# The Icon Analyst

*In-Depth Coverage of the Icon Programming Language*

**December 1995**
Number 33

### In this issue ...



## Designing a Visual Interface

This article discusses the design of visual interfaces, using as an example the kaleidoscope program described in the first article of this series [1].

Good visual interface design is a difficult and complicated subject that is beyond the scope of this newsletter. In this and subsequent articles, we'll illustrate common usage by example and comment from time to time on design considerations. For more information on the subject, see References 2-4.

### Planning the Interface

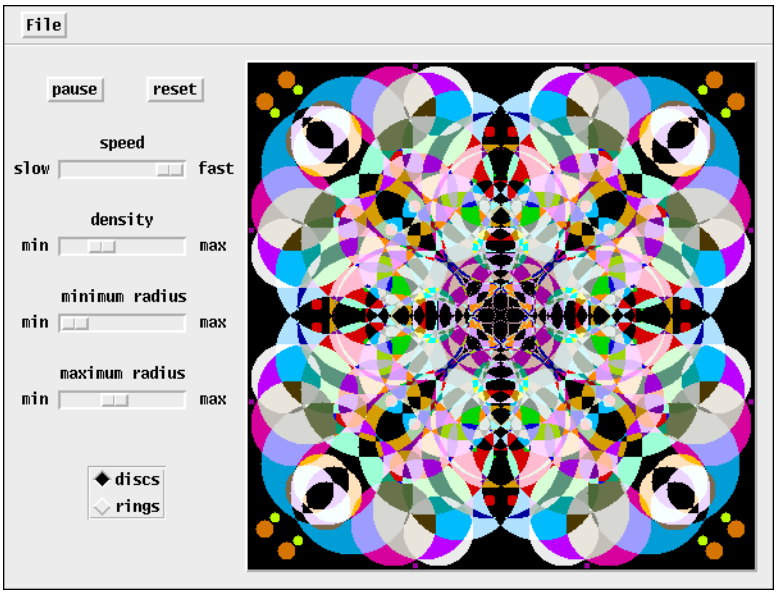
It's important to have a good idea of the functionality of an application before designing an interface for it. It's not necessary, however, to completely implement the functionality of the application before starting to build the interface.

The process of building an application with a visual interface usually is iterative, with focus shifting between the functionality of the interface and the interface itself. Design of the interface may suggest additional functionality or cause features to be cast in ways that are easily represented in the interface.

The interface for the kaleidoscope application is the end product of a process that involved many changes and refinements. We won't attempt to recapitulate the process here. Instead we'll sketch how it might have been done.

Building a visual interface, especially a well-designed and attractive one, is much easier if attention is given to planning before starting to construct the interface.

The size of the application canvas is an important consideration. Changes in the size of the application canvas after an interface is laid out may result in unnecessary work. Screen space often is a limiting factor. Many personal computers have screens that are only 640 by 480 pixels. In designing an application that is intended to be portable to various platforms, it's wise to work within these dimensions.




**The Kaleidoscope Interface**

In many situations, the screen is shared by several applications, so an application canvas generally should not be larger than is necessary. On the other hand, the application canvas should be large enough to be visually attractive and allow the user easy access to interface tools. An application that displays an image or provides user work areas generally is more attractive and useful

with a relatively large canvas. Achieving a good compromise may be difficult.

The image at the bottom of this page shows a sketch of the interface we designed for the kaleidoscope program.

It's often worth doing of series of rough sketches with different layouts before committing to interface construction. Sometimes more precise drawings done to scale, perhaps using graph paper, can save work later.

Our first consideration was the display region. We decided, somewhat arbitrarily, to make the region 400-by-400 pixels (the region needs to be square because of the drawing symmetry). This is large enough to provide an attractive display but small enough so that the entire canvas would fit within the 640-by-480 limit. We put the region at the right side of the canvas because it's conventional to put user controls at the top and left of visual interfaces. Following common, well-known conventions, in the absence of compelling reasons not to, makes learning the application easier for users.

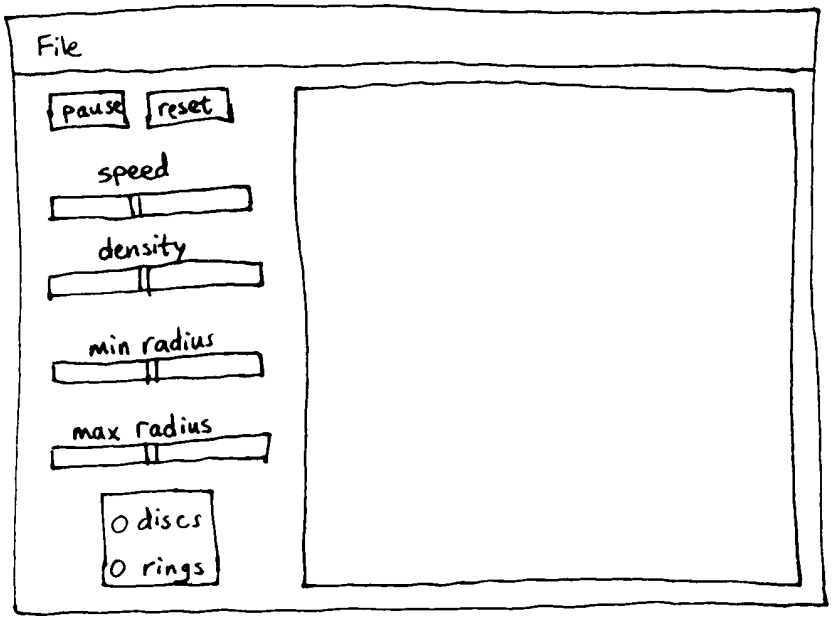
We put a menu bar at the top, also because that's conventional. The functionality we had in mind included the ability to save snapshots of the display. Such operations usually are put in a menu named File. An entry for quitting the application also typically is put in a menu named File, although it has little to do with files. The point is that experienced users expect it there. In this application, there are no other menus; many applications would have others.

