-weave pattern.

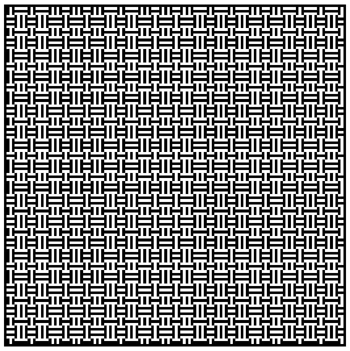


Figure 4. Rabbit Sequence Color-and-Weave

"Strongly" aperiodic sequences that plainly have no periodic properties can be interesting also. An example is the "multi" sequence, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, … . Here is a procedure that generates the "multi" sequence:

```
procedure multiseq()

  suspend (i := seq(), (|i \ i))

end
```

Figure 5 shows the color-and-weave pattern for the mod-2 residue sequence of the "multi" sequence.

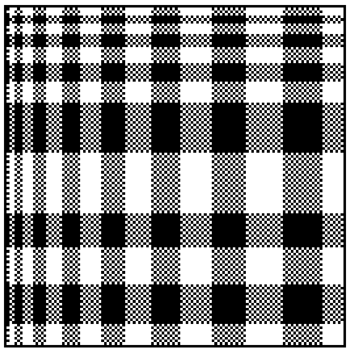


Figure 5. "Multi" Sequence Color-and-Weave

Of course, any mod-2 residue sequence is a candidate for a C-sequence. As we've shown earlier [10], many such sequences are periodic. For example, the repeat for the Fibonacci sequence mod-2 is 1, 1, 0. However, the mod-2 residues of the mod-5 residues of the Fibonacci sequence have period 20, the same as the mod-5 residues. Figure 6 shows the color-and-weave pattern for this sequence.

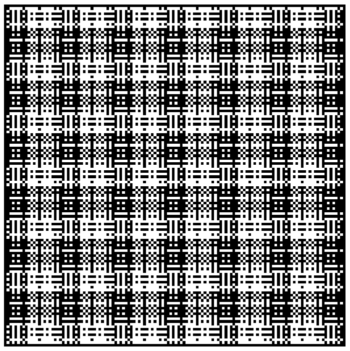


Figure 6. (fibseq() % 5) % 2 Color-and-Weave

Other possibilities for aperiodic sequences are signature sequences [11] and spectra sequences (see the article **Spectra Sequences**, which begins on page 1 of this issue of the Analyst).

Other Possibilities

The patterns produced by color-and-weave C-sequences depend strongly the interlacement structure used. Figures 1 through 6 all are for tabby tie-ups and ascending straight draw T-sequences.

The difference the structure makes is illustrated by Figure 7, which uses "multi" sequence C-sequences and a 2/2 twill tie-up with point draw T-sequences.

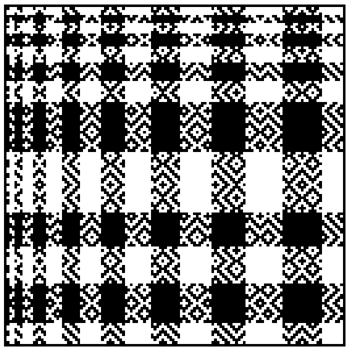


Figure 7. Twill "Multi" C-Sequence Color-and-Weave

It's also not necessary for the weft C-sequence to be the complement of the warp C-sequence. Figure 8 shows the twill structure used above with the two C-sequences the same.

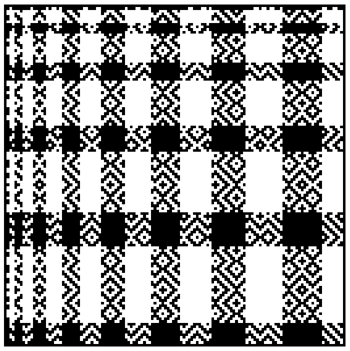


Figure 8. A Variant Twill Color-and-Weave

Notice that the patterns in Figures 7 and 8 are similar but not the same.

Imagine the other patterns that might be obtained from different interlacement structures.

The literature and lore of weaving is notable for its lack of unifying principles and generality. That leaves many opportunities for the analytically minded person. For example, as far as we can tell, no one had considered the obvious extension of color-and-weave to more than two different shades. Why not three — dark, medium, and light? Or more?

It does not necessarily follow that such generalizations will lead to interesting results, but they are worth exploring.

