****

COMPUTER SUPPLEMENT #19

****

In this issue:

Published in association with the American Cryptogram Association

INTRODUCTORY MATERIAL

The ACA and Your Computer (1p). Background on the ACA for computerists. (As printed in *ACA and You*, 1988 edition; [Also on Issue Disk #11]

Using Your Home Computer (1p). Ciphering at the ACA level with a computer. (As printed in *ACA and You*, 1988 edition).

Frequently Asked Questions (approx. 20p) with answers, from the Usenet newsgroup `sci.crypt`.

REFERENCE MATERIAL

BASICBUGS - Bugs and errors in `GW-BASIC` (1p). [Also on Issue Disk #11].
BBSFILES - List of filenames and descriptions of cryptographic files available on the ACA BBS (files also available on disk via mail).

BIBLIOG — A bibliography of computer magazine articles and books dealing with cryptography. (Updated August 89). [available on Issue Disk #11].

CRYPTOSUB - Complete listing of Cryptographic Substitution Program as published by PHOENIX in sections in *The Cryptogram* 1983–1985. (With updates from CS #2,3). [available on Issue Disk #3].

DISKEX - A list of programs and reference data available on disk in various formats (Apple—Atari—TRS80—Commodore—IBM—Mac). Revised March 1990.

ERRATA sheet and program index for Caxton Foster's *Cryptanalysis for Microcomputers* (3p). (Reprint from CS #5,6,7 and 9) [disk available from TATTERS with revised programs].

BACK ISSUES

$2.50 per copy. All back issues from #1 to #18 are once again available from the Editor.

ISSUE DISKS

$5 per disk; specify issue(s), format and density required. All issues are presently available on two IBM High Density 3.5 inch (1.44M) floppy disks, archived with PKZIP. For other disk formats, ask. Disks contain programs and data discussed in the issue. Programs are generally BASIC or Pascal, and almost all executables are for IBM PC–compatible computers. Issue text in TEX format is available for issues 16 to current. Available from the Editor.

TO OBTAIN THESE MATERIALS

```
     Write to:                       Or via Electronic Mail:

         Dan Veeneman                      dan@decode.com
         PO Box 2442                             or
         Columbia, Maryland             uunet!anagld!decode!dan
         21045-2442, USA.
```

Allow 6–8 weeks for delivery. No charge for hard copies, but contributions to postage appreciated. Disk charge $5 per disk; specify format and density required. ACA Issue Disks and additional crypto material resides on `Decode`, the ACA Bulletin Board system, `+1 410 730 6734`, available 24 hours a day, 7 days a week, 300/1200/2400/9600/14400 baud, 8 bits, No Parity, 1 stop bit. All callers welcome.

SUBSCRIPTION

Subscriptions are open to paid-up members of the American Cryptogram Association at the rate of US$2.50 per issue. Contact the Editor for non-member rates. Published three times a year or as submitted material warrants. Write to `Dan Veeneman, PO Box 2442, Columbia, MD, 21045-2442, USA.` Make checks payable to Dan Veeneman. UK subscription requests may be sent to G4EGG.

CHECK YOUR SUBSCRIPTION EXPIRATION by looking at the `Last Issue =` number on your address label. You have paid for issues up to and including this number.

Pattern Searching by Computer

D MELIORA

When we attack an aristocrat, our principal clues are apt to be be words with a distinctive pattern, such as `BOBKSKUI`. The repeated `B`s and `K`s should tell us something about the structure of the word, but while some of us have an uncanny eye for such things, the rest of us must simply scratch our heads and hope that inspiration will strike. I'm afraid I find myself in the latter category. Some of us may have gotten Raja's book, *Pattern and Non-pattern Words of 2 to 6 Letters*; but that book stops at six-letter words, while `BOBKSKUI` has eight letters. Furthermore, the longer the word, the more useful such information is apt to be, because the range of possibilities is less and so the number to be tried is smaller.

Personal computers make this task much easier. What we need to be able to do is (a) search a list of words and (b) test each word to see whether it matches the given pattern. The accompanying program, `Pat_Dict`, carries out both of these steps for us. I will start by describing step (b), since that is the more general part of the problem.

Pattern Matching

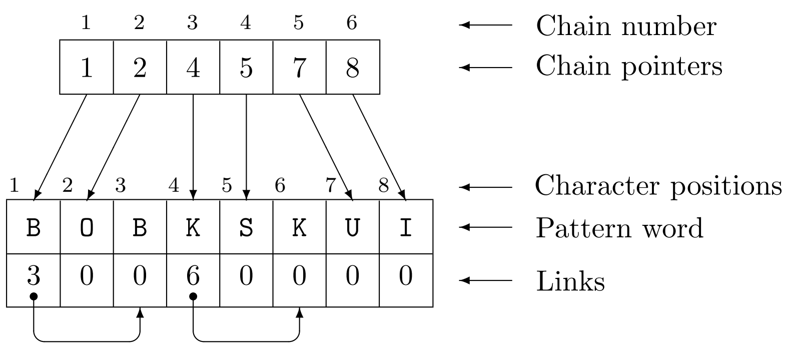
To test for a match to a given pattern, you need to characterize the pattern in some way that is independent of the ciphertext letters used and that will enable the computer to make the comparison quickly and easily. You must be able to show (in our example) that the first and third letters of any word matching the pattern are the same, that the fourth and sixth letters are the same, and that otherwise the letters are all different.

I chose to do this by representing the pattern by a set of chains which connect repeated letters together, together with pointers to the beginning of each chain. The easiest way to explain this is to use `BOBKSKUI` as an example. Before the search begins, a pattern-forming procedure goes through the pattern word letter by letter. If the letter has not appeared before, it is regarded as the beginning of a chain and the program assigns a pointer to it which gives its position in the pattern. For example, `B` is 1, `O` is 2. When it reaches a repeated letter, it creates a link from the most recent appearance of that letter to the current one. So the first letter, `B`, is linked to the third letter, `B`.

The pattern-forming procedure `Create_Links` returns an array of links and an array of chain pointers, the latter indicating where in the pattern each chain begins. In the links, a value of 0 indicates the end of the chain. For example, for `BOBKSKUI` we have

Chain pointers:	(1, 2, 4, 5, 7, 8)
Links:	(3, 0, 0, 6, 0, 0, 0, 0)

So to the program, the pattern is a set of six chains beginning at the first, second, fourth, fifth, seventh, and eighth letters in the pattern. The chain beginning at Letter 1 (`B`) points to letter 3 (`B`), and the 0 in position 3 indicates the end of that chain. Or, more graphically,



`Create_Links` creates the chains by scanning the pattern. If the current character is already associated with a chain (called a `class` in the code), its position is added to the end of the chain; otherwise a new chain is created for the character.

To test a candidate word, we first see whether its length is the same as that of the pattern. If we are starting out with a list of eight-letter words, this is, of course, unnecessary, but if we are doing a dictionary search, then we must use this requirement to speed the program.

If the lengths match, then the procedure `Test_Word` uses the pointer and link arrays to test the candidate. Suppose our candidate is `AGITATES`. The procedure runs through the array of chain pointers. It follows up each chain, examining each corresponding position in the candidate and testing whether all the letters are the same. For example, the first chain begins with the first letter of the word, `A`. The link value for this position is 3, which means that the third letter must also be an `A`. So `Test_Word` jumps immediately to the third letter of the candidate and compares it to `A`. In this example, the test fails, because the third letter is `I`, and the program gives up on that word right away without checking any other chains.

