The Icon Analyst

In-Depth Coverage of the Icon Programming Language

August 1997
Number 43

In this issue …

New Services for Subscribers.... 1
Kaleidoscopic Visualization 1
Factors of Versum Numbers 9
Debugging: Library Support... 14
What's Coming Up 16

New Services for Subscribers

Starting with this issue of the Analyst, we're placing images from Analyst articles on our Web site. This will allow you to see the colors and, of course, copy the images for your own use. To get to the images, start with the Icon home page:

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

In the **Documentation** section of our home page, follow the link to The Icon Analyst. On that page, follow the link Supplementary material for subscribers. On that page you'll find links to issues of the Analyst (right now, there is only Icon Analyst 43).

Since this is a service only for Analyst subscribers, there won't be any further explanation except for figure captions.

As time permits, we'll also add images from recent past issues of the Analyst.

Kaleidoscopic Visualization

In previous articles [1-5], we explored various ways of analyzing the behavior of Icon programs in nonvisual ways. In Icon Analyst 37 [6] we started a series of articles on visual tools for studying program behavior. In this second article on the subject, we'll show how the kaleidoscope program [7,8] can be converted into a visualization tool.

The kaleidoscope is just a visual amusement that produces a display based on randomly selected values. For visualization, we'll change the program to produce a display based on events that occur during Icon program execution using MT Icon [9]. We'll start with visualizing storage allocation — allocation is a good subject for visualization and there are many existing tools to which this new one can be compared [10,11].

Visualizations that produce abstract designs may seem silly if attractive. They have a serious purpose, however, and there are good reasons for using abstract displays and patterns. So-called right-brain thinking [12,13] perceives relationships in patterns that aren't evident to the analytic, word- and number-oriented left brain. This is an oversimplification, and we'll discuss the subject in more detail in a later article. For now, we'll just show how easy it is to convert a visual amusement into a tool for viewing the behavior of Icon programs.

Forewarning: Before launching into the technical details, we need to warn you that the end result will be seriously flawed. There are ways to reduce the impact of the flaw, and the result is still useful. We'll explain later on.

The Interface

Figure 1 on the next page shows the interface for the kaleidoscope as it appears in VIB. By way of review, pause is a toggle to stop and start the display, reset clears the display to start afresh, speed controls how fast the display runs, and density limits the number of simultaneously displayed circles. The radius sliders control the size of the circles, and the radio buttons at the bottom provide a choice between filled circles and outlines.

Most of these controls make sense for visualization. The sizes of the circles, however, can be

used to show the amount of space allocated. In place of the radius sliders, we'll add a slider to control scaling of allocation amounts. The new interface is shown in Figure 2.

Notice that we've added labels to the density slider to allow the user to set the density more accurately than for the kaleidoscope application.

Modifying the original interface in VIB is easy. Deleting a radius slider is just a matter of selecting it and entering @X or selecting delete from the edit menu.

The scale slider is a routine addition — just copying a few vidgets and adjusting them. Adding the labels and tick marks for the density slider is a bit tedious, but VIB's alignment features make precise positioning easy.

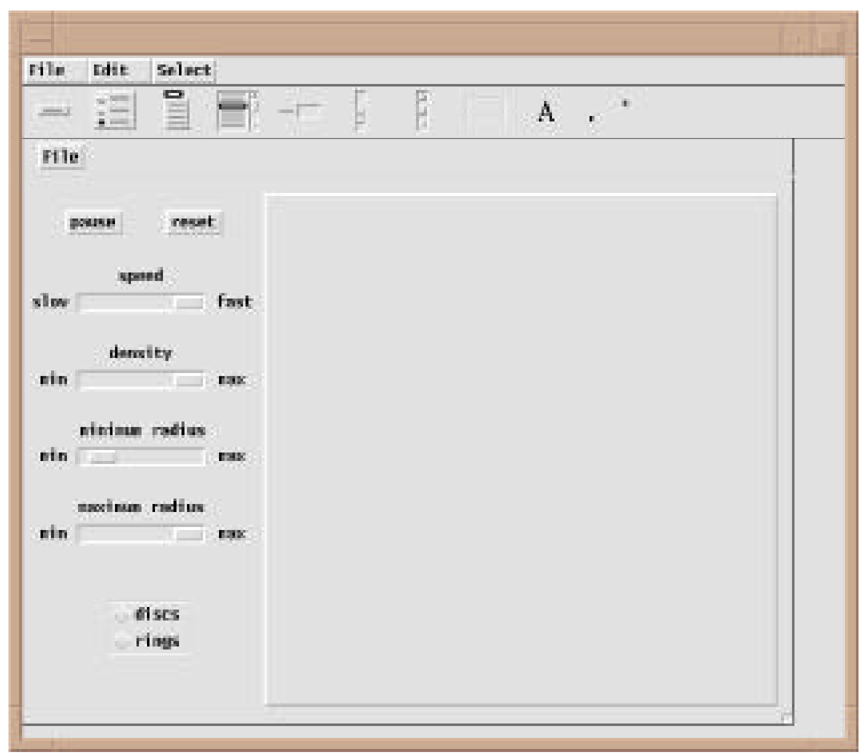


Figure 1. The Kaleidoscope Interface

Specifying Monitoring

Several things need to be done to adapt the program itself to monitoring and presenting allocation events visually. One is to decide how the user specifies the program to be monitored, its input data, and possible command-line options.

It seems as if a program with a visual interface should allow the user to specify these by means of menus and dialogs. In a command-line environment, however, it turns out to be easier for the user to specify these when the program is launched, much like the nonvisual monitoring programs we've described [1-5]. For example, if the kaleidoscopic visualizer for allocation is named alcscope, it might be launched as follows:

```
alcscope concord <concord.dat
```

where concord is the program to be monitored (the SP in earlier terminology) and concord.dat provides its input.

We'll use this approach here and leave open the possibility of interactive specification of the SP, its data, and so on for a later time.

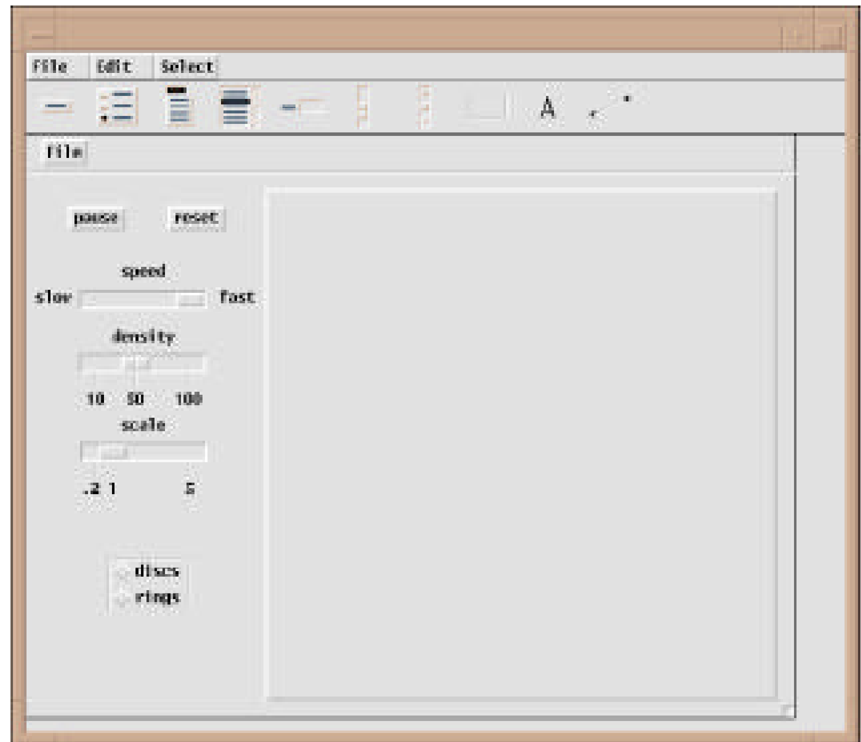


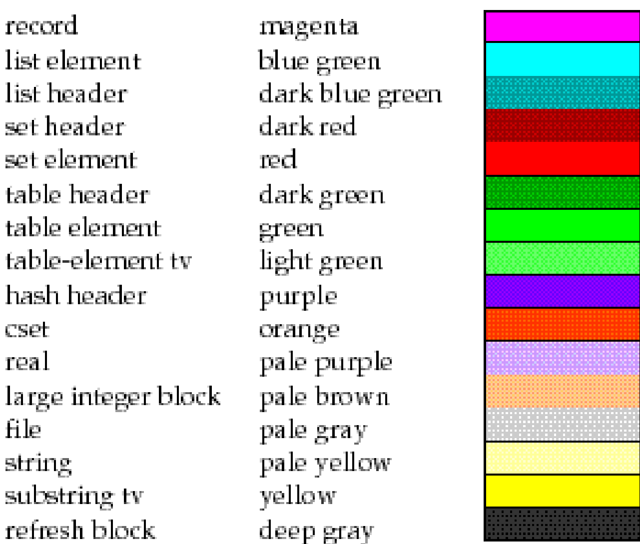
Figure 2. The Visualization Interface

Given the command-line interface described above, existing support for monitors (MPs) can be used.

