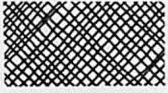
THEORY & PRACTICE



Tag—you're it

By Brian Hayes

Of course, unless one has a theory, one cannot expect much help from a computer (unless it has a theory). . . .
—Marvin Minsky

Minsky wrote these words in connection with a famous little stinker of a problem, one that has resisted both theorizing and computation for some 65 years.

The problem goes like this. Take any string of 0s and 1s at least three digits long. Examine the first digit. If it is a 0, delete the first three digits and append 00 to the end of the string; if it is a 1, delete the first three digits and add 1101 to the end. Now repeat the procedure with the resulting string, and then with the result of that transformation, and so on. The question is: what is the ultimate fate of the string? Does it grow infinitely long or dwindle away to nothing, or does it just meander forever between these extremes?

Consider the starting pattern 1000100. Since the first digit is a 1, we delete the three leftmost digits and add 1101, yielding 01001101. Now the leading digit is a 0, and so we delete three digits and append 00; the result is 0110100. In this case it is easy to see what the outcome will be: in just a few more steps the string shrinks to 000, and then it falls below the minimum length of three digits and disappears entirely.

The starting string 1001 provides an example that comes to a different end. In six applications of the rules it yields the following sequence of strings:

```
1001
   11101
      011101
         10100
            001101
               10100
                  001101
```

The last two patterns will now be repeated in an unending cycle and so the string never grows any longer than six digits or any shorter than five. A third example, which runs for 16 steps before it enters a repetitive cycle, is worked out in Figure 1.

Still more patterns could be traced in the same way. No amount of such case-by-case analysis, however, can settle the general question of what happens to all possible strings. Indeed, the case-by-case method cannot even be guaranteed to determine the fate of a single string. If the string eventually dwindles away or enters a repeating cycle, then of course the answer will be found. If the string grows without limit, however, we will never know it because the calculation will never end. What we need to solve the problem is a decision algorithm: a procedure that will decide the fate of any string and that is certain to terminate after a finite number of steps when given any valid input.

This little problem in string manipulation was first explored by Emil L. Post in 1921, when he was a doctoral candidate in mathematics at Princeton University. A colleague suggested naming the problem "tag" because the two ends of the string seemed to chase each other in a way that reminded him of the children's game. A number of other investigators (including Minsky) have worked on the problem since then, and in recent years they have brought the power of the computer to bear on it.

Nevertheless, we live in a state of almost perfect ignorance about the nature of Post's tag system. No one has solved the problem by devising a decision algorithm, but no one has proved that it cannot be solved. No one has found a string that does not either dwindle away or enter a cycle, but no one has shown that such strings do not exist.

I do not have a theory about tag systems, and neither does my computer. All the same, some interesting empirical observations can be made about the evolution of the systems. Maybe those observations will lead keener minds and machines toward the development of a theory.

The fates of strings

A tag system can have just three possible fates. First, the string can shrink to fewer than three digits and vanish. This will happen if at any point in the calculation all the digits becomes 0s. Actually, it is not even necessary for all the digits to be 0s; it is sufficient for 0s to appear at positions 1, 4, 7, 10, and so on. It is only these positions (which I shall call the key positions) that are examined in applying the tag rules, and so they alone determine the fate of a string.

The second possibility is that the system will enter a repetitive cycle. This fate is inevitable if any one string appears more than once in the evolution of the system. It is easy to see why. Each string uniquely determines its own successor; there can be no branching in the development of a tag system. Suppose string A gives rise to string B, which in turn produces C, D, and a series of further descendants; if any one of these succes-

Evolution of a tag system

Step	String	Digits
0.	10010	5
1.	101101	6
2.	1011101	7
3.	11011101	8
4.	111011101	9
5.	0111011101	10
6.	101110100	9
7.	1101001101	10
8.	10011011101	11
9.	110111011101	12
10.	1110111011101	13
11.	01110111011101	14
12.	1011101110100	13
13.	11011101001101	14
14.	111010011011101	15
15.	0100110111011101	16
16.	011011101110100	15
17.	01110111010000	14
18.	1011101000000	13
19.	11010000001101	14
20.	100000011011101	15
21.	0000110111011101	16
22.	011011101110100	15
23.	01110111010000	14
24.	1011101000000	13
25.	11010000001101	14
26.	100000011011101	15
27.	0000110111011101	16

Application of the tag rules to the starting string 10010 leads ultimately to a cycle with a period of six. The 15-digit string produced at step 16 appears again at step 22, and hence the entire cycle of six strings will be repeated indefinitely.