Allowing the user to stop the display temporarily and to reset it are part of the application design. We could have put these operations in a menu, but buttons are easier to use than menus and there is ample space on the canvas to provide buttons. Furthermore, since pausing the display involves a change of state, using a button rather than a menu item makes the state visible on the interface.

Since the speed of the display, the density of circles, and the maximum and minimum radii of circles all are numerical quantities, we chose sliders to let the user adjust these values. An alternative would have been to provide text-entry fields in which the user could enter numbers. For an application like the kaleidoscope there is little advantage to allowing the user to specify precise values — entering precise values is harder than moving sliders and the user would need to know what the numerical values mean. Sliders deprive the user of precision, but they allow a more intuitive approach to using the application. Because of the area available and the need to label the sliders, we oriented the sliders horizontally.

All that remains is a way for the user to select between discs and rings. Because there are only two choices and there is space available, we decided to use radio buttons, which make the choice visible on the interface. If there had been more choices for shapes or less available space on the canvas, a menu might have been a more appropriate choice.

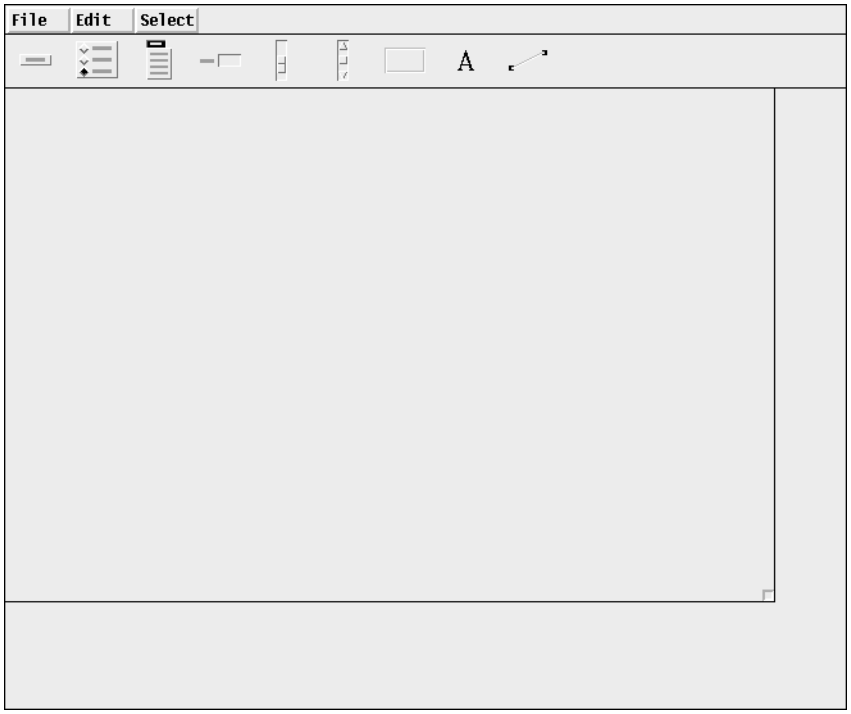
With this layout in hand, we're ready to build the interface.




**A Preliminary Sketch of the Interface**

## A Visual Interface Builder

The Icon program library contains procedures with which you can create vidgets (interface tools), configure them, and position them at specified places on an application canvas. Using procedures to do this, however, is a tedious and often intricate task that requires a substantial amount of specialized knowledge. Icon provides a visual interface builder, VIB, that automates much of this process.




**VIB**

## The VIB Application

The VIB window for building a new interface is shown above.

The menus at the top provide operations needed to use VIB. The icons below the menus represent the vidgets described in the last article. The inner rectangle represents the canvas of the interface being developed.

The icons below the VIB menu bar from left to right represent buttons, radio buttons, menus, text-entry fields, sliders, scroll bars, regions, labels, and lines. Clicking on one of these icons creates a vidget of the corresponding type. It then can be positioned, configured, and so on.

## Next Time

In subsequent articles on building visual interfaces, we'll explain how to use VIB. We'll walk through the process of creating the interface for the kaleidoscope application, taking the blank canvas shown above to the final interface for the kaleidoscope.

## References

1. "Visual Interfaces", Icon Analyst 31, pp. 1-4.