Translating Allocation Events to Images

In an earlier article [14], we discussed the category-magnitude model of events. This model nicely fits allocation, since there are relatively few different kind of allocation, and the amount of allocation is of obvious interest. As before, we'll use colors to identify the different kinds of allocation [10,11]. Thus, an allocation event produces a symmetric drawing of a circle of a specified size and color.

We've tried many color palettes for allocation. The current one we're using emphasizes bright colors for easy identification and related colors for allocations for structures where there is more than one internal type. For example, a list consists of list header block and one or more list-element blocks. This standard palette is shown in Figure 3.



record	magenta
list element	blue green
list header	dark blue green
set header	dark red
set element	red
table header	dark green
table element	green
table-element tv	light green
hash header	purple
cset	orange
real	pale purple
large integer block	pale brown
file	pale gray
string	pale yellow
substring tv	yellow
refresh block	deep gray

Figure 3. Allocation Color Coding

The notation "tv" is shorthand for trapped variable. For an explanation of the various types of allocation, see References 15 and 16.

Of course, you can't see the actual colors in this printed copy of the Analyst, but check out the image on our Web site.

Back Issues

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

The visualization program adds links and an include file for monitoring, as well as an argument to the main procedure:

```
link colormap            # color information
link evinit              # monitoring support

$include "evdefs.icn"    # event and mask definitions

procedure main(args)
      …
   init(args)
      ...
```

The command line arguments are passed to init(), which loads the SP and suppresses its output:

```
procedure init(args)
      …
   EvInit(args) | ExitNotice("Cannot  load the SP.")

   variable("write" , &eventsource) := –1
   variable("writes" , &eventsource) := –1
      …
```

There also are changes in the initialization related to the changed interface tools. See the complete listing at the end of this article.

SP Event Processing

There are no changes in the event loop that handles user interaction. SP events are processed in putcircle(), which does the drawing.

Instead of choosing random parameters for circles, SP events are used instead:

```
procedure putcircle()

   EvGet(AllocMask) | ExitNotice("SP terminated.")

   fg := color[&eventcode]
   radius := sqrt(&eventvalue    scale)
      …
```

The table color has keys for the event codes related to allocation with color values corresponding to the palette shown earlier. The global variable scale contains the user-specified scale, which is set by the callback procedure scale_cb():

```
procedure scale_cb(vidget, value)

   scale := value

   return

end
```

There are no other changes to the program.

Visualization Examples

An example of an alcscope display, taken from rsg, a program that generates random sentences, is shown in Figure 4:

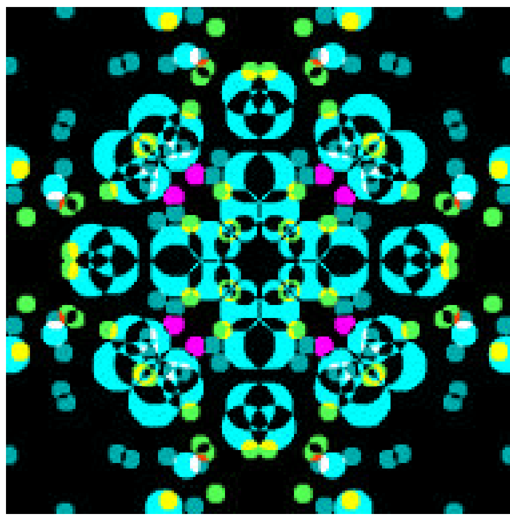


Figure 4. An Example Visualization

Several types of allocation are shown here: primarily list headers, list elements, hash headers, table elements, and substring trapped variables. See our Web site for the actual colors.

One observation in comparing displays from kaleido and alcscope is that although alcscope uses fewer different colors than kaleido, its displays are just as visually interesting.

"Phase changes", in which a program goes from allocating some kinds of data to allocating other types, are particularly noticeable. Consider this program for producing concordances:

```
global uses, lineno, width

procedure main(args)
   local word, line

   width := 15                        # width of word field
   uses := table()
   lineno := 0

   every tabulate(words())            #get the citations

   output()                           # write the citations

end

procedure tabulate(word)
   /uses[word] := set()
   insert(uses[word],lineno)

   return

end

procedure words()
   local s, line

   while line := read() do {
      lineno +:= 1
      write(right(lineno,6),"  ",line)
      map(line) ? while tab(upto(&letters)) do {
         s := tab(many(&letters))
         if   s < 3 then next          # skip short words
         suspend s
         }
      }

end

procedure output()
   local word, line, numbers

   write()

   uses := sort(uses,3)               # sort citations

   while word := get(uses) do {
      line := ""
      numbers := sort(get(uses))
      while line ||:= get(numbers) || ", "
      write(left(word,width),line[1:–2])
      }

end
```

There are two quite distinct phases in this

program: collecting the words and tabulating the words with their line numbers — in the procedures words() and tabulate() — and formatting and outputting the results — in the procedure output()).

Figure 5 shows allocation during the phase when words are being collected and tabulated.

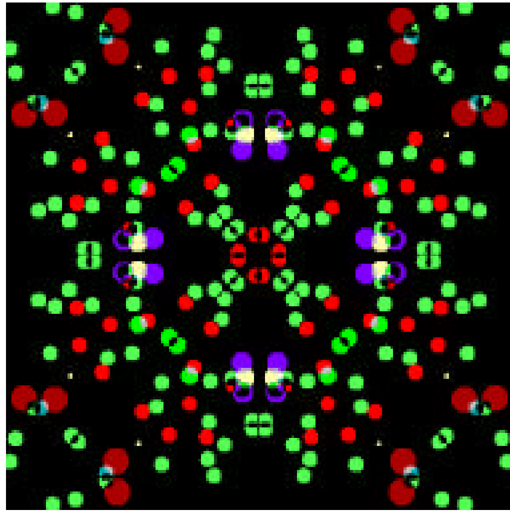


Figure 5. Allocation During Cataloging

This image shows several kinds of allocation: primarily set headers, set elements, table elements, ann substring trapped variables.

Figure 6 shows the output phase when words are being written.

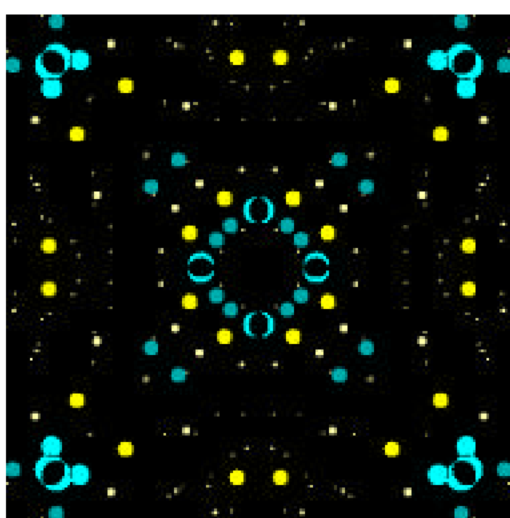


Figure 6. Outputting the Concordance

The larger circles in the center and the corners represent list header and list element allocation. The somewhat smaller circles around the edges and the center represent substring trapped-variable allocation. The tiny circles represent string allocation.