Figure 1.

sors generates string *A* again, then *B*, *C*, and all the rest must follow in an endless cycle.

The third possibility, of course, is that the system will go merrily on for an infinite number of steps. It is important to recognize that for this to happen, the string must grow longer without limit. The proof is straightforward. Suppose there were some initial string that evolved endlessly without dwindling away or cycling but also without ever growing longer than 100 digits. There are only 2^{100} binary strings of 100 or fewer digits, so after no more than 2^{100} steps at least one string would have to be repeated. The system then inevitably enters a cycle, which contradicts the original assumption; thus no such initial pattern exists.

In analyzing a tag system it is tempting to try a probabilistic approach. Each time a 1 comes to the head of the string, the length increases by one digit, and each time a 0 appears the length decreases by one. Thus if the digits at the key positions have a uniform, random distribution, one might expect the length to remain constant on the average.

This argument could be taken a step further. The variations in the length of the string should describe a one-dimensional random walk centered on the average length. Any one-dimensional random walk, if it is continued long enough, can be expected to visit all the sites available to it; in this case the string should at some point becomes arbitrarily short, drop below the three-digit threshold, and dwindle away. (The blind man, stumbling about at random on top of the cliff, eventually falls over the precipice.) Even before that happens, the system may light upon a pattern that leads into a cycle.

The trouble with this analysis is that the pattern of 0s and 1s is not random; on the contrary, it is generated by a fully deterministic process. Moreover, even if most of the strings have the statistical properties of random patterns, there may well be exceptional strings that behave very differently. At best a probabilistic argument can predict the typical outcome, but what is of greatest interest is the singular case.

The tag problem can be expressed as a Turing machine program. It is most conveniently formulated for a two-headed Turing machine, as shown in Figure 2. One head reads digits at the front of the string and instructs the other head to write new digits at the rear. The question is then whether the two heads ever come together or whether they grow infinitely far apart. This formulation gives a clue to why the problem is so hard. Predicting the outcome of all possible tag systems may be equivalent to the Turing machine halting problem, the granddaddy of all intractable problems in computer science.

Another way to view tag systems is in terms of formal languages. A language can be defined as a set of strings, where each string is made up of symbols drawn from a finite alphabet. Here the alphabet consists of the two symbols 0 and 1, and the tag transformation rules form a grammar defining which strings (out of all possible binary strings) are members of the language. The two grammar rules can be expressed as:

```
0XX$ → $00
1XX$ → $1101
```

In these rules *X* stands for any single symbol (that is, either 0 or 1), and *$* represents any string of symbols of arbitrary length, including the empty string. Hence the first rule states that if a string consists of a 0 followed by any two symbols and the arbitrary substring *$*, then it can be transformed into *$* followed by 00.

These rules differ in an important way from the rules commonly used to specify programming languages. The grammars of virtually all programming languages are said to be context-free, which means (among other things) that each grammar rule has only one symbol on its left side. In the tag-system grammar an arbitrarily long string of symbols appears on the left side of each rule. The grammar is a member of the most powerful and most complex class of grammars, the unrestricted recursive grammars.

How to play tag

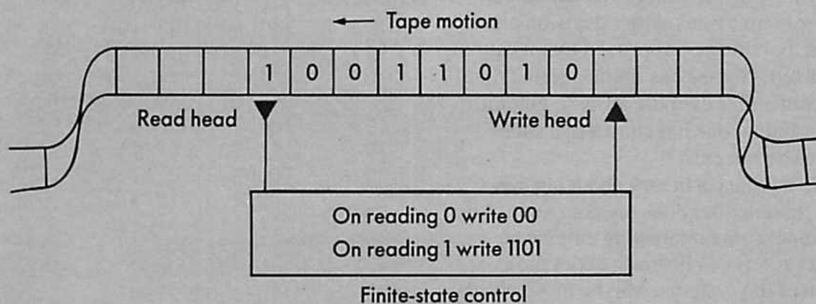
Although tracing individual tag systems cannot answer the deepest questions about the nature of the systems, it is a useful way to gather information about them. A computer, even one without theories, can be of much assistance in this information-gathering. Many initial patterns go on for thousands of steps before settling into a cycle or dwindling away, and it is not hard to find patterns that continue for millions of steps. Following their progress by hand would be tedious at best.