Next, suppose the candidate word is `NINETEEN`. Now the link tests will be satisfied because of the matching `N`s and the first two matching `E`s. But these tests miss the fact that the second-last letter is also an `E`. This is where the letter-to-chain associations come in. If all the link tests are passed, `Test_Word` then makes sure that all the chains are associated with different letters. But Chain 3 = `E` and Chain 5 = `E`, so this word is rejected.

Finally, suppose the word is `EVENINGS`. The link tests are passed, and all the chains are associated with different letters, so this word fits the pattern. The program can either write this word to a file or display it on the screen. In either case, it then goes on to the next candidate and continues until the list is exhausted.

This strategy is clumsy to explain, but simple to implement. A program written in Turbo Pascal (Version 5) for the IBM PC requires 40 lines of code to set up the link array and 26 lines of code to do the test. The program would be shorter still, except that I allowed '`?`' as a match-anything wildcard. (I also allowed capital letters to stand for themselves: Entering `bobkskui` now results in a search as described above; entering `AoAkskui` makes the program reject `EVENINGS` but accept `AVAILING` and `AWAITING`.)

Unpacking a Dictionary

The next question is, where do we get our list of words? The most obvious source is the dictionary that comes with our spelling checker or word-processing program. Such a dictionary is stored in a highly condensed form, in order to save disc space, and figuring out how to unpack it is where our cryptographic skills come into play. I cannot provide a general solution, because different checkers use different compression schemes, but I can show you a case study, using the dictionary that came with the spelling checker I use.

This dictionary uses a simple compression scheme based on the fact that consecutive words in a dictionary tend to begin with two or more letters the same. For example, the first word in this dictionary is `AARDVARK` (they don't include `A`; one-letter words are presumed to be spelled correctly) and the second word is `AARON`. Since these words share the common prefix `AAR-`, the second word is not stored whole, but rather as something like `(3)ON`. This means that it is made up of the first three letters of the previous word, followed by `ON`. This borrowing of prefixes from the immediately preceding words is the heart of the compression system.

When you examine the dictionary itself, you see nothing like this; the letters are not visible in the dump, which means that they are not represented by standard ASCII codes. It's pretty obvious that they have been further compressed. One cracks a compressed dictionary the way one would crack anything else, working from a known plaintext whenever possible, and we have `AARDVARK` and `AARON`. A good starting guess is that the letters are represented by their positions in the alphabet; thus `AARDVARK` would be 1, 1, 18, 4, 22, 1, 18, 11, or, in hexadecimal (as they will appear in a typical dump), `01 01 12 04 16 01 0B`. I saw nothing like this, either. But then, in scanning the dump, I found `00 00 11 03 15 01 8A`, not at the beginning of the file but 128 bytes down. These numbers are just off by 1 from my initial guess. So I was close: the letters are represented as *offsets* from the letter `A`. Also the presence of that `8A` suggested that the high bit was being used to mark the end of a word. (I found later that the first 128 bytes contained pointers to the first word for each letter—like a thumb-index—so a search for a specific word could start with any desired letter without scanning through the whole dictionary.)

The next word, `AARON`, should then have been `(3) 0E 8D`, where `0E` is the letter O and `8D` is N with the `8` indicating the end of the word. What I found was `2E 8D`. Now we must scratch our heads; the count of repeated letters has to be somewhere in that leading `2`. How many bits do we need to represent the letters themselves? Four aren't enough; it must be 5. So the assignment of bits within each byte is probably

```
s  x  x  a  a  a  a  a
```

where `s` is the sign bit, representing the end of the word and the `a` bits contain the letter. What we have is

```
s  x  x  a  a  a  a  a
0  0  1  0  1  1  1  0   (2E)
1  0  0  0  1  1  0  1   (8D)
```

So the `xx` bits in `2E` must give the number 3. But `xx = 01`, the number 1. Maybe the length is 2 greater than this? After all, if the number of repeated letters is only 1 or 2, you might just as well repeat them without compressing them. Comparing the next two or three words made this look like a good bet. Using it as a working hypothesis, I began to write `Get_Word`. I wrote (in Turbo Pascal),

```
icc := unpack;              { Unpack returns a byte from the file  }
low5 := icc and $1F;        { Low 5 bits contain letter offset     }
top3 := icc shr 5;          { Shift right to isolate top 3 bits    }
count := (top3 and 3) + 2;  { Isolate xx bits and add 2            }
```

This computation is done only on the first character of the word, and the program truncates the working string to the length given by `count` before tacking on any new characters. This procedure carried me through `AARDVARK`, `AARON`, `ABACK`, `ABACUS`, and `ABANDON`, but then I encountered `07 04 83`, and my program couldn't handle that. But this word is `ABANDONED`, which has the first 7 letters in common with the preceding word. Perhaps the rule was, if the variable `top3` was zero, then this byte wasn't a letter, but just the count alone. That would account for the `07`, and the `04 83` would supply the `-ED`.

The more I thought about this, the better it sounded. The `xx` bits can represent any number from 1 to 3, and when you add 2, they cover a range from 3 to 5. Thus any number greater than 5 is clearly out of range and would have to be represented by a byte of its own. When you remove prefixes this long, however, you are saving so much storage that you can easily afford a full byte for the count. So I revised the last line of my code to read,

```
if top3 = 0 then
   count := icc                { Long prefix  }
else
   count := (top3 and 3) + 2;  { Short prefix }
```

For the sake of brevity, I will omit the remaining details here, but what I have shown you is the key. You need a flag to tell when you are at the beginning of a word; when you are, you interpret the byte as I have described and truncate the previous word to the indicated length in order to obtain the start of the new word. After that, you simply append letters. When the high bit of `icc` is a 1, you have reached the end of a word. (There is also special handling for the apostrophe, which is stored unencoded.) The program for doing all this requires 36 lines of Pascal. As a test, I used the program to extract the first 6,000 words from the dictionary. When I fed the files containing these words back to the spelling checker, it reported no errors.

I have not examined any dictionaries other than this one, but in view of the fact that these files are coded for economy of storage rather than for encryption, I would not expect any dictionary to present serious problems, especially since any coding scheme must be simple enough for the program to be able to read and unpack the dictionary quickly. A familiarity with some of the standard data-compression techniques might be handy in some cases.

If you write your unpacking routine as a procedure along with the word-test routines I described previously, you can feed the pattern, `BOBKSKUI`, to the resulting program and get a list of seven words:

```
AVAILING  AWAITING  EVENINGS  IMITATED
IMITATES  IMITATOR  RARIFIED
```

On my system, this takes about 45 seconds, of which most of the time is taken by reading and unpacking the dictionary. In the crypt in which I found this word (A-11, by Funbug, from the Sept.–Oct., 1979 *Cryptogram*), the correct word turned out to be `AWAITING`.

Summary

Whether a program like this increases or reduces the pleasure of doing aristocrats is for you to say. For me, the pleasure was in the ingenuity and detective work in writing the program itself—that and the feeling that I was putting a foot, or at least a couple of toes, onto the professionals' turf. No doubt this is true for many people who do computer-aided cryptanalysis; maybe for them the appropriate wish is not "Good solving," but "Good programming."

References

Raja [R. V. Andree], *Pattern and Nonpattern Words of 2 to 6 Letters.* Norman, Okla., 1977.