2. Apple Computer Inc. 1987. *Human Interface Guidelines: The Apple Desktop Interface*. Reading, Mass.: Addison-Wesley.

3. Laurel, Brenda, ed. 1990. *The Art of Human-Computer Interface Design*. Reading, Mass.: Addison-Wesley.

4. Open Software Foundation. 1988. *OSF/Motif Style Guide*. Englewood Cliffs, N.J.: Prentice-Hall.

# Dynamic Analysis

This is the fourth in a series on the dynamic analysis of Icon programs — studying what goes on during program execution. In previous articles [1-3], we explained how dynamic analysis is done and showed some results for expression evaluation and storage allocation. This article looks at the ways programs use structures: records, lists, sets, and tables.

It's been several issues since we listed the programs we use for testing; here they are again for reference:

| *program* | *functionality* |
|---|---|
| csgen.icn | sentences from context-free grammars |
| deal.icn | randomly dealt bridge hands |
| fileprnt.icn | character display of files |
| genqueen.icn | solutions to the *n*-queens problem |
| iiencode.icn | text encoding for files |
| ipxref.icn | cross references for Icon programs |
| kwic.icn | keyword-in-context listings |
| press.icn | file compression |
| queens.icn | solutions to the *n*-queens problem |
| rsg.icn | sentences from context-free grammars |
| turing.icn | Turing machine simulation |

Just based on these short descriptions, can you guess what kinds of structures these programs might use? Of course, programs with such

functionalities can be written in many ways, and there often are alternatives for which structures are used and how.

Here's a listing of number of structures that each of these programs create:

| program | records | lists | sets | tables |
|---|---|---|---|---|
| csgen.icn | 0 | 22 | 0 | 2 |
| deal.icn | 0 | 3502 | 0 | 2 |
| fileprnt.icn | 0 | 1 | 0 | 5 |
| genqueen.icn | 0 | 4 | 0 | 0 |
| iiencode.icn | 0 | 1 | 0 | 0 |
| ipxref.icn | 24 | 122 | 0 | 64 |
| kwic.icn | 0 | 205 | 1 | 1 |
| press.icn | 0 | 5 | 1 | 3 |
| queens.icn | 0 | 6 | 0 | 2 |
| rsg.icn | 18061 | 18996 | 0 | 3 |
| turing.icn | 10 | 8 | 0 | 2 |
| total | 18095 | 22872 | 2 | 84 |

The figures for list creation include lists that are created automatically for command-line arguments when the main procedure has a parameter, as all of the test programs do.

What can we say about these figures? We already know that our test programs are not necessarily representative of the range of all Icon programs. We know that a different choice of test programs might have produced significantly different results. Nevertheless, we find these results interesting.

The extensive use of lists attests to their general utility. The use of tables suggests their value in many programs. Nor is it surprising that in most cases, only a few tables are used in all but one program — their most common use is for mapping keys into values, not, for example, in representing structural relationships [4].

We were a bit surprised at how few programs used records, but reflecting on the functionality of the test programs, it's understandable.

Sets come in a distant last. That didn't surprise us. For one thing, at least two of the programs were written before sets were added to Icon. We do think, however, that sets are under-used by Icon programmers. Perhaps that's because sets came along after many Icon programmers had developed different ways of dealing with the problems that sets handle so well. Most programmers also have little experience with using true sets — few programming languages support them (SETL is a notable exception [5]), and sometimes, as in Pascal, the term set is used for something much less general than Icon supports.

The figures in the left column indicate little about how structures are *used*. A single structure may be the focus of much activity during program execution, or it may be used briefly and represent an insignificant amount of activity.

Here's a tabulation for how often records are referenced in the three programs that use them:

| program | number | R.*f* |
|---|---|---|
| ipxref.icn | 24 | 121 |
| rsg.icn | 18061 | 18061 |
| turing.icn | 10 | 38940 |
| total | 18095 | 57122 |

Interesting?

We'll save lists for last, and go on to sets. For sets, the only access method used is member(S):

| program | number | member(S) |
|---|---|---|
| kwic.icn | 1 | 14136 |
| press.icn | 1 | 0 |
| total | 2 | 14136 |

Do these figures seem strange? Incorrect, perhaps? Why, for example, is there no use of insert() in kwic.icn? That's because the set is created from a list that provides its members, as in

```
S := set(L)
```

instead of inserting members one by one. And, yes, press.icn creates a set, but never uses it in the way that the program is run in our tests.

Here are the results for tables, again with only the access methods used by at least one of the programs:

| program | number | T[x] | !T |
|---|---|---|---|
| csgen.icn | 2 | 2 | 0 |
| deal.icn | 2 | 6 | 0 |
| fileprnt.icn | 5 | 41585 | 0 |
| ipxref.icn | 64 | 13485 | 140 |
| kwic.icn | 1 | 38340 | 0 |
| press.icn | 3 | 27166 | 0 |

| | | | |
|---|---|---|---|
| queens.icn | 2 | 5 | 0 |
| rsg.icn | 3 | 18092 | 0 |
| turing.icn | 2 | 2 | 0 |
| total | 84 | 138683 | 140 |

Not surprisingly, although there are not many tables, they are extensively used by more than half of the programs.