The first step in writing a tag program is to choose a data structure for the string of 0s and 1s. The data structure must allow digits to be deleted at the front of the string and added at the rear. The structure I chose is a circular queue, or ring. One of its advantages is that when digits are added or deleted, the remaining digits never have to be moved. Some Turbo Pascal procedures for manipulating the queue are shown in Listing 1.

The usual way of implementing a queue is to declare an array of memory cells and then maintain pointers to the two cells that represent the head and the tail of the queue. This works well enough, but my personal preference is for a slight variation: instead of a tail pointer, I prefer to record the current length of the queue. When using head and tail pointers, special care must be taken to distinguish a full queue from an empty one, since both conditions might be signaled by having the head and tail pointers point to the same cell. With a head pointer and a length variable, the ambiguity cannot arise.

The routines *EnQ* and *DeQ* add and remove digits from the queue. One small peculiarity in their code is worth mentioning. Because a circular queue is being implemented by a linear array of memory cells, all calculations of position within the queue must be made modulo the queue size. In other words, if the head pointer is at the last cell in the array and it is

A two-headed Turing machine plays tag



Although Emil Post investigated the tag problem 15 years before Alan Turing invented his universal computer, a tag system can be regarded as a Turing machine program. A *read* head examines the first digit written on an infinite tape and instructs a *write* head to add either 00 or 1101 after the last digit. The *read* head then advances by three digits. Thus the tag problem would be solved if one could solve the Turing machine halting problem (the general question of whether a Turing machine halts when given an arbitrary input string). The halting problem is known to be insoluble, but it has not been proved that the special case represented by the tag problem cannot be solved.

Figure 2.

advanced by one cell, it should point to the first cell. An expression such as *(Head + 1) mod Qsize* would accomplish this, but the *mod* operator is comparatively slow in most languages. (It entails a division, which is almost invariably the slowest arithmetic operation.) By making *Qsize* a power of 2, the modulo calculation can be avoided. *(Head + 1) and Qmask*, where *Qmask* is a constant equal to *Qsize − 1*, has the same effect and runs much faster.

I decided to implement the queue as an array of characters, which is an extravagant waste of space. Each digit takes up a full byte, when it could be represented by a single bit. The underlying reason for this decision was laziness—I didn't want to complicate the routines with code for extracting individual bits—but I rationalized that I was trading space for time. I thought that on a byte-oriented machine routines for manipulating a character array would be faster than those for a bit array. On thinking about it further I'm not so sure. It's true that extracting bits would require masking operations and eventual conversion to character form; on the other hand, the more compact bitmap would drastically reduce the number of memory accesses. Perhaps some reader will settle the issue by trying both methods.

The function *TagStep* is at the heart of the program; it is called once for each step in the evolution of the tag system. It examines the first digit of the string, deletes three characters, and then adds either 00 or 1101 to the end. An initial version, based on calls to *EnQ* and *DeQ*, is shown in Listing 2. A more efficient version that eliminates the overhead of these procedure calls is given in Listing 3.

Detecting repetition

Much else is needed in a complete program for exploring tag systems: routines for input and output, for stepping through a sequence of initial patterns, for determining the fate of a pattern, for collecting statistics, and so on. Most of this code is straightforward, but one other area has an element of subtlety to it. How can a program detect when a tag system has entered a cycle? A brute-force solution is guaranteed to work: store each string as it is generated and then compare every new string with all the preceding ones. As soon as a match is found, the tag system is forever committed to a cycle.

The trouble with the brute-force method is that it requires entirely too much brute force. For a pattern that runs through a million iterations before entering a cycle, with an average string length of 1,000 digits, the storage requirement would be at least a gigabyte. Furthermore, roughly 500 million string comparisons would be needed. There are better ways, based on sampling rather than exhaustive record-keeping.

Consider a program that stores every hundredth string and compares each new string with these selected stored ones. Suppose a particular tag system enters a cycle at step 917 and that the period of the cycle—the number of steps between repeated values—is six. Thus for every value of *N* equal to or greater than 917, the string at step *N* will be identical to the string at step *N* + 6.