PAT_DICT.PAS

```
Program PAT_DICT;
{$R+
        Program for finding words matching a given pattern.

        For use with the Word+ dictionary.  Using filetype FILE &
          Unpack faster than using filetype FILE OF BYTE & doing
          Reads (ca 11 seconds vs 50 seconds for a search).

        Operation:

          Reads pattern from console; then words from file & lists words
            matching the pattern.
                  (In this version, reads & decodes words from spelling
                  dictionary used by The Word (TM) spelling checker.)

          If output file requested, writes matching words to output
            file; otherwise lists them on console or printer.

          Pattern can consist of single digits or single alphabetics;
            any lower-case alphabetics may be used.  UPPER-CASE al-
            phabetics are literals: these are characters which must
            match exactly.  The special character "?" is a match-anything
            wildcard: characters in this position are excluded from
            consideration, as described below.

        Logic:

          Analyzes the pattern entered by user into a set of linked
            lists (represented by arrays).  Each list begins at the
            first instance of a given pattern element; the list item
            is the subscript of the next instance of that element
            (& 0 if there is no next instance).

          Also produces a list of equivalence classes, one for each dif-
            ferent pattern element.  The ith element of the Class array
            points to the ith different pattern element.
                  Example: the pattern, abbcbba generates the classes,
                  (1, 2, 4).  Class[1] points to the first a; Class[2]
                  points to the first b; Class [3] points to the c.

          Each element of the class array also points to the head of one
            of the linked lists.  Since the lists are disjoint, they are
            contained in a single array of bytes, named Link.
                  Example: for abbcbba, Link = (7, 3, 5, 0, 6, 0, 0).

          The characters which must be the same are found by using the
            Class pointer to locate the start of the list & then fol-
            lowing the chain of subscripts until a 0 is reached.
                  Example: for abbcbba, Class[2] = 2; that is, it points
                  to Link[2], which is the head of a list.  If you fol-
                  low up this list, you get (2, 3, 5, 6); these are the
                  locations of the b's in the pattern, & they give the
```

```
                locations of the four characters which must be the
                same in any word which matches the pattern.

        To test an unknown word for a match to the pattern word, the
          program loops through the Class array & follows up each
          linked list pointed to by Class[i] as described above.  The
          requirements for a match are:
                (a) All characters in any list must be the same;
                (b) all characters in different equivalence classes
                    must be different.

        A pattern symbol of "?" is a match-anything wildcard; charac-
          ters in positions marked by ?s may be anything.  They belong
          to no equivalence classes, they are ignored by Test_Word,
          they need not be all the same, & they need not be different
          from any of the identified equivalence classes.

        This strikes me as very complicated, but all the complication
          occurs in forming the Class & Link arrays, which is done
          only once, at the start; it does match testing very quickly
          & correctly & does it just the way one would want.  Reads,
          unpacks, & checks words in The Word Plus dictionary in about
          45 seconds on an AT-type machine (& about 11 seconds with
          a 386 at 25 MHz).

      Uses no floating-point arithmetic.

      T. Parsons      Aug.  2, 1990
             Revised June 26, 1991   Enabled repeated searches
                     May  30, 1993   Simpler logic in Create_Links;
                                       added search of updict;
                                       added pause on full screen.
                                                                      }
uses crt, dos, etime;
const
   lmax = 32;          { Max. acceptable word length:  }
                       {  must be < 255 (32 chars is   }
                       {  long enough to accommodate   }
                       {  'honorificabilitudinitatibus'}
                       {  with 5 chars to spare).      }
   signature: string = 'T. Parsons, May 30, 1993';
   buffsize = 256;             { Input buffer length   }
   bs1 = buffsize - 1;
   blks = buffsize div 128;    { Number of blocks read }

type
   fstring = string[72];       { For file names        }
   wtype = string[lmax];       { For words & pattern   }
   ltype = array [1..lmax]     { For links & classes   }
                   of byte;
   litrec = record             { Key for literal test  }
               pos: integer;   { Where to look         }
               let: char;      { What you should find  }
```

```
         end;
   litlist = array [1..lmax] of litrec;
   buffarray = array [0..bs1] of byte;

var                  { Globals                          }
   ibuf: buffarray;      { Input buffer                 }
   literals: litlist;    { Literals in pattern          }
   pstring: wtype;       { Pattern                      }
   class,                { Array of equivalence classes }
   link: ltype;          { Links to characters          }
   attribs: word;        { File's attributes            }
   ipoint,               { Input buffer pointer         }
   lgth,                 { Pattern length               }
   lgth2,                {    "      "    + 1           }
   lines,                { Output line count            }
   llgth,                { Cumulative output line length }
   nclass,               { Number of equivalence classes }
   nlits: integer;       { Number of literals in pattern }
   device: char;         { Output device code           }
   fname: fstring;
   main_done,            { True: ^Z in main dictionary  }
   done: boolean;        { True: ^Z in upd dictionary   }
   inf: file;
   upd: text;
                     { Local to Main                    }
   mcount: integer;      { Count of matching words      }
   t_word: wtype;        { Word being tested            }
   writing: boolean;     { True => output enabled       }
   outf: text;

{$I select.inc }         { For selecting output device  }
{$I agree.inc }

procedure CREATE_LINKS (pstring: wtype; lgth: integer;
                var class, link: ltype; var lits: litlist;
                var nclass, nlits: integer);
                                { Analyzes the pattern & makes equiva-  }
   const                        {   lence-class & link arrays           }
      caps: set of char = ['A'..'Z'];
   var
      i,                 { Loop          }
      j: integer;        {   counters    }
      cc: char;          { Current char. }
      found: boolean;
   begin
   for i := 1 to lgth do        { Link is initialized to 255 through-   }
      begin                     {   out; as links are found, 255's are  }
      class[i] := 0;            {   replaced by link values; 255 thus   }
      link[i] := 255;           {   identifies unlinked positions       }
      end;
   nlits := 0;
   nclass := 0;
   for i := 1 to lgth do                { Loop through pattern  }
```

```
      begin
      cc := pstring[i];
      if cc in caps then               { Literal:             }
         begin
         inc (nlits);
         lits[nlits].pos := i;
         lits[nlits].let := cc;
         end  { if }
      else if cc <> '?' then           { Not a wildcard       }
         begin
         link[i] := 0;
         found := false;
         j := 1;                       { Already in a chain?   }
         while (j <= nclass) and not found do
            if cc = pstring[class[j]] then
               found := true
            else
               inc (j);
         if found then                 {  --yes:              }
            begin
            j := class[j];             { Go out to end        }
            while link[j] <> 0 do      {   of chain           }
               j := link[j];
            link[j] := i
            end
         else
            begin                      {  --no:               }
            inc (nclass);              { Create new class     }
            class[nclass] := i         {   for this character  }
            end
         end  { else }
      end;  { for i }
   write ('Letter classes: ');         { Show the results     }
   for i := 1 to nclass do
      write (class[i]:3);
   writeln;
   write ('Links: ');
   for i := 1 to lgth do
      write (link[i]:4);
   writeln
   end;  { Create_Links }

function UNPACK: byte;          { Returns one input byte from file      }
   begin                               { NB: When last block read, Eof }
   if ipoint > bs1 then                { immediately goes True; hence  }
      if not eof(inf) then             { i/o loop must not terminate   }
         begin                         { on Eof, or contents of last   }
         blockread (inf, ibuf, blks);  { block will not be unpacked.   }
         ipoint := 0;                  { This code leaves IPoint > bs1 }
         end;                          { when Eof makes BlockRead im-  }
   if ipoint < buffsize then           { possible; i/o loops should    }
      unpack := ibuf[ipoint];          { terminate on IPoint > buff-   }
   inc (ipoint)                        { size & eof(inf).              }
   end;  { Unpack }

procedure GET_WORD (var workstring: wtype);
   var                                  { Unpacks & decodes a word      }
      icc,                              {   from the Word+ dictionary   }
      top3: byte;                       { On first call, workstring     }
      count,                            {   must be blank, & first      }
      low5: integer;                    {   call must access beginning  }
      cc: char;                         {   of dictionary (can't start  }
      new_word: boolean;                {   downstream).                }
   begin
   new_word := true;
   repeat
      icc := unpack;            { Get a byte            }
      top3 := icc shr 5;        { Separate code & char  }
      low5 := icc and $1f;
      if low5 = 26 then main_done := true;
      if low5 < 26 then low5 := low5 + ord('A');
      cc := chr(low5);          { Convert character     }
      if not main_done then
         if new_word then
            begin
            if top3 = 0 then
               count := icc
            else
               count := (top3 and 3) + 2;
            workstring[0] := chr(count);
            new_word := false;
            if top3 <> 0 then
               workstring := workstring + cc;
            end
         else
            workstring := workstring + cc;
      if icc and $80 <> 0 then
         new_word := true
   until new_word or main_done or done
   end;  { Get_Word }

function TEST_WORD (var t_word: wtype): boolean;
   var                                  { Examines word for match       }
      i, j,                             {   to pattern                  }
      nc,              { Equiv. class counter  }
      jfrom,           { Previous link         }
      jto: integer;    { Current link          }
      cc: char;        { Char for this class   }
      ok: boolean;     { True => match         }
   label
      quick_exit;
   begin
   ok := true;                          { Innocent until proved guilty  }
   nc := 0;
   while ok and (nc < nclass) do        { Loop through classes          }
      begin
      inc (nc);
      jfrom := class[nc];              { Examine new class            }
      cc := t_word[jfrom];
      while ok and (link[jfrom] <> 0) do       { Traverse list for     }
         begin                                 {   this class; linked  }
         jto := link[jfrom];                   {   chars must be all   }
         if t_word[jto] = cc then              {   the same            }
            jfrom := jto
         else
            ok := false;
         end;
      end;  { loop on classes }
   if ok then                          { Now make sure all classes     }
      for i := 1 to pred(nclass) do    {   different                   }
         for j := succ(i) to nclass do
            if t_word[class[i]] = t_word[class[j]] then
               begin
               ok := false;
               goto quick_exit  { Abort on any match    }
               end;
quick_exit:
   test_word := ok;
   end;  { Test_Word }

function LITERAL (t_word: wtype): boolean;
   var                                 { Checks for match of literals  }
      i: integer;                      {   if any.  Returns True if    }
      ok: boolean;                     {   literals match or if no     }
   begin                               {   literals; false otherwise.  }
   ok := true;
   i := 1;
   while ok and (i <= nlits) do
      if upcase(t_word[literals[i].pos]) <> literals[i].let then
         ok := false
      else
         inc (i);
   literal := ok
   end;  { Literal }

procedure CHECK (t_word: wtype);               { Tests word & displays }
   begin                                       {   if match found      }
   if literal(t_word) and test_word(t_word) then
      begin                    { Found a match:        }
      inc (mcount);            {   write it.           }
      write (outf, t_word:lgth2);
      llgth := llgth + lgth2;
      if llgth >= 80 then      { For formatting output }
         begin
         writeln (outf);
         inc (lines);
         if not writing and (lines > 23) then
            if agree ('More?') then
               lines := 1
            else
               begin
               main_done := true;
               done := true
               end;
         llgth := lgth2
         end
      end
   end;  { Check }

{ = = = = = = = =  M a i n   P r o g r a m  = = = = = = = = = = = = =  }

begin
   writeln ('This is Pat_Dict [6-26-91].');
   assign (inf, 'e:\text\maindict.cmp');
   getfattr (inf, attribs);     { Make dictionary readable      }
   setfattr (inf, attribs and $FE);
   assign (upd, 'e:\text\updict.cmp');
   getfattr (upd, attribs);     { Make dictionary readable      }
   setfattr (upd, attribs and $FE);
   ipoint := maxint;
   write ('Pattern (? for wildcards, caps for literals,',
                        ' <cr> to quit): ');
   readln (pstring);
   lgth := length(pstring);
   while lgth > 0 do
      begin
      lines := 0;
      reset (inf);
      seek (inf, 1);            { Bypass directory              }
      create_links (pstring, lgth, class, link,
                literals, nclass, nlits);
      repeat until select_output (outf, device, fname);
      writing := device = 'F';
      mcount := 0;
      lgth2 := lgth + 2;
      llgth := lgth2;
      main_done := false;
      done := false;
      timer (0);                { Start timing the search       }
      writeln ('Searching main dictionary.');
      repeat
         get_word (t_word);
         if not main_done then
            if length(t_word) = lgth then
               check (t_word)
      until main_done;
      reset (upd);
      if not writing then       { Output formatting details     }
         begin
         lgth2 := lgth + 2;
         llgth := lgth2;
         if wherey > 1 then
```

```
            writeln
         end;
      writeln ('Searching supplementary dictionary.');
      repeat
         readln (upd, t_word);
         if not done then
            if length(t_word) = lgth then
               check (t_word)
      until done or eof(upd);
      if wherex > 1 then writeln;
      timer (1);                { Report search time            }
      if mcount = 0 then
         begin
         write ('No match found');
         if writing then
            writeln ('; output file empty.')
         else
            writeln ('.');
         end
      else
         writeln (mcount, ' matching word(s) found.');
      if writing then close (outf);
                                { Query for next time           }
      write ('Pattern (? for wildcards, caps for literals,',
                        ' <cr> to quit): ');
      readln (pstring);
      lgth := length(pstring);
      end;  { while }
   close (inf);
   setfattr (inf, attribs);     { Restore dictionary attributes }
   close (upd);
   setfattr (upd, attribs);     { Restore dictionary attributes }
end.
```