Lists are more interesting than the other kinds of structures, because more access methods are used:

| program | number | put(L) | push(L) | get(L) | pull(L) | L[i] | ?L | !L |
|---|---|---|---|---|---|---|---|---|
| csgen.icn | 22 | 19 | 0 | 1 | 1 | 81391 | 47869 | 2642 |
| deal.icn | 3502 | 0 | 2800 | 3 | 1 | 12600 | 0 | 1400 |
| fileprnt.icn | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| genqueen.icn | 4 | 0 | 0 | 0 | 0 | 114132 | 0 | 0 |
| iiencode.icn | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| ipxref.icn | 122 | 7135 | 1 | 4903 | 50 | 2804 | 0 | 6692 |
| kwic.icn | 205 | 12780 | 0 | 405 | 2 | 1 | 0 | 202 |
| press.icn | 5 | 2 | 1 | 4 | 1 | 2 | 0 | 1 |
| queens.icn | 6 | 133 | 0 | 2 | 0 | 144535 | 0 | 352 |
| rsg.icn | 18996 | 0 | 0 | 42286 | 1 | 0 | 18061 | 15 |
| turing.icn | 8 | 5 | 0 | 1 | 1 | 22880 | 0 | 0 |
| total | 22872 | 20074 | 2802 | 47605 | 57 | 378348 | 65930 | 11304 |

*Notes:* The figures for !L are for the number of list elements generated, a form of subscripting. get() and pop() are equivalent. There are no uses of pop() in the test programs. The listing of pop() in the article on expression activity [2] was an error; it should have been get() instead.

We were a bit surprised by the fact that pop() was not used at all, but the uses of pull() in so many programs seemed suspicious to us; it's a function that's rarely used. The use in ipxref.icn is clear, but pull() doesn't even appear in any of the other programs. Then we realized that pull() is used in options(), which is linked by most of the test programs.

It's clear from the figures above that list subscripting figures in a large way in many programs. It's no surprise that ?L occurs in csgen.icn and rsg.icn; both do random generation.

## What Does it All Mean?

Several of the test programs rely heavily on the use of lists and tables. Although there is no way 11 programs can be representative of the class of all Icon programs, examination of many programs suggests that the use of these structures in the test programs is common to many programs.

It would require a much larger test suite to draw any conclusions about records and sets. We may attempt that later, but as we said in Reference 1, it's hard to come by programs that are suitable for testing, and dynamic analysis is very computationally expensive.

## What Else?

You'll note we listed only access to structures, not the many operations that can be performed on them, such as *X, copy(X), sort(X), S1 ++ S2, and so on. We expect to explore such things in the future.

There are some things we'd like to know that are hard to come by with our present monitoring system, such as:

- How large are structures and how are their sizes distributed? (Recall that all kinds of structures except records can change in size during program execution, which makes such studies even more problematical.)
- How many structures are used transiently

and hence subject to garbage collection, as opposed to structures that persist throughout a major portion of program execution.

• How many structures actually are garbage collected?

And so on. Once you get into questions like these, there is no end. With enough effort, of course, we can get answers to such questions. But such investigations become more and more specialized. We dread the possibility of a doctoral dissertation titled something like "A Study of List Access Methods in Icon and Their Impact on the Performance of Applications that Analyze World Wide Web Pages".

### Disclaimer

In studying structure usage in Icon programs, we found a few glitches in our event-monitoring instrumentation. We'll fix these in a future release of Icon, but if you use the present event-monitoring instrumentation, you'll run into some problems interpreting the results.

### More to Come

There's no end to what we can do with dynamic analysis, even given only our present tools. At some point such studies become tedious and of limited interest.

We do have material on type conversion, string operations, numerical operations, and so forth, which we plan to present in future articles.

These articles probably will be spaced out over a year or two. If there's an aspect of program execution in Icon that is of particular interest to you, let us know and we'll try to give it priority.

### References

1. "Dynamic Analysis of Icon Programs", Icon Analyst 28, pp.9-12.

2. "Dynamic Analysis of Icon Programs", Icon Analyst 29, pp. 10-12.

3. "Dynamic Analysis", Icon Analyst 30, pp. 6-11.

4. Griswold, Ralph E. and Griswold, Madge T. 1990. *The Icon Programming Language, Second Edition*. Englewood Cliffs, N.J.: Prentice-Hall.

5. Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., and Schonberg, E. 1986. Programming with Sets: An Introduction to SETL. New York, N.Y.: Springer-Verlag.



## Versum Sequence Mergers

As we showed in the last issue of the Analyst, a simple observation and a relatively simple calculation allows the determination of all equivalent *n*-digit versum sequence seeds. The result is a dramatic reduction in the amount of computation and storage space that is needed to study versum sequences.

Another potential source of savings is in versum sequences that start out differently but merge to a common term. For example, the sequence for the seed 1 is

1: {2, 4, 8, 16, 77, 154, 605, 1111, 2222, … }

It's obvious that the seed that's a term in this sequence immediately merges to it. For example, the sequence for 2 ,

2: { 4, 8, 16, 77, 154, 605, 1111, 2222, … }

is just the trailing part of the sequence for the seed 1. Of course, the sequence for 1 merges to the sequence for 2 after one term. To keep track of mergers, we'll deal with seeds in numerical order and merge the sequence for seed 2 to the sequence for seed 1, rather than the other way around.

Sequences also merge to sequences with smaller seeds after initial terms that do not merge to other sequences. For example, the sequence for 104 is

104: {505, 1010, 1111, … }

Although 505 and 1010 are not terms in sequences with smaller seeds, 1111 is the eighth term in the sequence for seed 1. Thus 1 and 104 have sequences with different "heads" but a common "tail".