The program will not detect this cycle at the earliest possible moment (step 923), but soon after the next checkpoint (step 1,000), the repetition will become apparent. The string at step 1,006 will be identical to the one stored at step 1,000. Furthermore, the interval between the checkpoint and the step number where the cycle is detected gives the period of the cycle. The point at which the cycle actually begins can be found by returning to the previous checkpoint (step 900) and

```
Data structure and routines for manipulating a circular queue

{ global declarations }
const
 Qsize = 8192;                { must be power of 2 }
 Qmask = 8191;                { must be Qsize - 1  }
type
 StepCode = (OK, Over, Under); {result returned by TagStep}
 Qtype = record
  D      : array[0..Qsize] of char;
  Head   : integer;
  Len    : integer;
  StepNo : real;
 end;
{ Add 'Digit' to the tail of 'Q'. }
procedure EnQ(var Q : Qtype; Digit : char);
 begin
  Q.D[(Q.Head + Q.Len) and Qmask] := Digit;
  Q.Len := Succ(Q.Len);
 end;
{ Return a character removed from the head of 'Q'. }
function DeQ(var Q : Qtype) : char;
 begin
  Q.Len  := Pred(Q.Len);
  DeQ    := Q.D[Q.Head];
  Q.Head := Succ(Q.Head) and Qmask;
 end;
```

Listing 1

```
A prototype function for the tag problem, based on EnQ and DeQ

function TagStep(var Q : Qtype) : StepCode;     { prototype version }
 var
  Digit, Dump : char;
 begin
  if Q.Len < 3 then TagStep := Under             { queue empty }
  else if Q.Len = Qsize then TagStep := Over     { queue full  }
  else
   begin
    TagStep := OK;
    Digit := DeQ; Dump := DeQ; Dump := DeQ;      { read first and }
    if Digit = '1' then                          { dump next two  }
     begin
      EnQ('1'); EnQ('1'); EnQ('0'); EnQ('1');    { append 1101    }
     end
    else
     begin
      EnQ('0'); EnQ('0');                        { append 00      }
     end;
    Q.StepNo := Q.StepNo + 1.0;
   end;
 end;
```

Listing 2.

comparing successive pairs of strings separated by an interval of six steps.

A number of refinements can be added to this cycle-detection method, and the storage requirement can be reduced to as few as two strings. (In my implementation I kept track of five.) The optimum algorithm was worked out by R. William Gosper of Symbolics Inc. It is described briefly by Donald E. Knuth of Stanford University in volume 2 of *The Art of Computer Programming*. (The context there, incidentally, is the problem of detecting cycles in the output of a pseudorandom-number generator.)

Whatever method is used for detecting cycles, the program is sure to spend a substantial amount of its time comparing strings of digits. It is worthwhile making the comparison as efficient as possible. The function *EqQ* in Listing 4 is one solution. It exploits the fact that if two strings are identical, their lengths must be identical. A comparison of the lengths can settle the question for many pairs of strings without a digit-by-digit comparison of the string contents.

```
A more efficient tag function

function TagStep(var Q : Qtype) : StepCode;     { optimized version }
 begin
  if Q.Len < 3 then TagStep := Under              { underflow }
  else if Q.Len = Qsize then TagStep := Over      { overflow  }
  else
   begin
    TagStep := OK;
    if Q.D[Q.Head] = '1' then
     begin
      Q.Head := (Q.Head + 3) and Qmask;
      Q.Len := Q.Len - 3;
      Q.D[(Q.Head + Q.Len) and Qmask] := '1';
      Q.Len := Succ(Q.Len);
      Q.D[(Q.Head + Q.Len) and Qmask] := '1';
      Q.Len := Succ(Q.Len);
      Q.D[(Q.Head + Q.Len) and Qmask] := '0';
      Q.Len := Succ(Q.Len);
      Q.D[(Q.Head + Q.Len) and Qmask] := '1';
      Q.Len := Succ(Q.Len);
     end
    else
     begin
      Q.Head := (Q.Head + 3) and Qmask;
      Q.Len := Q.Len - 3;
      Q.D[(Q.Head + Q.Len) and Qmask] := '0';
      Q.Len := Succ(Q.Len);
      Q.D[(Q.Head + Q.Len) and Qmask] := '0';
      Q.Len := Succ(Q.Len);
     end;
    Q.StepNo := Q.StepNo + 1.0;
   end;
 end;
```

Listing 3.

```
A function for comparing two tag strings

{ Return TRUE if two tag strings are identical. }
function EqQ(var Q1, Q2 : Qtype) : boolean;
 var
  N : integer;
 begin
  EqQ := False;
  if Q1.Len = Q2.Len then    { check for equal lengths first }
   begin
    for N := 0 to (Q1.Len-1) do
     if Q1.D[(Q1.Head+N) and Qmask] <> Q2.D[(Q2.Head+N) and Qmask]
      then exit;
    EqQ := True;
   end;
 end;
```

Listing 4.