SELECT.INC

```
{ Output selection utility.  Prompts user for choice of crt, printer,
    or file; if file selected, tries to open it.  Returns result: true
    if crt or printer selected or if output file requested.             }

function OPEN_OUT (var outf: text;
                               var fname: fstring): boolean; forward;
{$I open_out.inc }

function SELECT_OUTPUT (var outdev: text;
                   var device: char; var fname: fstring): boolean;
   begin
   write ('Output to C[onsole], P[rinter], S[erial port], or F[ile]? ');
   repeat
      device := upcase(readkey);
      write (device);
      readln;
      select_output := true;
      case device of
        'P': begin
             assign (outdev, 'lpt1');
             rewrite (outdev);
             write (outdev, #27, 'N', #10);
             end;
        'S': begin
             assign (outdev, 'com1');
             rewrite (outdev);
             end;
        'C': begin
             assign (outdev, 'con');
             rewrite (outdev)
             end;
        'F': select_output := open_out (outdev, fname);
        else write ('Huh??? ');
        end;  { case }
   until device in ['P', 'F', 'C', 'S'];
   end;  { Select_Output }
```

AGREE.INC

```
type __ptype = string[80];

function AGREE (prompt: __ptype): boolean;
   var
      response: __ptype;
      ok: boolean;
   begin
      write (prompt, ' ');
      repeat
         if keypressed then readln;
         readln (response);
         ok := (response <> '') and (response[1] in ['Y', 'y', 'N', 'n']);
         if NOT ok then write ('Huh??? ');
      until ok;
      agree := response[1] in ['Y', 'y'];
   end;
```