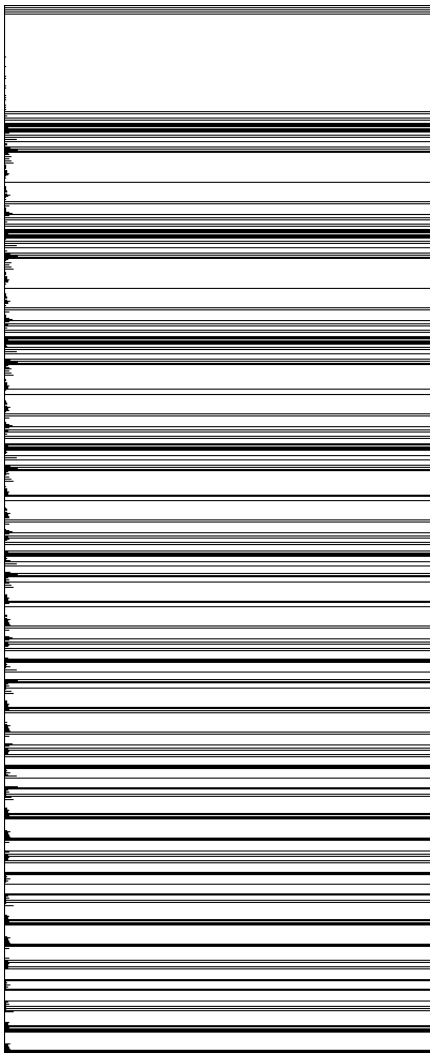
In order to perform computations on versum sequences, it's necessary to have the heads for all primary seeds, but the tails only for those sequences that do not merge to others. We'll call the seeds for sequences that do not merge to others *base seeds*. It's only the sequences for base seeds, then, that need to be carried out to the maximum number of terms needed.

You might wonder if all sequences eventually merge to a single base sequence. Empirical evidence strongly suggest that this is not the case, but it's unlikely that such a conjecture could be proved or disproved. We'll be satisfied in referring to base sequences as those that do not merge to others after a relatively large number of terms (like 500). We'll have more on this later.

Several questions concerning merging versum sequences come to mind:

- How common are mergers?
- More specifically, what percentage of *n*-digit primary sequences are base sequences?
- How far out do mergers occur?
- How much space for storing versum sequences can be saved by storing only the heads of non-base sequences?
- How should we organize the data if we store only the heads of non-base sequences?

It doesn't take much exploration to discover that mergers are quite common and that they usually occur after only a few terms. The image at the right shows how far sequences go before merger for seeds 1 through 999. Note that this image shows versum mergers for all seeds, many of which are equivalent.



**Versum Mergers for Seeds 1 Through 999**

It's clear from the amount of white space in this image (that is, terms after mergers), that there's a lot to gain in using merger information.

The information needed to represent non-base sequences is easily encoded, as in

104: {505, 1010, 1:8 }

where 1:8 is the terminating merger term.

With this in mind, we can create versum sequences, computing all terms for base sequences and only the heads for sequences that merge to others.

The approach we took to computing versum sequences with mergers was to create versum sequences for primary seeds in numerical order, keeping versum terms in a table. Then, when a new term is computed, if it's in the table, the sequence for the corresponding seed merges to a previously computed seed. By taking seeds in numerical order, we assured, as mentioned earlier, that the base seeds are the smallest members of their equivalence classes.

Before going on, we need to deal with another problem. Although the number of *n*-digit primary seeds is much smaller than the number of all *n*-digit seeds, as *n* gets large, keeping the sequences for all primary seeds in separate files becomes a significant problem for even modest values of *n*. As shown in the article on equivalent versum sequences, the number of primary seeds for *n*=6 is 6,498, for *n*=7, 64,980, and for *n*=8, 123,426.

Some operating systems handle a large number of files better than others, but few can do much with a large number of files in the same directory. All kinds of things go wrong. For example, if the

sequences for all 6-digit primary seeds are kept as separate files with the suffix .vsq in one directory, on a UNIX platform, attempting to list them by

```
ls *.vsq
```

is likely to produce the message

```
Arguments too long.
```

You can imagine various things to do about such problems, such as organizing files in subdirectories, combining several sequences in one file, and so on. All of these lead to complexities and maintenance problems. Without taking a Draconian approach (like recomputing versum sequences as needed), the sheer number of sequences becomes the limiting factor as *n* becomes larger.

Dealing with mergers provides an opportunity to reduce the number of files needed for versum-sequence information. Since the number of base seeds appears to be small compared to the number of primary seeds, and the number of terms in the heads of non-base sequences appears to be small on average, it seems reasonable to store the sequences for base seeds in separate files and put all the information for non-base seeds in a single file. It remains to be shown that this approach is practical — for example, that the file of non-base information is not monstrously large. It seems worth trying, in any event.

The program above uses this approach. The file containing merger information for non-base sequences is named vsq.mrg. The terms in a sequence are kept in a list until it is known whether the sequence is a base sequence or one that merges to one. In the first case, a new file is created and the terms written to it. In the second case, the seed followed by an identifying mark and its terms are appended to vsq.mrg. Some portions of vsq.mrg are shown in the center column of the next page.