Observations and questions

Figure 3 shows the fates of 50 consecutive starting patterns beginning with an arbitrarily chosen value 24 digits long. Note that I have not traced all possible binary strings in this range but only those that differ at key positions. The reason is that the digits between the key positions are deleted by the tag transformations without ever being examined. For example, the initial patterns 100, 101, 110, and 111 all yield the same tag system because only the first digit in each pattern is acted on by the tag rules.

Along with the outcome of each pattern, Figure 3 gives the run length: the number of steps taken before the string either vanishes or enters a cycle. For those patterns that do cycle two additional items of information are listed: the period of the cycle and the number of digits in the first repeated string. About two-thirds of the 50 systems become cyclic, and the rest dwindle away after no more than about 400 steps. Most of those that enter a cycle do so after fewer than 100 steps, but there are exceptions. One system goes on for 2,125 steps before it gets stuck in a cycle; another lasts for 1,543 steps. More than half of the cycles have a period of six, and another 30% have a period of 10.

Looking at a larger sample suggests that these observations are fairly representative. A statistical summary of results for the first 10,000 consecutive tag systems is given in Figure 4. Three-fourths of the systems cycle and the rest dwindle. Periods of 6 and 10 continue to dominate; indeed, together they account for more than 90 percent of all the cycles. The longest run before a cycle begins is 24,563 steps, and the longest run before a system dwindles to extinction is 1,398 steps.

Based on these findings, there are a few observations I would make and a few questions I would ask.

The period of every cycle is an even number. The reason is not hard to find. (Hint: Consider why there can be no cycle with a period of one, that is, a single pattern that repeats continuously.)

The distribution of periods in any large sample of tag systems is more difficult to understand. In the first 10,000 systems the only periods that occur are 2, 4, 6, 8, 10, 12, 16, 28, and 40. Do the intervening even numbers ever turn up? Why are there no long periods of, say, 1,000 or 10,000 steps? What can explain the curious predominance of 6 and 10? Is there something about the tag transformation itself that favors cycles with these periods?

One clue to the distribution of periods

A sample of 50 consecutive tag systems

Starting string	Fate	Run	Period	Digits
100100100100000100000000	cycle	59	6	37
100100100100000100000100	dwindle	396		
100100100100000100100000	cycle	143	6	33
100100100100000100100100	cycle	53	6	37
100100100100100000000000	cycle	45	10	31
100100100100100000000100	cycle	47	6	33
100100100100100000100000	cycle	917	6	37
100100100100100000100100	cycle	21	6	33
100100100100100100000000	cycle	927	6	37
100100100100100100000100	cycle	2125	28	85
100100100100100100100000	cycle	1543	6	37
100100100100100100100100	cycle	853	6	37
0000000000000000000000000	dwindle	23		
0000000000000000000000001	dwindle	27		
0000000000000000000001000	cycle	18	6	13
0000000000000000000001001	dwindle	33		
0000000000000000001000000	dwindle	25		
0000000000000000001000001	dwindle	27		
0000000000000000001001000	cycle	22	4	13
0000000000000000001001001	cycle	18	6	19
0000000000000001000000000	cycle	20	4	13
0000000000000001000000001	dwindle	31		
0000000000000001000001000	cycle	46	6	19
0000000000000001000001001	cycle	26	6	19
0000000000000001001000000	cycle	16	6	19
0000000000000001001000001	cycle	88	10	31
0000000000000001001001000	cycle	24	6	19
0000000000000001001001001	cycle	80	10	31
0000000000001000000000000	dwindle	31		
0000000000001000000000001	dwindle	37		
0000000000001000000001000	dwindle	43		
0000000000001000000001001	dwindle	415		
0000000000001000001000000	dwindle	421		
0000000000001000001000001	cycle	56	10	31
0000000000001000001001000	cycle	86	10	31
0000000000001000001001001	cycle	118	10	31
0000000000001001000000000	dwindle	37		
0000000000001001000000001	dwindle	419		
0000000000001001000001000	cycle	22	6	19
0000000000001001000001001	dwindle	49		
0000000000001001001000000	cycle	94	10	31
0000000000001001001000001	cycle	72	10	31
0000000000001001001001000	cycle	20	8	23
0000000000001001001001001	cycle	62	10	31
0000000001000000000000000	dwindle	25		
0000000001000000000000001	dwindle	27		
0000000001000000000001000	cycle	22	4	13
0000000001000000000001001	cycle	18	6	19
0000000001000000001000000	dwindle	33		
0000000001000000001000001	dwindle	423		

Total systems tested:	50	
Systems that cycle:	31	62.00%
Systems that dwindle:	19	38.00%