SWAGMAN CIPHER

BOATTAIL

The SWAGMAN cipher is a basic transposition cipher based on a numbered block of squares. The most common periods (block sizes) are 4, 5, and 6. A typical SWAGMAN block of period 4 looks like this:

```
1432
4213
3124
2341
```

In the columns and rows, no number is repeated. Enciphering in SWAGMAN using this block produces the following:

```
Plain:    THET RUEB EAUT YO
          FTHE COMP UTER LI
          ESIN ITSF LAWL ES
          SREP ETIT IONE TO

Cipher:   TSHP RTMT EAEE YS
          STIT EOSB ITWT TI
          EREE ITEP LOUR EO
          FHEN CUIF UANL LO

Cryptogram: TSEFS TRHHI EEPTE NREIC ....
```

As you can see, each column of the block is shifted according to the corresponding key numbers in the SWAGMAN square. The plaintext letters in squares numbered 1 are transposed into the first row of the ciphertext, those numbered 2 into the second row, and so on. The nulls `ETO` are added at the end of the plaintext to fill out the columns (Ref. ACA and You, page 50).

The main cryptanalytic weakness in this system is the relatively limited number of keys. Each row of the key square contains a simple rearrangement of 4 numbers. The number of possible rearrangements of 4 numbers is 4 factorial, that is 4x3x2x1 which equals 24. Each of the four rows contains one of only 24 possible keys. For squares of 5, 6, 7, and 8 there are 120, 720, 5040, and 40,320 possible keys, respectively. Periods 7 and 8 are rarely used in Cryptograms, and are not deciphered by this program.

It would be quite possible for a cryptanalyst to manually try all 24 keys in period 4 and, with a lot of time and patience, to try the 120 keys in period 5. Manually trying 720 keys for period 6 is too much. This is essentially 'grunt work', the kind that computers were designed to do.

The program `SWAGMAN.PAS` attacks the cipher by brute force, that is, it tries every possible key arrangement, stores each result, and then analyzes each result to find those that best resemble plain language. The user then chooses the correct lines of plaintext from the display and joins them together to read the complete message.

The first part of the program contains the three test ciphers. There is a test cipher for periods of 4, 5, and 6 to aid in testing any modifications to the program. This test cipher is entered into the working program by typing `test4`, `test5`, or `test6` when prompted by the cipher input module.

The three pattern arrays hold the basic number patterns used to generate all possible keys. For purposes of brevity, I chose to program in only one subset of numbers and then rotate them to produce the full pattern. The pattern 0123 is rotated to provide patterns 1230, 2301, and 3012. This cuts the size of the test pattern arrays by a factor equal to the cipher period.

The first step in using the `SWAGMAN` program is to enter the cipher. A maximum length of 288 cipher letters is allowed for, not including spaces. Procedure `SWAGCIPHERIN` takes the input in lines and puts it into character array `INTEXT`. All input spaces are removed by procedure `SQUEEZE`. `PERIODCHECK` does a simple test to determine which periods are possible; since there can be no incomplete columns in a

SWAGMAN, the cipher length must be evenly divisible by the period.

The next step is to correlate the cipher. Depending on the main menu choice, the period is set and master module `CORRELATE` transposes the cipher. If you choose the wrong period, you can return to the main menu and simply choose another.

`INITIALIZE` sets the block sizes, number of blocks, and size of last block, for the period chosen. The cipher text is arranged into blocks by `FILLBLOCKS`.

`ASSEMBLESOLUTIONS` is the module that derives every possible decipherment. The results are fed into `solarray`, character by character, using the pattern array `allposs` to control the transposition.

`DIGRAPHCHECK` compares the results in `solarray`, digraph by digraph, with the English digraph frequencies in file `DIGRAMS.DAT`, which are downloaded into array `table` by procedure `LOADDIGRAPHTABLE` at the beginning of the program. The scores and the location of each score are posted to arrays `score` and `location` respectively.

`SOLBUBBLESORT` sorts the two arrays `score` and `location` into descinding order of the scores. This means that the locations with the best scores and the closest resemblance to plaintext are put at the top. When you choose `Display All Solutions` from the main menu, the procedure `DISPLAYSOLUTIONS` prints the solutions on the screen in the order of the `location` array.

There are two `Hardcopy` options on the main menu; you can either print out the best 20 solutions or all the solutions. If the period is 6, there are 720 solutions, which takes about 12 pages to print. There are no printer-specific codes in the `HARDCOPY` procedure so it should work well with almost any printer.

This program has worked well with every Swagman cipher I have tried it on. Most of the plaintext lines are in the first 10 line displayed. If one line contains nulls, or unusual digraphs, it will score lower and you will need to search farther down the list.

Happy solving!

```
OUTPUT FROM ‘test5’ CIPHER:

UNDELDYEAPSOFFREITHORNES
UNDREDYEARSOFAGEITBURNED plaintext
DOFLEGENDONREACHINGFIVEH plaintext
ITSREFONARYREAGOMWBUSEAD
UESALDTHEPSHOEREXAROREQQ
UEDPLDTEIPSHFFREXTOOREER
IOFANFENERYREENOINROSVEQ
HESPHOTNIXPHEFAIXWOUSEAR
THEPHOENIXISAFABULOUSBIR plaintext
ITFRNFONARYREANOMNBOSEED
ITSELFONAPYREFROMWHOSEAS plaintext
TNDRNOYEARIOFANBITBOSNED
UTEPEDONIRSRAFGEMLOUREIR
HESANOTHERPHOENIXAROSEQQ plaintext
UODPEDEEIRSRFFGEITOURVER
     (105 more lines)
```