It's worth noting that the technique we've used creates multiple merger links, as in

104: {505, 1010, 1111, 1:8}

109: {104:2}

```
link options
link pvseeds

procedure main(args)
   local opts, n, limit, merge_tbl, i, j, k, count, output, name
   local tlist, merge_file, merge

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

   n := \opts["n"] | 3                          # number of digits; small default
   limit := \opts["l"] | 30                     # maximum number of terms

   merge_file := open("vsq.mrg", "w") |         # non–base sequences go here
      stop("*** cannot open vsq.mrg")

   merge_tbl := table()                         # mergers for terms

   every i := pvseeds(1 to n, 1) do {           # primary seeds through n digits
      j := i
      tlist := []                               # list of terms
      output := &null                           # no output file yet
      every count := 1 to limit do {
         j +:= reverse(j)                       # next term
         if merge := \merge_tbl[j] then {       # term is already in table
            put(tlist, merge)                   # save merge information
            output := merge_file                # file for merger information
            break                               # terminate loop for this seed
            }
         else {                                 # term not in table
            put(tlist, j)                       # add term
            merge_tbl[j] := i || ":" || count   # merger information
            }
         }
      if /output then {                         # no file; new base sequence
         output := open(i || ".vsq", "w") |
            stop("*** cannot open ", i, ".vsq")
         every write(output, !tlist)
         close(output)
         }
      else {                                    # identify the seed
         write(output, i, "=")
         every write(output, !tlist)
         }
      }
end
```

**Program to Create Versum Sequences with Merger Information**

Computing versum sequences and merger information in this way is not without its problems. It's necessary to create sequence information up to the chosen limit for all primary seeds up to the value of *n* chosen, all in one run. If *n* is even moderately large, like 6, and a few hundred terms are needed, the amount of memory to store all terms in all base sequences is *very* large.

Using an DEC Alpha workstation with 96 MB of RAM, we've managed to push *n* to 8 (on an Alpha with "only" 64 MB, swapping brought the machine to its knees, and it's unlikely that the process for *n*=8 would have completed before the machine crashed. Our patience was considerably more limited).

Having done the computation through *n*=8, we have enough information about the number of base sequences to be interesting:

| *n* | *primary seeds* | *base seeds* | *ratio* |
|---|---|---|---|
| 1 | 9 | 5 | 0.55556 |
| 2 | 18 | 0 | 0.00000 |
| 3 | 139 | 41 | 0.29496 |
| 4 | 342 | 16 | 0.04938 |
| 5 | 3420 | 464 | 0.13567 |
| 6 | 6498 | 220 | 0.03385 |
| 7 | 64980 | 4953 | 0.07622 |
| 8 | 123462 | 3061 | 0.02479 |
| *sum* | 198860 | 8760 | 0.02931 |

Remember that the number of base sequences given here is conjectural. It's certainly possible that some might merge if taken to more terms. However, the evidence strongly suggests otherwise; for *n* through 8 and 500 terms, the maximum number of steps to a merger is only 24 and more than 93% of all mergers occur within 3 steps.

From the information above, it's obvious that much less data is needed for storing only base sequences and merger information. For *n*=1 though 8, it takes "only" about 95 MB of disk space to store all the information; about 5 MB for vsq.mrg and 90 MB for the base sequences. From the ratios above, you can estimate the amount of space that would be required for storing sequences for all primary seeds.

```
vsq.mrg
2=
1:2
4=
1:3
6=
3:2
8=
1:4
10=
5:2
11=
5:3
12=
3:3
13=
5:4
14=
7:2
15=
3:4
16=
1:5
17=
5:5
18=
9:2
19=
7:3
29=
7:4
39=
3:5
49=
143
7:6
59=
1:6
...
409=
1313
1:10
509=
106:2
609=
1515
102:5
709=
100:5
809=
1717
1:11
909=
108:2
110=
7:4
115=
626
1252
3773
5:9
219=
1131
2442
3:9
719=
1636
7997
15994
65945
120901
229922
459844
908798
1806607
7:15
...
```

Incidentally, there are suggestions of patterns of digits in the seeds for base sequences. There are too many of them to show for all but the most modest values of *n*. Here are the ones for *n*=3:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 100 | 112 | 118 | 133 | 184 | 399 | 739 | 879 | 999 |
| 102 | 113 | 119 | 135 | 186 | 459 | 759 | 919 | |
| 106 | 114 | 122 | 137 | 196 | 539 | 779 | 939 | |
| 108 | 116 | 124 | 138 | 199 | 659 | 799 | 959 | |
| 111 | 117 | 128 | 166 | 359 | 679 | 859 | 979 | |

There are fewer for *n*=4:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1000 | 1011 | 1022 | 1037 | 1057 | 1068 | 1088 | 6999 |
| 1006 | 1013 | 1033 | 1046 | 1066 | 1077 | 4999 | 8999 |


# The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
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-project@cs.arizona.edu

THE UNIVERSITY OF ARIZONA®
TUCSON ARIZONA

*and*

***Bright Forest Publishers***

***Tucson Arizona***

There are more striking patterns for larger values of *n*, but there are too many to show here. Here's a curiosity for you to ponder, however: At least for *n*=1 through 8, all base seeds that start with a digit greater than 1 end in the digit 9. Can you prove or disprove this?