A particularly striking image occurs at the beginning of output() where the table uses is sorted to produce a list. Figure 7 shows the result:

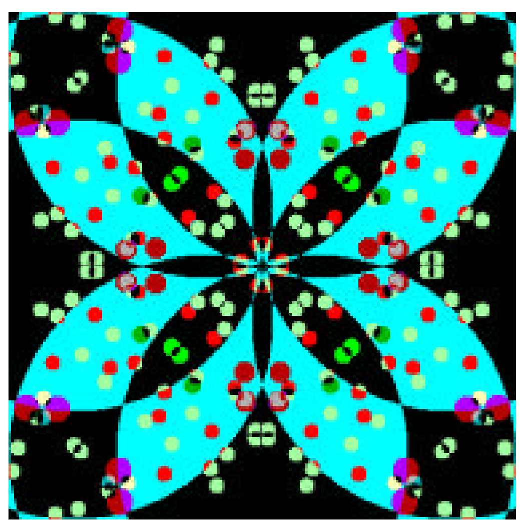


Figure 7. List Allocation from Sorting a Table

The reason the circles are so big is that when a list is created by sorting, all the elements are put in one list-element block. In this case, the block is quite large. When watching visualizations of allocation that show the size, such an effect usually results from sorting, and it commands attention.

You'll find all these images above on our Web site, as well as some others for which there was not space here.

Observations About the Program

The conversion from kaleido.icn to alcscope.icn was easy. Not only were there few changes to the program proper, but the conversion was essentially routine, accomplished quickly, and without any problems. The visualization program actually is 22 lines shorter than the kaleidoscope program on which it was based.

Although we chose storage allocation for this example, the program easily can be adapted to visualizing other kinds of program activity, such as string creation. In fact, it is not difficult to extend the program to allow the user to select interactively the kind of activity to visualize, switching between them while the program runs. All that is needed is a dialog for the choices. For kinds of activity where

there is no natural magnitude component associated with events, a constant value or randomly selected values can be used. The hardest part seems to be finding suitable color palettes.

The Flaw

The drawing attribute drawop=reverse is used so that previously drawn circles can be erased to maintain a constant density.

Reversible drawing works like you'd expect when the foreground and background colors don't change. When they change (in this case, the foreground color), the results are unpredictable. In particular, if a circle is drawn on top of previously drawn circles, the previously drawn circles "show through". The colors that show through are unpredictable, in the sense that they may not correspond to any colors previously used in drawing and in fact, may depend on colors used by other applications.

This is not a problem for the kaleidoscope, where you have no idea what the colors are supposed to be, but in visualizations in which colors are used to code different kinds of allocation, the artifact can be misleading. This problem is particularly noticeable in Figure 7. When the large circles for the list element block are drawn, the underlying circles show through, but in altered colors (see out Web site).

This property of reversible drawing is not a bug. It is an undesirable feature that is a by-product of being able, for example, to erase a drawing that was created at arbitrary time in the past.

What actually is going on is not easy to explain, although we'll try to muster the courage and energy for a future article. For now, take it as given.

Does this flaw render the visualization technique unusable or worse, completely misleading? Not entirely. The artifacts of reversible drawing can be reduced in two ways:

- reduce the density of the display so that there are fewer overlaps in drawing
- use rings instead of discs

Using rings instead of discs greatly reduces the amount of overlapping drawing, and the continuity of the lines reduces the visible effect of incorrect colors where there is overlapping. This is shown in Figure 8, which is a ring-visualization corresponding to the disc visualization of Figure 8.

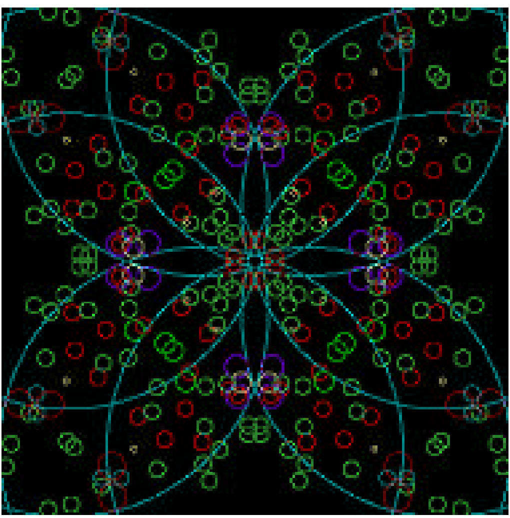


Figure 8. List Allocation from Sorting a Table

An alternative is to dispense with reversible drawing altogether and let the drawing pile up, as was done in the splatter visualization [11]. The visual effect is quite different and not nearly as comprehensible as maintaining a constant density of circles.

Yet another alternative is to clear the non-erasing version frequently to prevent excessive buildup. This, however, produces annoying discontinuities in the display.

If alcscope is used as-is, even restricted to rings, it's important to clearly identify the problem to users — several students have used the program without knowing of the problem and never noticing that anything was wrong. Too right-brained, perhaps? Not our students.

Program Listing

A complete listing of alcscope.icn follows. Comments have been removed to reduce the bulk; they are mostly the same as in the kaleidoscope program. You may notice some differences from the listing of kaleido.icn given in Reference 8 that are not related to the conversion to visualization. The reason for this is that kaleido.icn is our main example of a graphics program with a visual inter-

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

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

face. We try to improve on it when we can, and alcscope.icn is based on the latest version of kaleido.icn.

```
link colormap
link evinit
link interact
link vsetup

global vidgets
global root
global size
global half
global pane
global delayval
global density
global draw_proc
global scale
global color
global draw_list
global reset
global state

$include "evdefs.icn"

procedure main(args)

  init(args)

  kaleidoscope()

end

procedure init(args)

  vidgets := ui()

  root := vidgets["root"]
  size := vidgets["region"].uw
  if vidgets["region"].uh ~= size then
    stop("     improper interface layout")

  draw_proc := FillCircle

  state := &null

  density := VGetState(vidgets["density"])
  delayval := VGetState(vidgets["speed"])
  scale := VGetState(vidgets["scale"])
  VSetState(vidgets["shape"], "discs")

  half := size / 2

  pane := Clone("bg=black", "dx=" || (vidgets["region"].ux + half),
    "dy=" || (vidgets["region"].uy + half), drawop=reverse")
  Clip(pane, –half, –half, size, size)

  EvInit(args) | ExitNotice("Cannot load SP.")

  variable("write", &eventsource) := –1
  variable("writes", &eventsource) := –1

  color := colormap()

  return

end

procedure kaleidoscope()

 repeat {
    EraseArea(pane, –half, –half, size, size)
    draw_list := []
    reset := &null
    repeat {
      while (  Pending() > 0) | \state do {
        ProcessEvent(root, , shortcuts)
        if \reset then break break next
        }
      putcircle()
      WDelay(delayval)
      if   draw_list > (4    density) then clrcircle()
      }
    }
end

procedure putcircle()
  local off1, off2, radius, fg

  EvGet(AllocMask) | ExitNotice("SP terminated.")

  fg := color[&eventcode]
  radius := sqrt(&eventvalue    scale)

  off1 := ?size % half
  off2 := ?size % half

  put(draw_list, off1, off2, radius, fg)

  outcircle(off1, off2, radius, fg)

  return

end

procedure clrcircle()

  outcircle(
    get(draw_list),
    get(draw_list),
    get(draw_list),
    get(draw_list)
    )

  return

end

procedure outcircle(off1, off2, radius, color)

  Fg(pane, color)

  draw_proc(pane, off1, off2, radius)
  draw_proc(pane, off1, –off2, radius)
  draw_proc(pane, –off1, off2, radius)
  draw_proc(pane, –off1,–off2, radius)
  draw_proc(pane, off2, off1, radius)
  draw_proc(pane, off2, –off1, radius)
  draw_proc(pane, –off2, off1, radius)
  draw_proc(pane, –off2,–off1, radius)

  return

end

procedure density_cb(vidget, value)

  density := value

  reset := 1

end

procedure speed_cb(vidget, value)

  delayval := value
```

```
  return
end

procedure file_cb(vidget, value)

  case value[1] of {
    "snapshot   @S":   snapshot(pane, –half, –half, size, size)
    "quit          @Q":   exit()
    }

  return

end

procedure scale_cb(vidget, value)

  scale := value

  return

end

procedure pause_cb(vidget, value)

  state := value

  return

end

procedure reset_cb(vidget, value)

  reset := 1

  return

end

procedure shape_cb(vidget, value)

  draw_proc := case value of {
   "discs":   FillCircle
   "rings":   DrawCircle
   }

  reset := 1

  return

end

procedure shortcuts(e)

  if &meta then
    case map(e) of {
      "q":exit()
      "s": snapshot(pane, –half, –half, size, size)
      }

  return

end

#===<<vib:begin>>===        modify using vib
procedure ui_atts()
  return ["size=600,455", "bg=gray–white", "label=kaleido"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
  [":Sizer:::0,0,600,455:kaleido",],
  ["density:Slider:h:1:41,171,100,15:10,100,50",density_cb],
  ["file:Menu:pull::12,3,36,21:File",file_cb,
    ["snapshot @S","quit     @Q"]],
  ["label07:Label:::7,120,28,13:slow",],
  ["label08:Label:::151,120,28,13:fast",],
  ["label10:Label:::64,270,7,13:1",],
  ["label11:Label:::124,270,7,13:5",],
  ["label12:Label:::47,200,14,13:10",],
  ["label13:Label:::116,200,21,13:100",],
  ["label14:Label:::78,200,14,13:50",],
  ["label9:Label:::43,270,14,13:.2",],
  ["lbl_density:Label:::67,151,49,13:density",],
  ["lbl_scale:Label:::74,220,35,13:scale",],
  ["lbl_speed:Label:::74,100,35,13:speed",],
  ["line:Line:::0,30,600,30:",],
  ["line1:Line:::68,256,68,266:",],
  ["line2:Line:::128,256,128,266:",],
```