SWAGMAN.PAS

```
program SWAGMAN;  {Uses brute force to decipher Swagman Cipher
                    in periods 4, 5, & 6}
    USES Crt,Dos,Printer,Ciphlib; {uses CIPHLIB library routines}
    CONST
    test41='UTEON ITOKE ESANE VEEPE EANPO MDIRS UORTO ACFEO FEOCO OTFFR';
    test42='LLHDE OUCEF EABTI LEDBT VE/';
    {MA93 E-20 G4EGG ‘To keep out of trouble one needs a cool head but even
```

```
 more effective is a pair of cold feet.}
allposs4 :array[0..5,0..3] of Integer =((0,1,2,3),(0,1,3,2),(0,2,1,3),
         (0,2,3,1),(0,3,1,2),(0,3,2,1));
test51='UIDTH HNTEO SSDFE LPAER NEHEL DFGOO EYOTE NHENN DIEAA ROXRP';
test52='SYNIP SORHR EOFEA AFEFA NCAGR EOHBI UIMXI WATNL GORHB OFUUO';
test53='RSISS BNEEV AQEEI HRQSD/';
{SO92 E-19 ZYZZ ‘The phoenix is a fabulous bird of legend. On reaching
 five hundred years of age it burned itself on a pyre from whose ashes
 another phoenix arose QQ  key = 30412,24301,01243,12034,43120}
allposs5 :array[0..23,0..4] of Integer =
        ((0,1,2,3,4),(0,1,2,4,3),(0,1,3,2,4),(0,1,3,4,2),(0,1,4,2,3),
         (0,1,4,3,2),(0,2,1,3,4),(0,2,1,4,3),(0,2,3,4,1),(0,2,3,1,4),
         (0,2,4,1,3),(0,2,4,3,1),(0,3,1,2,4),(0,3,1,4,2),(0,3,2,1,4),
         (0,3,2,4,1),(0,3,4,1,2),(0,3,4,2,1),(0,4,1,2,3),(0,4,1,3,2),
         (0,4,2,1,3),(0,4,2,3,1),(0,4,3,1,2),(0,4,3,2,1));
test61='DTTLESKHANIEEGAMSYHSAAHTHHTWRAUYORONRMDINANRKDNEIGENBRCELAVRSSAFOE';
test62='TWEOONRDNSOENHMRTNERMSEAITTCRCRAKTENUIEEIENTWTSNOCGEWAOIDFRTCHHIMA';
test63='APEEUPODLUHIASTNWSOERETERHRNRDEDNVSSIUSREHRWCEIAVELMGYOOEEESSPARBA';
test64='LNKUTNSDADWADHINRCENEEAOONIMJETUAPSNSSESIGWETTRRDAHEAA/';
{Lightning can strike twice in the same place. Just ask Mary Lowenstein of
Hudson, New York, who operates a home based secretarial services business.
When a thunderstorm caused a power surge and damaged the hard drive of her
new computer, she lost an entire day’s work. Reformatting the hard drive
and reinsta. keys=514023,240531,453102,031254,302415,125340}
allposs6 : array[0..119,0..5] of Integer =
 ((5,0,1,2,3,4),(5,1,0,2,3,4),(5,2,0,1,3,4),(5,3,0,1,2,4),(5,4,0,1,2,3),
 (5,0,1,2,4,3),(5,1,0,2,4,3),(5,2,0,1,4,3),(5,3,0,1,4,2),(5,4,0,1,3,2),
 (5,0,1,3,2,4),(5,1,0,3,2,4),(5,2,0,3,1,4),(5,3,0,2,1,4),(5,4,0,2,1,3),
 (5,0,1,3,4,2),(5,1,0,3,4,2),(5,2,0,3,4,1),(5,3,0,2,4,1),(5,4,0,2,3,1),
 (5,0,1,4,2,3),(5,1,0,4,2,3),(5,2,0,4,1,3),(5,3,0,4,1,2),(5,4,0,3,1,2),
 (5,0,1,4,3,2),(5,1,0,4,3,2),(5,2,0,4,3,1),(5,3,0,4,2,1),(5,4,0,3,2,1),
 (5,0,2,1,3,4),(5,1,2,0,3,4),(5,2,1,0,3,4),(5,3,1,0,2,4),(5,4,1,0,2,3),
 (5,0,2,1,4,3),(5,1,2,0,4,3),(5,2,1,0,4,3),(5,3,1,0,4,2),(5,4,1,0,3,2),
 (5,0,2,3,4,1),(5,1,2,3,0,4),(5,2,1,3,0,4),(5,3,1,2,0,4),(5,4,1,2,0,3),
 (5,0,2,3,1,4),(5,1,2,3,4,0),(5,2,1,3,4,0),(5,3,1,2,4,0),(5,4,1,2,3,0),
 (5,0,2,4,1,3),(5,1,2,4,0,3),(5,2,1,4,0,3),(5,3,1,4,0,2),(5,4,1,3,0,2),
 (5,0,2,4,3,1),(5,1,2,4,3,0),(5,2,1,4,3,0),(5,3,1,4,2,0),(5,4,1,3,2,0),
 (5,0,3,1,2,4),(5,1,3,0,2,4),(5,2,3,0,1,4),(5,3,2,0,1,4),(5,4,2,0,1,3),
 (5,0,3,1,4,2),(5,1,3,0,4,2),(5,2,3,0,4,1),(5,3,2,0,4,1),(5,4,2,0,3,1),
 (5,0,3,2,1,4),(5,1,3,2,0,4),(5,2,3,1,0,4),(5,3,2,1,0,4),(5,4,2,1,0,3),
 (5,0,3,2,4,1),(5,1,3,2,4,0),(5,2,3,1,4,0),(5,3,2,1,4,0),(5,4,2,1,3,0),
 (5,0,3,4,1,2),(5,1,3,4,0,2),(5,2,3,4,0,1),(5,3,2,4,0,1),(5,4,2,3,0,1),
 (5,0,3,4,2,1),(5,1,3,4,2,0),(5,2,3,4,1,0),(5,3,2,4,1,0),(5,4,2,3,1,0),
 (5,0,4,1,2,3),(5,1,4,0,2,3),(5,2,4,0,1,3),(5,3,4,0,1,2),(5,4,3,0,1,2),
 (5,0,4,1,3,2),(5,1,4,0,3,2),(5,2,4,0,3,1),(5,3,4,0,2,1),(5,4,3,0,2,1),
 (5,0,4,2,1,3),(5,1,4,2,0,3),(5,2,4,1,0,3),(5,3,4,1,0,2),(5,4,3,1,0,2),
 (5,0,4,2,3,1),(5,1,4,2,3,0),(5,2,4,1,3,0),(5,3,4,1,2,0),(5,4,3,1,2,0),
 (5,0,4,3,1,2),(5,1,4,3,0,2),(5,2,4,3,0,1),(5,3,4,2,0,1),(5,4,3,2,0,1),
 (5,0,4,3,2,1),(5,1,4,3,2,0),(5,2,4,3,1,0),(5,3,4,2,1,0),(5,4,3,2,1,0));
TYPE
bigintarray = array[0..719] of Integer;
VAR
test4,test5,test6    :String;  {test cipher texts}
```

```
    exitflag       :Boolean;
    yesflag        :Boolean;
    period         :Integer; {period of cipher, if 6 then 0-5}
    numsols        :Integer; {number of solutions for each period}
    corr_flag      :Boolean; {is cipher correlated yet?}
    selnum         :Integer; {menu selection}
    intext         :txtarr;
    clast          :Integer; {last element of cipher arrays}
    ciphblk        :array[0..7,0..5,0..5] of Char; {8 blocks, max. 6x6}
    lastblk        :Integer; {last cipher block used}
    lastcol        :Integer; {last column of lastblk used}
    solarray       :array[0..719,0..48] of Char; {all possible solutions}
    sollast        :Integer; {last element of solution arrays}
    table          :array['A'..'Z','A'..'Z'] of Byte;
    score          :bigintarray; {correlation scores of sol's}
    location       :bigintarray; {location of each score}
function factorial(p:Integer):Integer;
    var   x,y   :integer;
    begin
    y:=p;for x:=(p-1) downto 2 do y:=y*x; factorial:=y;end;
procedure swagcipherin;
          var
          Instr :String; b,clth :Integer;
          begin
               ClrScr;Textcolor(LightGreen);
               Writeln('Enter Swagman Cipher, line by line');
               Writeln('288 characters max.');
               Writeln('Ctrl-x to restart, back to correct');
               Writeln('"test4" for test cipher in period 4');
               Writeln('"test5" for test cipher in period 5');
               Writeln('"test6" for test cipher in period 6');
               Writeln('"/" to end entry');
               Textcolor(White);
               b:=1;instr:='';clth:=0;
               repeat
                  if b>Length(instr) then begin
                     b:=1;Readln(instr);
                     if instr='test4' then instr:=test4;
                     if instr='test5' then instr:=test5;
                     if instr='test6' then instr:=test6;
                     end;
                     if instr[b]<>'/' then begin
                        intext[clth]:=Upcase(instr[b]);
                        Inc(clth);Inc(b); end;
               until (clth>287) or (instr[b]='/');
          clast:=clth-1;
          end;
procedure periodcheck;
    var   check       :array[4..6] of String[5];
          x,z         :Integer;
    begin
    for x:=4 to 6 do begin
        z:=clast+1;
        if ((z mod x)<>0) or ((z div (x*x))>8)
          then check[x]:='FALSE' else check[x]:='TRUE';
        end;
    ClrScr; TextColor(Black);GotoXY(1,5);
    Writeln('Cipher Length is ',z,' Letters',^J^J);Textcolor(White);
    for x:=4 to 6 do
        Writeln('Period ',x,' is ',check[x],^J);
    any_key;
    end;
procedure solbubblesort;
          {sorts solution digraph scores in descending order}
    var    x    :Integer;
           flag   :Boolean;
    begin
    repeat
        flag:=true;