References

1. *Color and Weave Effects for Four-Harness Twills*, Margaret Ball, self-published, 1976.

2. *Color-and-Weave*, Margaret B. Windeknecht and Thomas G. Windeknecht, Van Nostrand Reinhold, 1981.

3. *Color-and Weave Design: A Practical Reference Book*, Ann Sutton, Bellew Publishing, 1984.

4. *Point Twill with Color-and-Weave*, Margaret B. Windeknecht, self-published, 1989.

5. *The Pinwheel: An Exploration in Color-and-Weave*

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

Design, Margaret B. Windeknecht, self-published, 1992.

6. *Color-and-Weave II*, Margaret B. Windeknecht, self-published, 1994.

7. "Weave Drafts", Icon Analyst 53, pp. 1-4.

8. "The Morse-Thue Sequence", Icon Analyst 65, pp. 15-18.

9. *Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise*, Manfred Schroeder, W. H. Freeman, 1991, p. 55.

10. "Residue Sequences", Icon Analyst 58, pp. 4-6.

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

Befriending a Sequence

A *friendly sequence* is one in which successive terms differ by one. Close proximity amounts to friendship. Figure 1 shows a friendly sequence, which we'll label ☺.

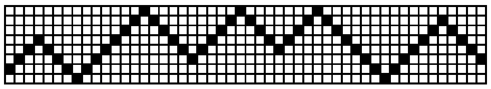


Figure 1. ☺ : A Friendly Sequence

Figure 2 shows a fairly unfriendly sequence, which we'll label 😐 and Figure 2 shows a downright hostile sequence,☹.

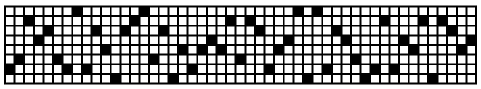


Figure 2. 😐 : A Fairly Unfriendly Sequence

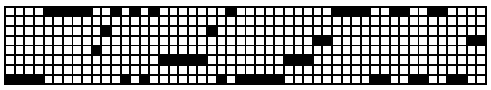


Figure 3. ☹ : A Hostile Sequence

☺ exudes good vibes; it's a cheerful sequence. The tension in 😐 is evident, while ☹ reeks of discord.

Our goal here is to convert unfriendly sequences to friendly ones — to befriend unfriendly sequences.

These are the rules:

- Only friendly terms may be added.
- Terms may not be deleted.
- Existing friends may not be separated.

Under these rules, befriending a friendly sequence does not change it.

The most straightforward, conservative approach is to add the fewest terms necessary to achieve a friendly result. This involves inserting a friend between pairs of equal, self-focussed terms and adding a run of friendly terms between unfriendly terms that are some distance apart.

Here are procedures that do this:

```
procedure befriend_con(s)
   local lseq, i, tail

   s := copy(s)

   lseq := [get(s)] | return [ ]

   while i := get(s) do
      lseq |||:= connect(lseq[–1], i)

   return lseq

end

procedure connect(j, i)
   local k, result

   result := []

   k := i – j

   if abs(k) = 1 then put(result, i)
   else if k = 0 then
      put(result, i + ?[1, –1], i)
```

Supplementary Material

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

```
      else if k > 0 then
         every put(result, j + 1 to i)
      else
         every put(result, j – 1 to i by –1)

      return result

   end
```

Note that there is only one place where a choice is made — whether to insert a friend above or below a pair of self-focussed terms.

A more enthusiastic approach is to allow some leeway in inserting friends between unfriendly terms — letting the friendly path wander a little. Of course we expect friend-binding paths to be finite so that befriending terminates. Here are procedures for enthusiastic befriending:

```
   procedure befriend_ent(s)
      local lseq, i, tail

      s := copy(s)

      lseq := [get(s)] | return []

      while i := get(s) do
         lseq |||:= wander(lseq[–1], i)

      return lseq

   end

   procedure wander(j, i)
      local result, k, incr

      result := [j]

      repeat {
         k := i – result[–1]
         if abs(k) = 1 then {
            put(result, i)
            break
            }
         incr := [1, –1]
         if k < 0 then
            every 1 to –k do
               put(incr, –1)
         else
            every put(incr, 1)
         put(result, result[–1] + ?incr)
         if result[–1] == i then break
         }

      if *result > 1 then get(result)

      return result

   end
```

Note that the choice of direction is biased toward the target friend. This does not guarantee that the process will terminate, but the probability that it will is high.