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–project@cs.arizona.edu

and

Bright Forest Publishers

Tucson Arizona

© 1997 by Ralph E. Griswold, Madge T. Griswold, and Gregg M. Townsend

All rights reserved.

```
  ["line3:Line:::54,256,54,266:",],
  ["line4:Line:::128,186,128,196:",],
  ["line5:Line:::55,186,55,196:",],
  ["line6:Line:::86,186,86,196:",],
  ["pause:Button:regular:1:33,55,45,20:pause",pause_cb],
  ["reset:Button:regular::111,55,45,20:reset",reset_cb],
  ["scale:Slider:h:1:42,240,100,15:0.1,5,1",scale_cb],
  ["shape:Choice::2:64,330,64,42:",shape_cb, ["discs","rings"]],
  ["speed:Slider:h:1:41,121,100,15:100,0,0",speed_cb],
  ["region:Rect:raised::187,42,400,400:",],
  )
end
#===<<vib:end>>===        end of section maintained by vib
```

References

1. "Dynamic Analysis of Icon Programs", *Icon Analyst* 28, pp. 9-11.

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

3. "Dynamic Analysis", *Icon Analyst* 30, pp. 9-11.

4. "Dynamic Analysis", *Icon Analyst* 33, pp. 3-6.

5. "Dynamic Analysis", *Icon Analyst* 37, pp. 3-9.

6. "Visualizing Concatenation", *Icon Analyst* 39, pp. 6-8.

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

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

9. "Monitoring Icon Programs", *Icon Analyst* 15, pp. 6-10.

10. "Memory Monitoring", *Icon Analyst* 2, pp. 5-9.

11. "Program Visualization", *Icon Analyst* 16, pp. 1-8.

12. *Left Brain, Right Brain*, Sally P. Springer and Georg Deutsch, W. H. Freeman and Co., New York, 1985.

13. *Drawing on the Right Side of the Brain*, Betty Edwards, Jeremy P. Tarcher, Inc., Los Angles, 1989.

14. "A Framework for Monitoring", *Icon Analyst* 39, pp. 1-5.

15. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986.

16. *Supplementary Information for the Implementation of Version 8 of Icon*, Ralph E. Griswold, Icon Project Document 112, Department of Computer Science, The University of Arizona, 1990.

Factors of Versum Numbers

This is the eleventh article on versum numbers [1-10]. Since the last article was in February, it may help you get back on track to recall that a versum number results from the addition of a number and its digit reversal. For example, for 196, 196 + 691 = 887. A versum sequence results from starting with a *seed* and continuing the reverse-addition process. Thus, the seed 196 produces the sequence 887, 1675, 7436, 13783, 52514, ... A predecessor of a versum number is a number whose reverse sum is the versum number. For example, 887 is a predecessor of 1675.

For this article, you'll also need to recall that all palindromic numbers are versum numbers unless they have an odd number of digits and an odd middle digit.

In the last article, we started to explore factors of versum numbers, concentrating on factors of 11, since the reverse sum of a number with an even number of digits is divisible by 11, and all subsequent reverse sums also are divisible by 11.

Powers

The logical place to start when looking for interesting things about factors of versum numbers is among numbers that have a well-defined factor structure, such as powers. In the last article we showed that 11^n is a versum number for 1 n 5, but beyond 5 we didn't find any.

In this article, we'll look at versum numbers that are of the form i^n, ($n > 1$). There aren't many of them. For all integers through 12 digits, there are only 180:

n	versum powers
2	160
3	14
4	4
5	2
6 and up	0

The count of squares does not include the 4th powers.

All of the versum powers in this range are listed below. The symbols in the left margin refer to the power and those at the right to its root. The symbol identifies versum numbers (we've left these out of the left column, since all the powers are versum). The symbol identifies a palindrome. The symbol identifies a prime root. We'll deal with primes in a subsequent article.