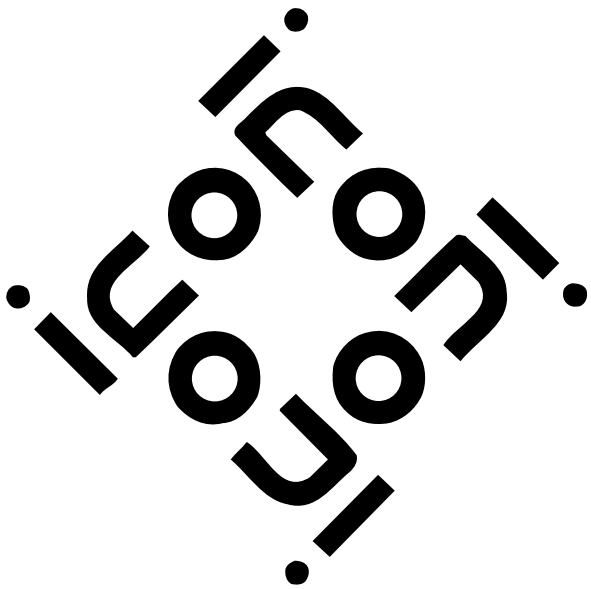
Treating base sequences the way we have has complicated one aspect of using versum sequences. The procedure vsterm(i), which generates the terms in the sequence for seed i, now must deal with vsq.mrg and the way information in it is stored.

As with all the things we're considering, there are various approaches. We chose to have vsterm() read vsq.mrg the first time it is called, putting the information it contains in a table. We chose to have table keyed by non-base seeds whose corresponding values are lists of the terms in the head of the sequence, ending with the merger term.

The code is a bit lengthy and complicated, but it's not that difficult to write:

```
link vprimary

procedure vsterm(i)
   local term, merge_file, tlist, j, k, line
   static input, merge_tbl

   initial {
      merge_tbl := table()

      terms := 0

      merge_file := open("vsq.mrg") | {
         write(&errout, "*** cannot open vsq.mrg")
         fail
         }

      while line := read(merge_file) do {
         line ? {
            j := integer(tab(upto('=')))
            merge_tbl[j] := tlist := []
            while term := read(merge_file) do {
               if term := integer(term) then put(tlist, term)
               else {
                  put(tlist, term)
                  break
                  }
               }
            }
         }
      close(merge_file)
      write(&errout, "\ninitialization done")
      }

   close(\input)

   k := 0
   i := vprimary(i)

   if tlist := \merge_tbl[i] then {
      k := 1
      repeat {
         term := tlist[k]
         if term := integer(term) then suspend term
         else {
            term ? {
               i := integer(tab(upto(':')))
               move(1)
               k := integer(tab(0)) |
               if tlist := \merge_tbl[i] then next
               else break
               }
            }
         k +:= 1
         }
      }

   input := open(i || ".vsq") | {
      write(&errout, "*** cannot find sequence for ", i)
      fail
      }

   every 1 to k – 1 do read(input)

   while term := read(input) do
      suspend integer(term)

   close(input)

end
```

The functionality of vsterm() is the same as it was before, and programs that use it need not be changed for the new way of recording versum sequences.



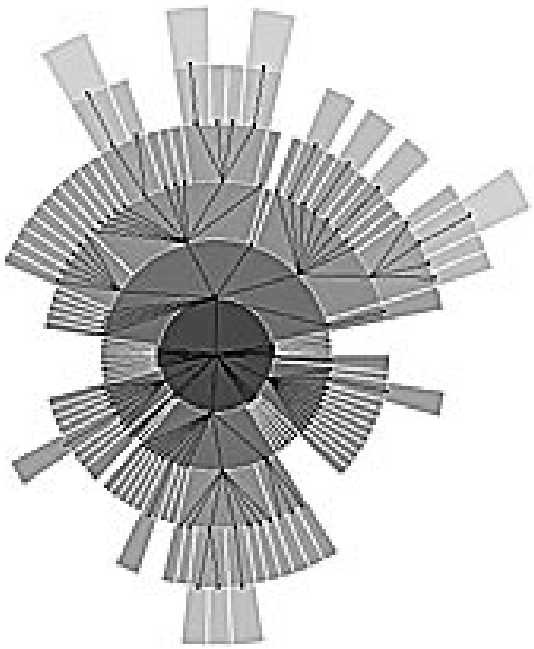
## Merger Trees

One thing we began to wonder about when working with versum sequence mergers was the structure of mergers. Clearly, they're trees. And, of course, the nature of the trees depends on the particular way we've cast mergers and computed them.