Distribution of periods among systems that enter a cycle

Period	Number	Percent
4	3	9.68%
6	17	54.84%
8	1	3.23%
10	9	29.03%
28	1	3.23%

Fifty initial patterns are classified according to whether they eventually dwindle away or enter a cycle; the third possible fate—endless growth—has never been observed. The column labeled "Run" gives the number of steps required before the fate is determined. For strings that dwindle the run is assumed to end when the number of digits in the string falls below three; in other words, the starting pattern 000 has a run length of 1. For cyclic patterns the run ends when a string that will be repeated first appears; for example, the pattern traced in Figure 1 has a run length of 16. The period of a cycle is the number of steps from one appearance of any repeated string to the next appearance. The "Digits" column records the length of the string at the step where a cycle begins.

Figure 3.

is that many systems are in fact entering the same cycle. For example, a number of systems converge on the 19-digit string 0000011011101110100, which cycles with a period of 6. The 37-digit string 0000011011101110100000011011101110100 is essentially a doubling of this pattern, and it cycles with the same period. There are analogous patterns of 55 digits, 73 digits, and so on. Given that all the cycles investigated so far have quite short periods, there may be only a small number of distinct cycles being observed. Questions of this kind were addressed in the 1960s by Shigeru Watanabe of the University of Tokyo.

There is some evidence that as the starting patterns get longer, the proportion of systems that cycle increases (and hence the proportion that dwindle decreases). Does this trend continue indefinitely, so that for very long starting patterns the probability of entering a cycle approaches unity?

One might guess that as the number of 1s at key positions increases the average run length would also increase. (The converse is unquestionably true: with 0s in all key positions, the system dwindles away immediately.) Figure 5 tabulates the fates of the first 50 strings that have 1s in all key positions. In the notation used in the figure, $(100)^2$ means 100100, $(100)^3$ means 100100100, etc. There are certainly some long runs among these systems, most notably $(100)^{24}$, which continues for 4,346,269 steps before degenerating into a commonplace cycle with a period of 6. But if the curve is generally headed upward, there is a great deal of noise in it. Neither the size of the starting pattern nor the number of 1s in key positions would appear to be very reliable predictors of run length.

In both Figure 3 and Figure 5 the column headed "Digits," which gives the length of the string at the step where the system first enters a cycle, has a mysterious regularity. All of the numbers in this column are odd. When I first observed this fact after examining several hundred tag systems, I made the obvious conjecture that it is always true and set about finding out why.

I thought I was on the right track when I was able to show that if the number in the "Digits" column is always odd, then the cycle must always be entered on a 00 step, rather than a 1101 step. In other words, the last string before the repetitive sequence begins must have a 0 as its leading digit, so that the string length is reduced by one in the following step. This line of argument was looking very promising when I discovered that the conjecture itself is not true: a few starting patterns enter a cycle on a string with an even number of digits. In one sample of 1,000 tag systems, I found six such anomalous patterns. This is extremely vexing. In much of life, perhaps, the exception proves the rule, but in

mathematics a law valid in 99 and 44/100% of the cases is an abomination.

The tag problem has a tantalizing resemblance to another famous little stinker, generally known as the $3X+1$ problem. In this problem one begins with any positive integer X. If X is even, replace it with $X/2$; if it is odd, replace it with $3X+1$. Then apply the same rule to the new X. The question is: does X invariably descend to a value of 1 or are there some initial values of X for which the series diverges toward infinity? Like the tag problem, the $3X+1$ problem is unsolved. Is there any deep connection between them?

The tag problem as stated here is merely one example of an infinite class of related problems. The transformation rules could call for deleting any number of digits from the head of the string and appending any sequence of digits to the end. For example, one might delete two digits and add either 0 or 101.

A further generalization would be to use an alphabet with more than two symbols and devise transformation rules to be followed when each of these symbols is found at the head of the string. Post was able to solve all tag systems based on a two-symbol alphabet where no more than two symbols are deleted at each step. Minsky proved that when the alphabet has six symbols and six symbols are removed at each step, the tag problem is intrinsically insoluble. The gray area in between, including the (00, 1101) problem itself, is the area of active interest.

Post mortem

When Post began work on the tag problem in the early 1920s, his motive was not idle curiosity. He was engaged in an ambitious attempt to secure the foundations of mathematics. At the turn of the century David Hilbert had argued that all of mathematics and logic could be reduced to a formal system: a set of symbols and a set of rules for manipulating the symbols without regard for their meaning. Hilbert envisioned a mechanical process—what we would now call an algorithm—that in principle could complete mathematics. Given a set of axioms, or self-evident truths, the algorithm would generate all true statements derived from the axioms and no falsehoods.