Power		Factorization		Root
4			=	2^2
16	=	2^4	=	4^2
121			=	11^2
484	=	$2^2 \times 11^2$	=	22^2
625	=	5^4	=	25^2
1089	=	$3^2 \times 11^2$	=	33^2
10201	=	101^2	=	101^2
14641	=	11^4	=	121^2
19881	=	$3^2 \times 47^2$	=	141^2
40804	=	$2^2 \times 101^2$	=	202^2
49284	=	$2^2 \times 3^2 \times 37^2$	=	222^2
69696	=	$2^6 \times 3^2 \times 11^2$	=	264^2
91809	=	$3^2 \times 101^2$	=	303^2
94249			=	307^2
203401	=	$11^2 \times 41^2$	=	451^2
698896	=	$2^4 \times 11^2 \times 19^2$	=	836^2
1002001	=	$7^2 \times 11^2 \times 13^2$	=	1001^2
1234321	=	$11^2 \times 101^2$	=	1111^2
1490841	=	$3^2 \times 11^2 \times 37^2$	=	1221^2
1517824	=	$2^8 \times 7^2 \times 11^2$	=	1232^2
4008004	=	$2^2 \times 7^2 \times 11^2 \times 13^2$	=	2002^2
4276624	=	$2^4 \times 11^2 \times 47^2$	=	2068^2
4460544	=	$2^{12} \times 3^2 \times 11^2$	=	2112^2
4937284	=	$2^2 \times 11^2 \times 101^2$	=	2222^2
5313025	=	$5^2 \times 461^2$	=	2305^2
6325225	=	$5^2 \times 503^2$	=	2515^2
6895876	=	$2^2 \times 13^2 \times 101^2$	=	2626^2
6948496	=	$2^4 \times 659^2$	=	2636^2
7706176	=	$2^6 \times 347^2$	=	2776^2
9018009	=	$3^2 \times 7^2 \times 11^2 \times 13^2$	=	3003^2
15665764	=	$2^2 \times 1979^2$	=	3958^2
15776784	=	$2^4 \times 3^2 \times 331^2$	=	3972^2
16120225	=	$5^2 \times 11^2 \times 73^2$	=	4015^2
16654561	=	$7^2 \times 11^2 \times 53^2$	=	4081^2
63600625	=	$5^4 \times 11^2 \times 29^2$	=	7975^2
66977856	=	$2^6 \times 3^2 \times 11^2 \times 31^2$	=	8184^2
95199049	=	$11^2 \times 887^2$	=	9757^2
100020001	=	$73^2 \times 137^2$	=	10001^2
104060401	=	101^4	=	10201^2
108180801	=	$3^2 \times 3467^2$	=	10401^2
121242121	=	$7^2 \times 11^4 \times 13^2$	=	11011^2
123676641	=	$3^2 \times 11^2 \times 337^2$	=	11121^2
125686521	=	$3^2 \times 37^2 \times 101^2$	=	11211^2
144288144	=	$2^4 \times 3^2 \times 7^2 \times 11^2 \times 13^2$	=	12012^2
144504441	=	$3^2 \times 4007^2$	=	12021^2
149352841	=	$11^4 \times 101^2$	=	12221^2
170198116	=	$2^2 \times 11^2 \times 593^2$	=	13046^2
274929561	=	$3^2 \times 5527^2$	=	16581^2
400080004	=	$2^2 \times 73^2 \times 137^2$	=	20002^2
408120804	=	$2^2 \times 3^2\ 7^2 \times 13^2 \times 37^2$	=	20202^2
441504144	=	$2^4 \times 3^2 \times 17^2 \times 103^2$	=	21012^2
449948944	=	$2^4 \times 5303^2$	=	21212^2
484968484	=	$2^2 \times 7^2 \times 11^4 \times 13^2$	=	22022^2
522808225	=	$5^2 \times 17^2 \times 269^2$	=	22865^2
562069264	=	$2^4 \times 5927^2$	=	23708^2
582160384	=	$2^{12} \times 13^2 \times 29^2$	=	24128^2
617323716	=	$2^2 \times 3^2 \times 41^2 \times 101^2$	=	24846^2
631768225	=	$5^2 \times 11^2 \times 457^2$	=	25135^2
677352676	=	$2^2 \times 7^2 \times 11^2 \times 13^4$	=	26026^2
715776516	=	$2^2 \times 3^2 \times 7^6 \times 13^2$	=	26754^2
900180009	=	$3^2 \times 73^2 \times 137^2$	=	30003^2
942060249	=	$3^2 \times 13^2 \times 787^2$	=	30693^2
1046069649	=	$3^2 \times 10781^2$	=	32343^2
1101576100	=	$2^2 \times 5^2 \times 3319^2$	=	33190^2
1133736241	=	$11^2 \times 3061^2$	=	33671^2
1183979281	=	$19^2 \times 1811^2$	=	34409^2
1254505561			=	35419^2
1271564281	=	$13^4 \times 211^2$	=	35659^2
1273847481	=	$3^2 \times 11897^2$	=	35691^2
1439291844	=	$2^2 \times 3^2 \times 6323^2$	=	37938^2
1522404324	=	$2^2 \times 3^2 \times 7^2 \times 929^2$	=	39018^2
1615718416	=	$2^4 \times 13^2 \times 773^2$	=	40196^2
2548129441	=	$11^2 \times 13^2 \times 353^2$	=	50479^2
4045214404	=	$2^2 \times 7^4 \times 11^2 2 \times 59^2$	=	63602^2
5581882944	=	$2^6 \times 3^2 \times 11^2 \times 283^2$	=	74712^2
9504885049	=	$11^2 \times 8863^2$	=	97493^2
10000200001	=	$11^2 \times 9091^2$	=	100001^2
10221412201	=	$7^2 \times 11^2 \times 13^2 \times 101^2$	=	101101^2
10445044401	=	$3^2 \times 11^2 \times 19^2 \times 163^2$	=	102201^2
10774647601			=	103801^2
10989328900	=	$2^2 \times 5^2 \times 11^2 \times 953^2$	=	104830^2
11081772900	=	$2^2 \times 3^2 \times 5^2 \times 11^4 \times 29^2$	=	105270^2
12078229801	=	$11^2 \times 97^2 \times 103^2$	=	109901^2
12102420121	=	$11^2 \times 73^2 \times 137^2$	=	110011^2
12345654321	=	$3^2 \times 7^2 \times 11^2 \times 13^2 \times 37^2$	=	111111^2
12370110841	=	$11^2 \times 10111^2$	=	111221^2
12591308521	=	$11^2 \times 101^4$	=	112211^2
14405040441	=	$3^2 \times 11^2 \times 3637^2$	=	120021^2
14670296641	=	$7^2 \times 11^6 \times 13^2$	=	121121^2
14937972841	=	$11^2 \times 41^2 \times 271^2$	=	122221^2
15648008464	=	$2^4 \times 11^2 \times 2843^2$	=	125092^2
15657266641	=	$157^2 \times 797^2$	=	125129^2
17803831761	=	$3^2 \times 79^2 \times 563^2$	=	133431^2
19115551081	=	$11^2 \times 12569^2$	=	138259^2
22048983121	=	$11^2 \times 13499^2$	=	148489^2
26120701161	=	$3^2 \times 17^2 \times 3169^2$	=	161619^2
27313842361	=	$13^2 \times 12713^2$	=	165269^2
29569897681	=	$61^2 \times 2819^2$	=	171959^2
40000800004	=	$2^2 \times 11^2 \times 9091^2$	=	200002^2
40279687204	=	$2^2 \times 23^2 \times 4363^2$	=	200698^2
40442014404	=	$2^2 \times 3^2 \times 11^4 \times 277^2$	=	201102^2
40885648804	=	$2^2 \times 7^2 \times 11^2 \times 13^2 \times 101^2$	=	202202^2
44105040144	=	$2^4 \times 3^2 \times 11^2 \times 37^2 \times 43^2$	=	210012^2
44568276544	=	$2^6 \times 11^2 \times 2399^2$	=	211112^2
45033932944	=	$2^4 \times 7^2 \times 11^2 \times 13^2 \times 53^2$	=	212212^2
46551514564	=	$2^2 \times 233^2 \times 463^2$	=	215758^2
48409680484	=	$2^2 \times 11^2 \times 73^2 \times 137^2$	=	220022^2
48894938884	=	$2^2 \times 11^2 \times 19^2 \times 23^4$	=	221122^2
49496460484	=	$2^2 \times 173^2 \times 643^2$	=	222478^2
49582819584	=	$2^8 \times 3^2 \times 4639^2$	=	222672^2
50715940804	=	$2^2 \times 112601^2$	=	225202^2
51280508304	=	$2^4 \times 3^2 \times 113^2 \times 167^2$	=	226452^2
53261716225	=	$5^2 \times 101^2 \times 457^2$	=	230785^2

53785958724	$= 2^2\times3^2\times38653^2$	$= 231918^2$
57916272964	$= 2^2\times11^2\times10939^2$	$= 240658^2$
59913331984	$= 2^4\times11^2\times5563^2$	$= 244772^2$
62562515625	$= 3^2\times5^6\times23^2\times29^2$	$= 250125^2$
62998490025	$= 3^2\times5^2\times29^2\times577^2$	$= 250995^2$
63259795225	$= 5^2\times11^2\times17^2\times269^2$	$= 251515^2$
63541805625	$= 3^2\times5^4\times3361^2$	$= 252075^2$
63592230625	$= 5^4\times7^2\times11^2\times131^2$	$= 252175^2$
67561525476	$= 2^2\times3^2\times43321^2$	$= 259926^2$
67613520676	$= 2^2\times13^2\times73^2\times137^2$	$= 260026^2$
68631424576	$= 2^6\times11^2\times13^2\times229^2$	$= 261976^2$
70630503696	$= 2^4\times3^2\times22147^2$	$= 265764^2$
74696169636	$= 2^2\times3^2\times11^2\times41^2\times101^2$	$= 273306^2$
90001800009	$= 3^2\times11^2\times9091^2$	$= 300003^2$
91447574409	$= 3^2\times100801^2$	$= 302403^2$
101507871609	$= 3^2\times61^2\times1741^2$	$= 318603^2$
109845844900	$= 2^2\times5^2\times11^2\times23^2\times131^2$	$= 331430^2$
110908980900	$= 2^2\times3^2\times5^2\times17^2\times653^2$	$= 333030^2$
116506851561	$= 3^2\times113777^2$	$= 341331^2$
117243923281	$= 43^2\times7963^2$	$= 342409^2$
120861217801		$= 347651^2$
122963136921	$= 3^2\times179^2\times653^2$	$= 350661^2$
126216062361	$= 3^2\times118423^2$	$= 355269^2$
148164946084	$= 2^2\times192461^2$	$= 384922^2$
149166660841	$= 11^2\times35111^2$	$= 386221^2$
152244334225	$= 5^2\times73^2\times1069^2$	$= 390185^2$
152306770225	$= 5^2\times89^2\times877^2$	$= 390265^2$
153712611844	$= 2^2\times11^2\times71^2\times251^2$	$= 392062^2$
155281707364	$= 2^2\times7^2\times4021^2$	$= 394058^2$
156255765264	$= 2^4\times3^2\times32941^2$	$= 395292^2$
158421512484	$= 2^2\times3^2\times66337^2$	$= 398022^2$
161174146225	$= 5^2\times23^2\times3491^2$	$= 401465^2$
170985558016	$= 2^{12}\times7^2\times13^2\times71^2$	$= 413504^2$
172469106436	$= 2^2\times11^2\times43^2\times439^2$	$= 415294^2$
173810612836	$= 2^2\times7^2\times97^2\times307^2$	$= 416906^2$
176968296976	$= 2^4\times251^2\times419^2$	$= 420676^2$
226872168721	$= 11^2\times19^2\times{}^2\times53^2$	$= 476311^2$
251172371241	$= 3^2\times11^2\times15187^2$	$= 501171^2$
257585685841	$= 11^2\times29^2\times37^2\times43^2$	$= 507529^2$
269328822961	$= 11^4\times4289^2$	$= 518969^2$
282057650281	$= 11^2\times48281^2$	$= 531091^2$
526676775625	$= 5^4\times7^2\times11^2\times13^2\times29^2$	$= 725725^2$
595477675584	$= 2^6\times3^2\times11^2\times37^2\times79^2$	$= 771672^2$
596258863684	$= 2^2\times11^2\times35099^2$	$= 772178^2$
637832238736	$= 2^4\times7^2\times11^2\times2593^2$	$= 798644^2$
653778976356	$= 2^2\times3^2\times11^2\times12251^2$	$= 808566^2$
709529936896	$= 2^{10}\times11^2\times2393^2$	$= 842336^2$
718079980816	$= 2^4\times11^2\times19259^2$	$= 847396^2$
728370074916	$= 2^2\times3^2\times11^2\times67^2\times193^2$	$= 853446^2$
968525634769	$= 7^2\times11^2\times12781^2$	$= 984137^2$
987803417689	$= 11^2\times90353^2$	$= 993883^2$