        for x:=0 to (numsols-1) do
            if score[x]<score[x+1] then begin
                intswap(score[x],score[x+1]);
                intswap(location[x],location[x+1]);
                flag:=false;
                end;
    until flag=true;
    end;
procedure digraph_check;
     var
         r,x     :Integer;
         begin
         for x:=0 to numsols do begin
             score[x]:=0;location[x]:=0;end; {clear score & location arrays}
         for x:= 0 to numsols do begin
             location[x]:=x;
             for r:= 0 to sollast-1 do
                 score[x]:=score[x]+table[solarray[x,r],solarray[x,r+1]];
             end;
         end;
procedure initialize;
    var   n,square   :Integer;
    begin
    numsols:=factorial(period+1)-1; {index of possible solutions, 0-numsols}
    sollast:=((clast+1) div (period+1))-1; {index of last solution, 0-sollast}
    square:=(period+1)*(period+1); {number of squares in each block}
    lastblk:=((clast+1) div square); {index of last cipher block}
    n:=(clast+1) mod square; {if last block complete, n=0}
    if n=0 then begin Dec(lastblk); lastcol:=period; end
       else lastcol:=(n div (period+1))-1;
    end;
procedure fill_blocks;
    var    cl,v,x,y,z  :Integer;
    begin
    cl:=0;
    FillChar(ciphblk,Sizeof(ciphblk),#32); {clear ciphblk to blanks}
    for x:=0 to lastblk do begin
```

```
        if x=lastblk then v:=lastcol else v:=period;
        for y:=0 to v do
            for z:=0 to period do begin
                ciphblk[x,y,z]:=intext[cl];
                Inc(cl);
                end;
        end;
    end;
procedure display_blocks; {display cipher blocks}
    var   c,r,x,y,z   :Integer;
    begin
    ClrScr; Textcolor(Green);
    c:=1;r:=4;
    for x:=0 to lastblk do begin
        for y:=0 to period do
            for z:=0 to period do begin
                GotoXY(c+y,r+z);Write(ciphblk[x,y,z]);
                end;
            Inc(c,(period+2)); if c>(4*(period+2)+1) then begin c:=1;r:=12;end;
            end;
    GotoXY(1,20); Write('Cipher Length = ',clast+1);
    any_key;
    end;
procedure assemble_solutions(numkeys:Integer);
    var n,p,s,w,v,x,y,shft   :integer;
    begin
    p:=0;
    for n:=0 to numkeys do
        for shft:=0 to period do begin {shift each key set}
           s:=0;
           for x:=0 to lastblk do begin
               if x=lastblk then v:=lastcol else v:=period;
               for y:=0 to v do begin
                   w:=(y+shft) mod (period+1); {add in shift factor}
                   case period of
                      3: solarray[p,s]:=ciphblk[x,y,allposs4[n,w]];
                      4: solarray[p,s]:=ciphblk[x,y,allposs5[n,w]];
                      5: solarray[p,s]:=ciphblk[x,y,allposs6[n,w]];
                      end; {of case statement}
                   Inc(s);
                   end;
               end;
           Inc(p); {index of solutions array}
           end;
    end;
procedure load_digraph_table;
          var
          digfile      :file of byte;
          rl,cl        :char;
          begin
          Assign(digfile,'digrams.dat');reset(digfile);
          for rl:='A' to 'Z' do
             for cl:='A' to 'Z' do
```

```
                 Read(digfile,table[rl,cl]);
          Close(digfile);
          end;
procedure display_solutions;
    label 5;
    var   c,x,y   :Integer;
          a       :Char;
    begin
    if sollast>39 then Textmode(CO80);
    TextBackground(Black);ClrScr;
    for x:=0 to numsols do begin
        c:=(x mod (period+1))+9;Textcolor(c);
        for y:=0 to sollast do Write(solarray[location[x],y]);
        Write(#10#13);
        if ((x mod 22)=0) and (x<>0) then begin
           message('Any Key to Continue, Esc for menu',White,1,24,false);
           a:=Readkey;
           if a=#27 then goto 5 else ClrScr; {Esc to return to main menu}
           end;
        end;
    any_key;
5:  Textmode(CO40);
    end;
procedure correlate;
    var numkeys  :Integer;
    begin
    numkeys:=factorial(period)-1;
    initialize; {set global variables for period chosen}
    fill_blocks;
    ClrScr;message('DECIPHERING',White,11,10,false);
    assemble_solutions(numkeys);
    message('CHECKING DIGRAPHS',White,11,12,false);
    digraph_check; {check all solutions for best English digraphs}
    message('SORTING SOLUTIONS',White,11,14,false);
    solbubblesort; {sort solutions in descending order of best digraphic score}
    Sound(500);message('FINISHED',White,11,16,false);Delay(1500);NoSound;
    corr_flag:=true;
    end;
procedure hardcopy(n:Integer);
    var x,y  :Integer;
    begin
    for x:=0 to n do begin
        for y:=0 to sollast do Write(LST,solarray[location[x],y]);
        Write(LST,#10#13);
        end;
    end;
begin         {main body of program}
    Textmode(CO40); {40 column color}
    exitflag:=false; {initialize Boolean flags}
    test4:=test41+test42; {period 4 test cipher assembled}
    test5:=test51+test52+test53; {period 5 test cipher assembled}
    test6:=test61+test62+test63+test64; {period 6 test cipher assembled}
    load_digraph_table;
    repeat
        TextBackground(Brown);ClrScr;Textcolor(Black);GotoXY(8,1);
        Writeln('SWAGMAN 6 CRYPTANALYSIS'^J^J^J^J);
        Writeln('(1) Enter Cipher'^J);
        Writeln('(2) Correlate Solutions for Period 6'^J);
        Writeln('(3) Correlate Solutions for Period 5'^J);
        Writeln('(4) Correlate Solutions for Period 4'^J);
        Writeln('(5) Display Cipher Blocks'^J);
        Writeln('(6) Hardcopy Top 20 Solutions'^J);
        Writeln('(7) Hardcopy All Solutions'^J);
        Writeln('(8) Display All Solutions'^J);
        Writeln('(0) Exit to DOS'^J);
        digit_in(sbn,White,10,24,selnum);
        case selnum of
             1 : begin
                 swagcipherin;
                 yes_no(itc,LightGreen,1,24,yesflag);
                 if yesflag=true then begin
                     squeeze(ualph,clast,intext);
                     periodcheck;corr_flag:=false; {not correlated yet}
                     end;
                 end;
             2 : begin period:=5;correlate;end;
             3 : begin period:=4;correlate;end;
             4 : begin period:=3;correlate;end;
             5 : display_blocks;
             6 : if corr_flag=true then hardcopy(19);
             7 : if corr_flag=true then hardcopy(numsols);
             8 : if corr_flag=true then display_solutions;
             0 : begin TextBackground(Black);ClrScr;TextBackground(Blue);
                       yes_no(qtp,White,7,12,exitflag);
                 end;
        end; {of case statement}
     until exitflag=true;
     Textmode(CO80);Textcolor(White); {80 column color}
end. {of program }
```