Post was trying to prove the existence of such an algorithm when he stumbled onto the tag problem. He had already demonstrated that one very simple set of symbols and rules is complete and consistent; applying the rules to any axiom written in these symbols would yield all statements logically consistent with the axiom and only those statements. Unfortunately,

Statistics for the first 10,000 tag systems

Longest run before entering a cycle
000000100000000100100100100000010 (24,563 steps)
Longest cycle period
1001001001001001001000010 (40 steps)
Longest run before dwindling away
000000100000100100000100100000010 (1,398 steps)

Systems that cycle:	7,562	75.62%
Systems that dwindle:	2,438	24.38%

Distribution of periods among systems that enter a cycle

Period	Number	Percent
2	22	0.29%
4	109	1.44%
6	4110	54.35%
8	113	1.49%
10	2837	37.52%
12	14	0.19%
16	94	1.24%
28	226	2.99%
40	37	0.49%

The distribution of cycle periods is one of the most puzzling aspects of the tag problem. Periods of 6 and 10 seem to be strongly favored over all other possibilities.

Figure 4.

WALTZ LISP

The universal, superefficient LISP for MS-DOS and CP/M.

Waltz Lisp is a very powerful and complete implementation of Lisp. It is substantially compatible with established mainframe Lisps such as *Franz* (the Lisp running under *Unix*), *Common Lisp,* and *MacLisp.*

Ultra fast. In independent tests, **Waltz Lisp** was up to twenty(!) times faster than competing microcomputer Lisps.

Easy to use. Built-in WS-compatible full-screen file editor. Full debugging and error handling facilities are available at all times. No debuggers to link or load.

Practical. Random file access, binary file support, and extensive string operations make **Waltz Lisp** suitable for general programming. Several utilities are included in the package.

Full Lisp. Functions of type *lambda (expr), nlambda (fexpr), lexpr, macro.* Splicing and non-splicing character macros. Full suite of mappers, iterators, etc. Long integers (up to 611 digits). Fast list sorting using user defined comparison predicates. Built-in prettyprinting and formatting facilities. Nearly 300 functions in all.

Flexible. Transparent (yet programmable) handling of undefined function references allows large programs to reside partially on disk at run time. Automatic loading of initialization file. Assembly language interface.

Superbly documented. Each aspect of the interpreter is described in detail. The 300+ page manual includes an exhaustive index. Hundreds of illustrative examples.

Order **Waltz Lisp** now and receive *free* our
PROLOG Interpreter
Clog PROLOG is a tiny (but very complete) PROLOG implementation written entirely in **Waltz Lisp.** In addition to the full source code, the package includes a 50 page **Clog** manual.

16-bit versions require DOS 2.x or 3.x and 128K RAM (more recommended). Z-80 version requires CP/M 2.x or 3.x and 48K RAM minimum. **Waltz Lisp** runs on hundreds of different computer models and is available in all disk formats.

WALTZ LISP

$169

*Manual only: $30 (refundable with order). Foreign orders: add $5 for surface mail, $20 for airmail. COD add $3. Apple CP/M, hard sector, and 3" formats add $15. MC/Visa accepted.