8		$= 2^3$
343		$= 7^3$
1331		$= 11^3$
166375	$= 5^3\times11^3$	$= 55^3$
1030301		$= 101^3$
217081801		$= 601^3$
353393243	$= 7^3\times101^3$	$= 707^3$
938313739	$= 11^3\times89^3$	$= 979^3$
1003003001	$= 7^3\times11^3\times13^3$	$= 1001^3$
1058089859		$= 1019^3$
1371330631	$= 11^3\times101^3$	$= 1111^3$
12503322161	$= 11^3\times211^3$	$= 2321^3$
117001919971	$= 67^3\times73^3$	$= 4891^3$
344030029343	$= 49^3\times11^3 13^3$	$= 7007^3$

16		$= 2^4$
625		$= 5^4$
14641		$= 11^4$
104060401		$= 101^4$

161051		$= 11^5$
10510100501		$= 101^5$

It is surprising to us that so many of the versum roots also are palindromes — only four are not.

It's well known that palindromic powers often have palindromic roots. In fact, the only known palindromic cube that does not have a palindromic cube root is $10662526601 = 2201^3$ [11]. Neither the power or its root are versum numbers. Furthermore, there is no known palindrome that is an nth power, $n > 3$, whose nth root is not a palindrome.

From the list given at the beginning of this article, it's clear that many non-palindromic versum powers have palindromic versum roots as well. Is there something deeper underlying these relationships?

Here's a listing of just the versum powers that have palindromic versum roots with the palindromic powers identified for the range we've covered. Of the 60, 25 are palindromic and 35 not:

4		$= 2^2$
16	$= 2^4$	$= 4^2$
121		$= 11^2$
484	$= 2^2\times11^2$	$= 22^2$
1089	$= 3^2\times11^2$	$= 33^2$
10201	$= 101^2$	$= 101^2$
14641	$= 11^4$	$= 121^2$
19881	$= 3^2\times47^2$	$= 141^2$
40804	$= 2^2\times101^2$	$= 202^2$
49284	$= 2^2\times3^2\times37^2$	$= 222^2$
91809	$= 3^2\times101^2$	$= 303^2$
1002001	$= 7^2\times11^2\times13^2$	$= 1001^2$
1234321	$= 11^2\times101^2$	$= 1111^2$
1490841	$= 3^2\times11^2\times37^2$	$= 1221^2$
4008004	$= 2^2\times7^2\times11^2\times13^2$	$= 2002^2$
4460544	$= 2^{12}\times3^2\times11^2$	$= 2112^2$
4937284	$= 2^2\times11^2\times101^2$	$= 2222^2$
9018009	$= 3^2\times7^2\times11^2\times13^2$	$= 3003^2$
100020001	$= 73^2\times137^2$	$= 10001^2$
104060401	$= 101^4$	$= 10201^2$
108180801	$= 3^2\times3467^2$	$= 10401^2$
121242121	$= 7^2\times11^4\times13^2$	$= 11011^2$
125686521	$= 3^2\times37^2\times101^2$	$= 11211^2$
144504441	$= 3^2\times4007^2$	$= 12021^2$
149352841	$= 11^4\times101^2$	$= 12221^2$
400080004	$= 2^2\times73^2\times137^2$	$= 20002^2$
408120804	$= 2^2\times3^2\ 7^2\times13^2\times37^2$	$= 20202^2$

441504144	$= 2^4 \times 3^2 \times 17^2 \times 103^2$	$= 21012^2$
449948944	$= 2^4 \times 5303^2$	$= 21212^2$
484968484	$= 2^2 \times 7^2 \times 11^4 \times 13^2$	$= 22022^2$
900180009	$= 3^2 \times 73^2 \times 137^2$	$= 30003^2$
10000200001	$= 11^2 \times 9091^2$	$= 100001^2$
10221412201	$= 7^2 \times 11^2 \times 13^2 \times 101^2$	$= 101101^2$
10445044401	$= 3^2 \times 11^2 \times 19^2 \times 163^2$	$= 102201^2$
12078229801	$= 11^2 \times 97^2 \times 103^2$	$= 109901^2$
12102420121	$= 11^2 \times 73^2 \times 137^2$	$= 110011^2$
12345654321	$= 3^2 \times 7^2 \times 11^2 \times 13^2 \times 37^2$	$= 111111^2$
12591308521	$= 11^2 \times 101^4$	$= 112211^2$
14405040441	$= 3^2 \times 11^2 \times 3637^2$	$= 120021^2$
14670296641	$= 7^2 \times 11^6 \times 13^2$	$= 121121^2$
14937972841	$= 11^2 \times 41^2 \times 271^2$	$= 122221^2$
40000800004	$= 2^2 \times 11^2 \times 9091^2$	$= 200002^2$
40442014404	$= 2^2 \times 3^2 \times 11^4 \times 277^2$	$= 201102^2$
40885648804	$= 2^2 \times 7^2 \times 11^2 \times 13^2 \times 101^2$	$= 202202^2$
44105040144	$= 2^4 \times 3^2 \times 11^2 \times 37^2 \times 43^2$	$= 210012^2$
44568276544	$= 2^6 \times 11^2 \times 2399^2$	$= 211112^2$
45033932944	$= 2^4 \times 7^2 \times 11^2 \times 13^2 \times 53^2$	$= 212212^2$
48409680484	$= 2^2 \times 11^2 \times 73^2 \times 137^2$	$= 220022^2$
48894938884	$= 2^2 \times 11^2 \times 19^2 \times 23^4$	$= 221122^2$
90001800009	$= 3^2 \times 11^2 \times 9091^2$	$= 300003^2$
8		$= 2^3$
1331		$= 11^3$
1030301		$= 101^3$
353393243	$= 7^3 \times 101^3$	$= 707^3$
1003003001	$= 7^3 \times 11^3 \times 13^3$	$= 1001^3$
1371330631	$= 11^3 \times 101^3$	$= 1111^3$
344030029343	$= 49^3 \times 11^3 13^3$	$= 7007^3$
16		$= 2^4$
14641		$= 11^4$
104060401		$= 101^4$
161051		$= 11^5$
10510100501		$= 101^5$

Numbers of the Form $\underline{1}_n$

Numbers of the form 1, 11, 111, 1111, … are called *repunit numbers* (for repeated units) and have special properties. In the notation we've used before, they are given by $\underline{1}_n$.