Figure 4 shows the results of befriending 😐 and ☹ in a conservative way. Figure 5 shows the results for the more enthusiastic befriending.

The horizontal tick marks at the left edges of these plots show the upper and lower bounds for the original sequences. Befriending in the way we've done it can add values larger or smaller than those in the original sequences.

Befriending T-Sequences

Of course, we want to consider befriending T-sequences. Friendly sequences make good connected runs [1], and since friendly sequences have alternating parity, they can be used for weaves that require this [2].

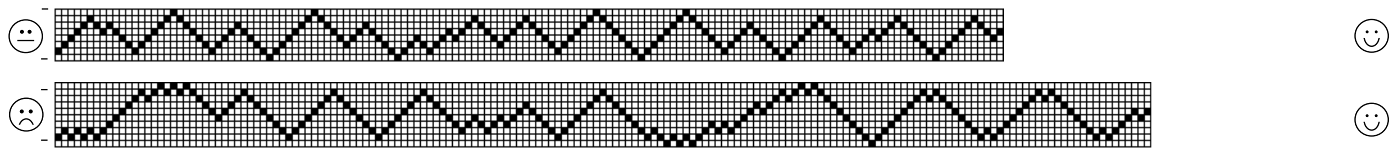


Figure 4. Conservatively Befriended Sequences

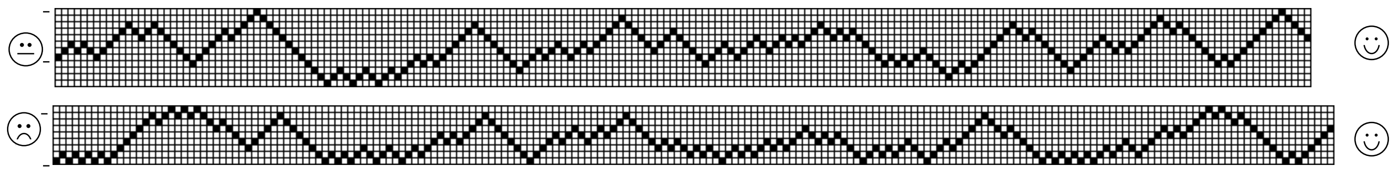


Figure 5. Enthusiastically Befriended Sequences

The key question in befriending T-sequences is what constitutes a friend. As described in the article **Modular Expansion**, which begins on page 4, most T-sequences are best interpreted on a modular wheel on which the largest value (the modulus) is adjacent to 1. In this view, the modulus and 1 are friends.

The easiest way to deal with this is to expand the sequence as described in the article **Modular Expansion** and consider friendship in the usual way on the result. Figure 6 shows a typical T-sequence and Figure 7 shows its modular expansion, which is friendly.

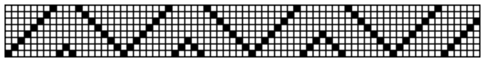


Figure 6. A Point Draw T-Sequence

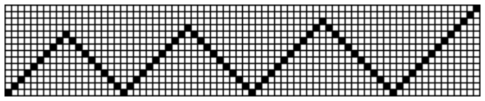


Figure 7. The Modular Expanded Point Draw

If the expanded T-sequence is not friendly, it can be made friendly and then reduced according to the original modulus. Figure 8 shows a T-sequence that is not friendly when it is expanded, as shown in Figure 9. Figure 10 shows the result of conservatively befriending this sequence and Figure 11 shows the result of modular reduction of the sequence by its original modulus.

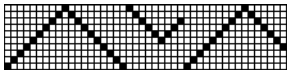


Figure 8. A T-Sequence

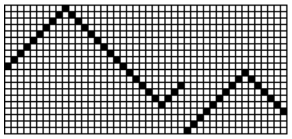


Figure 9. The Modular-Expanded T-Sequence

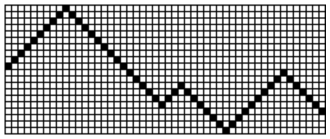


Figure 10. The Befriended Modular-Expanded T-Sequence

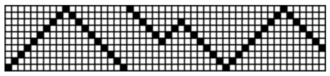


Figure 11. The Reduced Befriended T-Sequence

References

1. "Understanding T-Sequences II", The Icon Analyst 64, pp. 6-12.

2. "Name Drafting", The Icon Analyst 57, pp. 11-14.

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