Here's the program we used to produce string encodings of trees from vsq.mrg:

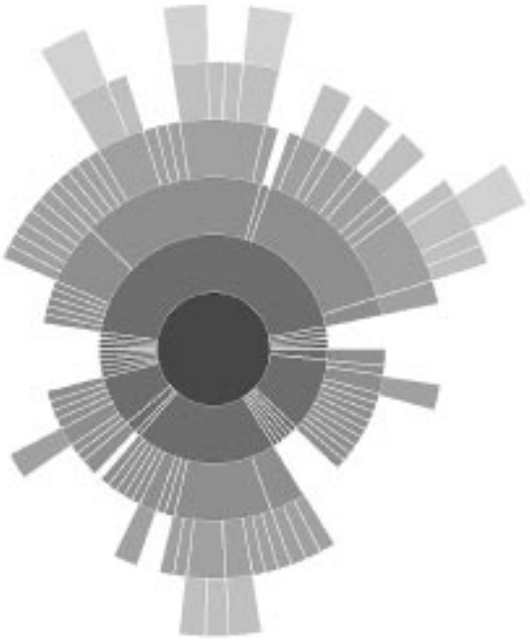
```
record arc(head, tail)

global seed_sets
global seed_names

procedure main(args)
   local n, seeds, arcs, i, x
   local merge_file, base, line, head, tail, node

   n := (0 < integer(args[1])) |
      stop("*** invalid command-line argument")

   merge_file := open("vsq.mrg") |
      stop("*** cannot open vsq.mrg")
   base := open("base." || n) |
      stop("*** cannot open base.", n)

   seeds := set()
   arcs := []
   seed_names := table()
   seed_sets := table()

   #  Process the merger information.

   every line := !merge_file do
      line ? {
         if head := integer(tab(upto('='))) then {
            if *head > n then break
            insert(seeds, head)
            }
         else if tail := integer(tab(upto(':'))) then {
            insert(seeds, tail)
            put(arcs, arc(head, tail))
            }
         }

   #  Create a set for each seed and build
   #  cross reference.

   every name := !seeds do {
      node := set()                    # create new node
      seed_sets[name] := node          # name to node
      seed_names[node] := name         # node to name
      }

   #  Insert the arcs.

   every x := !arcs do
      insert(\seed_sets[x.tail], \seed_sets[x.head])

   #  Output trees for every base name.

   every i := integer(!base) do
      write(tree(\seed_sets[i]))

end

#  Construct strings for merge trees.

procedure tree(node)
   local children, node_names

   children := ""
   node_names := []
   every put(node_names, seed_names[!node])
   every children ||:=
      tree(seed_sets[!sort(node_names)])

   return seed_names[node] || "[" || children || "]"

end
```

These trees gave us a chance to try out a visualization tool that is designed to provide a variety of ways of viewing trees. Here's a visualization of the merger tree leading to the base seed 9 for *n*=1 through 6.



This visualization uses rings to allow "wide" trees to be represented in an understandable way. Labels are omitted to allow the structure to be understood more easily.

The conventional tree diagram superimposed on the rings can be omitted, giving a simpler overall view of the tree:



We can easily see that merge links are at most 6 levels deep, but the tree has considerable breadth. The merger tree for seed 9 is larger than for most base seeds, but its shape is similar to that of most others.

We don't know what the nature of merger trees might say about versum sequences, but we find them interesting, nonetheless.

### Acknowledgment

The tree visualization program was written as an honors project by Michael Shipman, a Computer Science senior. You've seen his name before; his project in the recent graphics programming course was one of the best and was featured in a recent *Icon Newsletter*.

### Icon on the Web

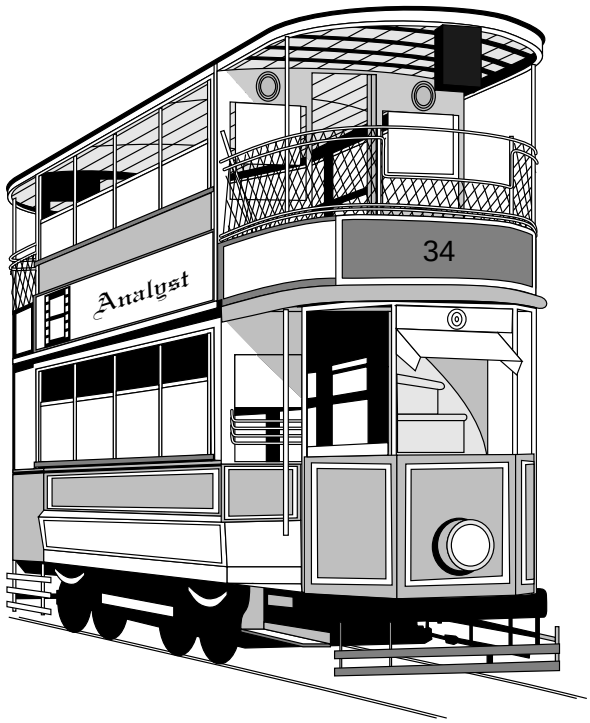
Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/www/

### Next Time

The observations and techniques described in the last two articles on versum sequences have made it possible to study their properties for much larger values of *n* than otherwise would have been possible.

Now it's time to use this capability to explore at greater length the original motivation for the study of versum sequences: palindromes. We'll tell you in advance that we haven't cracked the "big question": whether all versum sequences contain palindromes. But we do have some interesting results about the location of palindromes in versum sequences and the nature of their structure.




## What's Coming Up

Next time we'll continue our series on building visual interfaces for Icon programs, working through the kaleidoscope application in VIB.

As indicated above, there's also another article on versum sequences in the works.

We also expect to have an article on another of the most useful programs in the Icon program library and perhaps a programming tip or two.