For further information or to order call

1-800-522-LISP

In NJ and outside USA call 1-201-755-LISP

ProCode™ INTERNATIONAL
475 Watchung Ave.
Watchung, NJ 07060

CIRCLE 110 ON READER SERVICE CARD

Patterns with 1s in all key positions

Sequence	Fate	Run	Period	Digits
$(100)^{1}$	cycle	4	2	5
$(100)^{2}$	cycle	15	6	15
$(100)^{3}$	cycle	10	6	15
$(100)^{4}$	cycle	25	6	19
$(100)^{5}$	dwindle	409		
$(100)^{6}$	cycle	47	10	31
$(100)^{7}$	cycle	2,128	28	85
$(100)^{8}$	cycle	853	6	37
$(100)^{9}$	cycle	372	10	31
$(100)^{10}$	cycle	2,805	6	37
$(100)^{11}$	cycle	366	6	55
$(100)^{12}$	cycle	2,603	6	37
$(100)^{13}$	dwindle	701		
$(100)^{14}$	dwindle	37,910		
$(100)^{15}$	cycle	612	6	91
$(100)^{16}$	cycle	127	28	85
$(100)^{17}$	cycle	998	10	31
$(100)^{18}$	cycle	2,401	6	127
$(100)^{19}$	cycle	1,00	10	31
$(100)^{20}$	cycle	623	6	33
$(100)^{21}$	cycle	5,280	6	37
$(100)^{22}$	dwindle	1,776		
$(100)^{23}$	cycle	1,462	6	37
$(100)^{24}$	cycle	4,346,269	6	37
$(100)^{25}$	dwindle	4,127		
$(100)^{26}$	cycle	3,241	6	73
$(100)^{27}$	cycle	7,018	6	73
$(100)^{28}$	cycle	3,885	6	163
$(100)^{29}$	cycle	14,632	6	55
$(100)^{30}$	cycle	7,019	6	19
$(100)^{31}$	cycle	4,564	6	73
$(100)^{32}$	cycle	4,277	52	157
$(100)^{33}$	cycle	147,688	6	37
$(100)^{34}$	cycle	1,857	6	73
$(100)^{35}$	cycle	11,128	6	37
$(100)^{36}$	cycle	81,141	6	37
$(100)^{37}$	cycle	20,204	6	37
$(100)^{38}$	cycle	3,847	6	163
$(100)^{39}$	cycle	116,014	6	55
$(100)^{40}$	cycle	7,635	6	37
$(100)^{41}$	cycle	6,488	6	163
$(100)^{42}$	cycle	5,665	6	37
$(100)^{43}$	cycle	6,142	6	37
$(100)^{44}$	cycle	73,515	28	85
$(100)^{45}$	cycle	5,826	6	37
$(100)^{46}$	dwindle	6,060		
$(100)^{47}$	dwindle	3,779		
$(100)^{48}$	cycle	7,865	28	85
$(100)^{49}$	cycle	28,630	6	37
$(100)^{50}$	cycle	815,321	6	127

Statistics for run of 50 systems

Systems that cycle:	43	86%
Systems that dwindle:	7	14%

Distribution of periods among systems that enter a cycle

Period	Number	Percent
2	1	2.33%
6	33	76.74%
10	4	9.30%
28	4	9.30%
52	1	2.33%

A reasonable hypothesis is that strings with 1s at all the key positions might have longer runs than other strings. A tabulation of results for the initial strings 100, 100100, 100100100, etc. includes some quite long runs, but there is no clear pattern. Note that only one period length that was not seen in the first 10,000 tag systems appears in this sample: the string $(100)^{32}$, or the concatenation of 100 written 32 times, cycles with a period of 52 steps.

Figure 5.

the set of symbols was too limited to express much of interest about logic or mathematics.

When Post tried to extend his results to richer formal systems, he ran into trouble. There were too many symbols that could be manipulated in too many ways, and so he retreated, in several steps, to the tag problem, which at first appeared to be more tractable. As we have seen, he failed there, too.

The reason became apparent a decade later, when Kurt Gödel, a young Austrian mathematician, showed that no formal system could possibly accomplish what Hilbert dreamed of. As soon as a formal system becomes powerful enough to encode its own rules, it also becomes powerful enough to express contradictions.

Gödel's incompleteness theorem came as a great surprise to mathematicians, and not a pleasant one. It did not, however, end work on formal systems. Gödel had shown what cannot be done by pushing symbols around on paper; Alonzo Church, Alan Turing, and Post himself went on to show what can be done. And the power of the formal systems based on their work is not negligible; today we call those formal systems languages, and we give them names such as Pascal, LISP, and Ada. ■

References

Post, Emil. "Absolutely Unsolvable Problems and Relatively Undecidable Propositions: Account of an Anticipation," in *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, ed. by Martin Davis. Hewlett, New York: Raven Press, 1965. [Post did not write a full account of his work on tag systems until 20 years after the fact, and that account was not published until another 20 years had passed. This collection is the only place it appears. The volume also includes fundamental papers by Gödel, Church, Turing, and others.]

Watanabe, Shigeru. "Periodicity of Post's Normal Process of Tag," in *Mathematical Theory of Automata*. Brooklyn, N.Y.: Polytechnic Press, 1963. [Watanabe analyzes the kinds of patterns that can become periodic in a tag system.]

Minsky, Marvin L. *Computation: Finite and Infinite Machines.* Englewood Cliffs, N.J.: Prentice-Hall, 1967. [Includes a brief but lucid description of the tag problem in the context of computer science and language theory.]

Brian Hayes is a writer who works in both natural and formal languages. Until 1984 he was an editor of Scientific American, *where he launched the Computer Recreations department.*