Since these numbers are palindomes composed entirely of odd digits, numbers of the form $\underline{1}_{2n}$, $n > 0$ are versum numbers, while numbers of the form $\underline{1}_{2n+1}$, n 0 are not. Note that in the roots of versum powers in the range we've examined, all numbers of the form $\underline{1}_{2n}$, $n > 0$ are included but none of the numbers of the form $\underline{1}_{2n+1}$, n 0 are.

As numbers of the form $\underline{1}_{2n}$ are raised to successively higher powers, the power at which the result ceases to be a palindrome or a versum number drops off. As far as we have been able to determine, if $(\underline{1}_{2n})^m$ is not a palindrome, $(\underline{1}_{2n})^k$ is not a palindrome for $k > m$, and similarly for versum numbers. As the power increases, carries disrupt patterns. The mathematically correct term for this is that "things get messed up".

The following table shows what we've found for n from 1 through 32. The notations m and m stand for the highest powers for which the result is a palindrome and a versum number, respectively. The question marks indicate incomplete information. For example, 2? means that the square meets the criterion but it is not known if the cube does. The range of results is limited by our inability to determine if very large numbers are versum. Palindromes present no problems.

n	m	m
1	4	5
2	2	3
3	2	2
4	2	2
5	1	2
6	1	2
7	1	2?
8	1	2?
9-32	1	?

Largest Palindromic and Versum Powers of $\underline{1}_{2n}$

Of course, it's possible that palindromes and versum numbers occur for higher powers. It just seems unlikely.

Numbers of the Form 1 $\underline{0}_n$ 1

If you look closely at the palindromic versum roots in the listings above, you'll see that many are of the form $(1\,\underline{0}_n\,1)^m$, m 0. Thus for n starting at 0, these numbers have the form 11^m, 101^m, 1001^m, 10001^m, and so on.

For m 0, we can give formulas for some of the mth powers:

m	formula	range
0	1	n 0
1	$1\,\underline{0}_n\,1$	n 0
2	$1\,\underline{0}_n\,2\,\underline{0}_n\,1$	n 0
3	$1\,\underline{0}_n\,3\,\underline{0}_n\,3\,\underline{0}_n\,1$	n 0

Icon on the Web

Information about Icon is available on the World Wide Web at

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

4	$1\,\underline{0}_n\,4\,\underline{0}_n\,6\,\underline{0}_n\,4\,\underline{0}_n\,1$	$n \quad 0$
5	$1\,\underline{0}_{n-1}\,5\,\underline{0}_{n-1}\,1\,\underline{0}_n\,1\,\underline{0}_{n+1}\,5\,\underline{0}_n\,1$	$n > 0$
6	$1\,\underline{0}_{n-1}\,6\,\underline{0}_n\,15\,\underline{0}_{n-1}\,2\underline{0}_n\,15\,\underline{0}_n\,6\,\underline{0}_n\,1$	$n > 0$
7	$1\,\underline{0}_n\,7\,\underline{0}_{n-1}\,21\,\underline{0}_{n-1}\,\underline{0}_n\,35\,\underline{0}_{n-1}\,35\,\underline{0}_{n-1}\,21\,\underline{0}_n\,7\,\underline{0}_n\,1$	$n > 0$

Do the patterns suggest anything? Does it help if we change the form for $m = 5$ to give

m	
0	1
1	$1\,\underline{0}_n\,1$
2	$1\,\underline{0}_n\,2\,\underline{0}_n\,1$
3	$1\,\underline{0}_n\,3\,\underline{0}_n\,3\,\underline{0}_n\,1$
4	$1\,\underline{0}_n\,4\,\underline{0}_n\,6\,\underline{0}_n\,4\,\underline{0}_n\,1$
5	$1\,\underline{0}_{n-1}\,5\,\underline{0}_{n-1}\,10\,\underline{0}_{n-1}\,10\,\underline{0}_n\,5\,\underline{0}_n\,1$
6	$1\,\underline{0}_{n-1}\,6\,\underline{0}_n\,15\,\underline{0}_{n-1}\,2\underline{0}_n\,15\,\underline{0}_n\,6\,\underline{0}_n\,1$
7	$1\,\underline{0}_n\,7\,\underline{0}_{n-1}\,21\,\underline{0}_{n-1}\,\underline{0}_n\,35\,\underline{0}_{n-1}\,35\,\underline{0}_{n-1}\,21\,\underline{0}_n\,7\,\underline{0}_n\,1$

How about removing the 0 repeats and putting what remains in columns?

0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Ah! Pascal's Triangle; binomial coefficients. Of course! The coefficients of the terms in $(a + b)^m$.

We'll come back to digit patterns in a later article. For now, we'll conclude with some observations about versum number in $(1\,\underline{0}_n\,1)^m$, $m \quad 0$, $n \quad 0$.

We included $m = 0$ to fill out the triangle. Since $(1\underline{0}_n1)^0 = 1$, this case is uninteresting.

Numbers of the form $(1\underline{0}_n1)^2$ are versum palindromes with middle digit 2.

Numbers of the form $(1\underline{0}_n1)^3$ are versum palindromes with middle digit 0.

Numbers of the form $(1\underline{0}_n1)^4$ are versum palindromes with middle digit 6.

Numbers of the form $(1\underline{0}_n1)^5$ are non-palindromic versum numbers. The fact that they are versum numbers can be shown by constructing explicit predecessors for numbers of this form.

Numbers of $(1\underline{0}_n1)^6$ are non-palindromic and non-versum. The fact that they are non-versum can be shown by determining that numbers of this form have no predecessors.

Numbers of the form $(1\underline{0}_n1)^m$, $m > 6$ most probably also are non-palindromic and non-versum. Tests of all numbers in the range $6 \quad m \quad 20$ with $0 \quad n \quad 50$ support this conjecture.

So what do we have? Numbers of the form $(1\underline{0}_n1)^m$, $1 \quad m \quad 5$, $n \quad 0$ are versum; all others most likely are not.

Testing for Versumness

You may recall that our test for versumness consists of searching for predecessors. This is fast and easy for palindromes and many numbers also can be rejected outright because of their first and last digits. For other numbers, however, the computational complexity of the method limits testing for versumness to numbers with 25 digits or less.

When working on numbers of the form $(1\underline{0}_n1)^m$, we can take advantage of their special structure: Their heads and tails are reversals:

$x\,y\,\underline{x}$

where $\underline{x}$ here denotes the reverse of x.

It's trivial to produce a predecessor for the heads and tails of such numbers:

$x\,y'\,\underline{\mathbf{0}}_n$

where n is the length of x and y' is a predecessor of y.

It only remains to determine y', which is significantly shorter than the original number: for numbers of the form $(1\underline{0}_n1)^m$ the shortening is on the order of $4n$.

Here's the procedure we used:

```
link vpred

procedure isversum(s)
  local bound, strip1, strip2, p, middle
  local pred, midpred, head

  if s[1] ~== s[–1] then return vpred(s)    # can't handle
  bound :=  s / 2 + 1

  if s == reverse(s) then {           # handle palindromes
    if   s % 2 = 0 then
      return s[1:bound] || repl("0", bound – 1)
    else if s[bound] % 2 = 0 then
      return s[1:bound] || s[i] / 2 || repl("0", bound – 1)
    else fail
    }

  strip1 := s[1:bound]
  strip2 := reverse(s)[1:bound]

  strip1 ? {                    # look for common substrings
```

```
  every head := =strip2[1:(  strip2 + 1) to 1 by –1] do {
    p :=   head + 1
    middle := s[p:–p + 1]
    if   middle = 0 then return head || repl("0", *head)
    if middle[1] == "0" then
      midpred := vpred_(middle) | next
    else midpred :=
      right(vpred(middle),   middle, "0") | next
    pred := strip1[1:p] || midpred || repl("0", p – 1)
    if s == pred + reverse(pred) then
      return pred else next
    }
  }

  return vpred(s)        # if all else fails

end
```

Palindromes are handled separately for efficiency.

The strings strip1 and strip2 contain the first half and reversal of the last half of s, respectively. If the length of s is odd, there is a middle character that does not figure in the computation of the longest initial substring, head, of strip1 and strip2.

The computation of the longest common initial substring is one of the few in which every-do can be used to advantage in string scanning. The idea is to look in strip1 for successively shorter portions of strip2 until a match is found, which then is assigned to head.

Unfortunately, the longest common initial string does not always produce a middle portion that has a predecessor. This situation is identified when no value for midpred emerges and next takes the computation back to the beginning of the every-do loop.

The computation of the predecessor for the middle is muddled by the fact that vpred() does not handle strings with an initial 0; vpred_() is needed for this. To make matters even messier, vpred_() may produce "false positives" for predecessors, which requires the proposed predecessor to be tested. If it passes the test, it is returned (isversum() makes no attempt to produce a possible second predecessor). If the test fails, control is returned to the beginning of the every-do loop to try a shorter common initial substring. If no common substring works, vpred() is called as a last resort.

Working on this procedure and dealing with the problems encountered with vpred() reminded us that vpred() needs to be reworked. And perhaps the code in isversum() should be integrated into vpred().

Incidentally there are many ways of determining the longest common initial substring of two strings. Steve Wampler recently posed this problem to the Icon users's group. We plan to have an article on different approaches in the next Analyst.

References

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

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

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

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

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

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

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

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

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

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

11. *The New Ambidextrous Universe*, Martin Gardner, W. H. Freeman and Co., New York, 1990.

Next Time

We haven't finished with versum factors. In the next article on versum numbers, we'll look at the versum factors of other classes of numbers that have well-defined factor structures. To end the investigation of factors, we'll look at versum factors of all numbers. Beyond that, it's primes.

Debugging: Library Support

As mentioned in earlier articles on debugging [1-3], one of the main problems with Icon's built-in debugging facilities is voluminous output — to the point where it is difficult or impractical to find relevant information.

This problem can be reduced by programs in the Icon program library that filter debugging output and extract specified information. This articles describes these programs. Before getting to the programs, however, we need to say something about getting diagnostic output in a form it can be processed.

Capturing Diagnostic Output

Diagnostic output produced by Icon is written to standard error output. In many environments, standard output (such as normal program output) and standard error output both are written to the console by default, causing them to be intermixed.

Even if the output streams are not intermixed, debugging output that appears on a console is of limited use. Instead, it needs to be saved in a file where it can be examined or piped into another program.

Ways to deal with this depend on the platform and its environment. In MS-DOS, the command-line option –e allows the specification of a file to which standard error output is written. For example,

```
iconx –e prog.err prog
```

runs progs and saves standard error output in prog.err. Unfortunately, this only works when iconx is used to run a .icx file produced by the –I option to icont, as in

```
icont –I prog
```

which produces prog.icx as opposed to

```
icont prog
```

which produces an executable file prog.exe. For debugging, though, switching to .icx files usually is worth the effort.

In UNIX, separating standard output and standard error output depends on the shell you use. It's easy using the Bourne shell, since file descriptor 2, for standard error output, can be redirected explicitly, as in

```
prog 2>prog.err
```

Filtering Trace Output

The program itrcfltr filters program output, discarding lines that are not the result of tracing and only showing trace messages for specified procedures.

The procedures whose trace messages are to be shown are specified on the command line. For example,

```
itrcfltr vpred_ < vpred.out
```

writes only the lines of vpred.out that contain trace messages related to the procedure vpred_(). For example, if vpred.out consists of

```
vtest.icn     :   7  | isversum("1234321")
isversum.icn  :  41  | isversum returned "1232000"
vtest.icn     :   7  | isversum("1234322")
isversum.icn  :  35  | | vpred("1234322")
vpred.icn     :  51  | | | vpred_("1234322")
vpred.icn     :  75  | | | | vpred_i1("1234322")
vpred.icn     : 157  | | | | | vpred_noinc("3431")
vpred.icn     : 192  | | | | | | vpred_("3431")
vpred.icn     :  76  | | | | | | | vpred_in("3431")
vpred.icn     : 178  | | | | | | | vpred_in failed
vpred.icn     :  80  | | | | | | vpred_ failed
vpred.icn     : 194  | | | | | vpred_noinc failed
vpred.icn     : 158  | | | | | vpred_noinc(13431)
vpred.icn     : 192  | | | | | | vpred_(13431)
                ...
```

then the output of the example above is:

```
vpred.icn     :  51  | | | vpred_("1234322")
vpred.icn     : 192  | | | | | | vpred_("3431")
vpred.icn     :  80  | | | | | | vpred_ failed
vpred.icn     : 192  | | | | | | vpred_(13431)
                ...
```

The program itrcfltr uses options() [4], which supports "response files" that can contain arguments that otherwise might appear on the command line. This can be useful if the trace messages from many different procedures are wanted. For example,

```
itrcfltr @vprocs.rsp <vtest.out
```

produces trace messages for the procedures listed in the file vprocs.rsp.

If no procedures are specified for itrcfltr, all trace messages are produced.

Note: It is, of course, possible to fool itrcfltr by deliberately producing output that looks like error messages

Summarizing Trace Output

Sometimes it is useful to know how many times different procedures are called, fail, and so on. The program itrcsum does this. It summarizes procedure activity based trace messages. For example,

```
itrcsum <vpred.out
```

produces output like this:

```
maximum recursion depth = 11
average recursion depth = 7.490

File references:

isversum.icn      3
vpred.icn       145
vtest.icn         2

procedure activity:
```

name	call	return	suspend	fail	resume
isversum	2	1	0	1	0
main	1	0	0	0	0
vpred	1	0	0	1	0
vpred_	17	0	8	17	8
vpred_1	2	2	0	0	0
vpred_2	3	0	0	3	0
vpred_3	9	0	3	9	3
vpred_i1	4	0	5	4	5
vpred_in	1	0	0	1	0
vpred_inc	6	0	2	6	2
vpred_noinc	10	0	3	10	3

Piping the output of itrcfltr into itrcsum can be used to obtain summary information for selected procedures. For example,

```
itrcfltr vpred_ <vpred.out | itrcsum
```

produces

```
maximum recursion depth = 9
average recursion depth = 7.440

File references:

vpred.icn        48

procedure activity:
```

name	call	return	suspend	fail	resume
vpred_	17	0	8	17	8

Note that the information about recursion reflects the incomplete information itrcsum gets in this case.

Abbreviating Traceback Output

The program itrbksum filters out all input except traceback and also replaces all procedure calls in the traceback except for the first and last ones by ellipses. This program primarily is useful for handling traceback output that results from excessive recursion or from programs with visual interfaces in which the traceback output is voluminous.

Typical output from itrbksum is:

```
Run-time error 301
File parse.icn; Line 9
evaluation stack overflow

  main()
      .
      .
      .
  parse("time") from line 9 in parse.icn

at level 622
```

Next Time

Other facilities in the Icon program library that are useful for debugging will be described in the next article in this series.

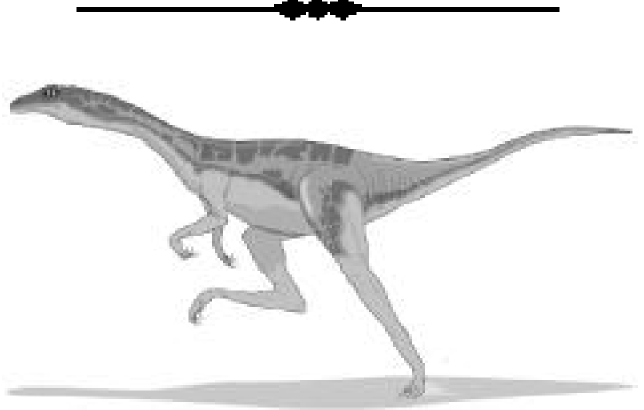
References

1. "Debugging: Error Messages" *Icon Analyst* 40, pp. 5-9.

2. "Debugging", *Icon Analyst* 41, pp. 4-7.

3. "Debugging: Tracing", *Icon Analyst* 42, pp. 4-6.

4. "From the Library", *Icon Analyst* 32, 9-10.



What's Coming Up?

We have more articles on debugging in the works. And, of course, another article on versum numbers is an ever-present threat.

We plan to change direction temporarily in the series of articles on visualizing program behavior. We're exploring three-dimensional visualization, and we expect to have something to say about that in the next issue of the *Analyst*.