CIPHLIB.PAS

```
UNIT CIPHLIB; {Standard Cryptographic routines}
INTERFACE
USES Crt;
    const
    Maxlen = 600;
    upalph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    lowalph = 'abcdefghijklmnopqrstuvwxyz';
    digitstr = '0123456789';
    itc = 'Is This Correct?';
    sbn = 'Select By Number:';
    qtp = 'Quit This Program?';
    ipr = 'Is Printer Ready?';
    ualph : set of Char = ['A'..'Z'];
    lalph : set of Char = ['a'..'z'];
    digset: set of Char = ['0'..'9'];
    type
    txtarr=array[0..Maxlen] of Char;
    intarr=array[0..Maxlen] of Integer;
    str30=String[30];
    str6=String[6];
    charset=Set of Char;
    function Lowcase(a:Char):Char;
    function FileExists(filename :String):Boolean;
    procedure cipherin(test:String;Maxlen:Integer;
                       var clast:Integer;var cipher:txtarr);
    procedure yes_no(prompt:Str30;col,x,y:Integer;var a:Boolean);
    procedure show_value(name:Str30;col,x,y,v:Integer);
    procedure number_in(prompt:Str30;col,x,y,min,max:Integer;
                        var n:Integer);
    procedure digit_in(prompt:Str30;col,x,y:Integer;
                       var d:Integer);
    procedure any_key;
    procedure message(info:Str30;col,x,y:Integer;clr:Boolean);
    procedure squeeze(mask:charset;var clast:Integer;var cipher:txtarr);
    procedure char_to_int(clast:Integer;ctxt:txtarr; var cint:intarr);
    procedure ciph_str_in(test:String;var ciphstr:String);
    procedure squeeze_str(mask:charset;var str:String);
    procedure intswap(var x,y:Integer); {exchanges two integer values}
    procedure charswap(var a,b:Char); {exchanges two char values}
    procedure bubblesort(n:Integer;var narr1,narr2:intarr);
    procedure move_cursor(wid,lx,fr,lr,sp:Integer;var x,y:Integer);
    procedure morsecode(f:Char;var mstr:Str6;var pl:Char);
IMPLEMENTATION
procedure cipherin;
          var
          Instr :String; b,clth :Integer;
          begin
               ClrScr;Textcolor(LightGreen);GotoXY(1,5);
               Writeln('Enter Cipher, line by line');
               Writeln(MaxLen,' characters.');
               Writeln('Ctrl-x to restart, back to correct');
               Writeln('"test" for test cipher');
               Writeln('"/" to end entry');
               Textcolor(White);
               b:=1;instr:='';clth:=0;
               repeat
                     if b>Length(instr) then begin
                        b:=1;Readln(instr);
                        if (instr='test') or (instr='TEST') then instr:=test;
                        end;
                     if instr[b]<>'/' then begin
                        cipher[clth]:=Upcase(instr[b]);
                        Inc(clth);Inc(b); end;
               until (clth>MaxLen) or (instr[b]='/');
          clast:=clth-1;
          end;
procedure yes_no(prompt:Str30;col,x,y:Integer;var a:Boolean);
         var   z  :Char;  p: Integer;
         begin
         Textcolor(col);
         repeat
               GotoXY(x,y);ClrEol;Write(prompt,' (Y/N): ');
               z:=Readkey;p:=Pos(z,'YyNn');
         until p<>0;
         if p<3 then a:=true else a:=false;
         end;
procedure show_value(name:Str30;col,x,y,v:Integer);
      begin
      GotoXY(x,y);Textcolor(col);Write(name,v);
      end;
procedure number_in(prompt:Str30;col,x,y,min,max:Integer;var n:Integer);
          begin
          repeat
             Textcolor(col);GotoXY(x,y);Write(prompt,'(',min,'-',max,'): ');
             Readln(n);
          until (n>=min) and (n<=max);
          end;
procedure digit_in(prompt:Str30;col,x,y:Integer;var d:Integer);
          var    r   :Integer; s :Char;
          begin
          repeat
             Textcolor(col);GotoXY(x,y);Write(prompt);
             s:=Readkey;Val(s,d,r); if r<>0 then Write(#7);
          until r=0;
          end;
procedure any_key;
          var
          z  :Char;
          begin
          GotoXY(1,25);Textcolor(White);ClrEol;
          Write('Any Key to Continue');z:=Readkey;
          end;
procedure message(info:Str30;col,x,y:Integer;clr:Boolean);
    begin
    if clr=true then ClrScr;
    GotoXY(x,y);Textcolor(col);Write(info);
    if clr=true then any_key;
    end;
procedure squeeze(mask:charset;var clast:Integer;var cipher:txtarr);
    var
    x,y :Integer;
    begin
    y:=0;
    for x:=0 to clast do begin
        if cipher[x] in mask then begin
           cipher[y]:=cipher[x]; y:=y+1;
           end;
    end;
    clast:=y-1;
    end;
procedure char_to_int(clast:Integer;ctxt:txtarr;var cint:intarr);
    var   x :Integer;
    begin
    for x:=0 to clast do cint[x]:=Ord(ctxt[x])-65;
    end;
procedure ciph_str_in(test:String;var ciphstr:String);
    var    a,b,c,d       :Integer;
           instr         :array[0..11] of String;
           over          :Boolean;
    begin
    ClrScr;Textcolor(Lightgreen);GotoXY(1,5);
    Write('Enter Cipher String',#13#10,
          '255 Chars., 12 lines, max.',#13#10,
           'Ctrl-X to restart, backspace to correct',#13#10,
           '"test" for test cipher',#13#10,'"/" to end entry'#13#10#10);
    Textcolor(Blue);ciphstr:='';a:=0;over:=false;
    repeat
          Readln(instr[a]);if instr[a]='test' then instr[a]:=test;
          b:=Length(instr[a]);
          if instr[a,b]<>'/' then a:=a+1 else over:=true;
    until over=true;
    for c:=0 to a do ciphstr:=ciphstr+instr[c];
    Delete(ciphstr,Length(ciphstr),1); {remove /}
    end;
procedure squeeze_str(mask:charset;var str:String);
    var   x   :Integer;
    begin
    x:=1;
    while x<= Length(str) do
          if (str[x] in mask) then x:=x+1 else Delete(str,x,1);
    end;
function Lowcase(a:Char):Char;
    const  ualph:Set of Char = ['A'..'Z'];
    begin
    if a in ualph then a:=Chr(Ord(a)+32);
    Lowcase:=a;
    end;
function FileExists; {Boolean function that returns True if the file exists;
                     otherwise, it returns false; Closes file if it exists}
    var   f :file;
    begin  {$I-}
    Assign(f, Filename); Reset(f); Close(f); {$I+}
    FileExists:= (IOResult=0) and (filename<>'');
    end;
procedure intswap(var x,y:Integer);
    var    t   :integer;
    begin t:=x;x:=y;y:=t; end;
procedure charswap(var a,b:Char);
    var    t    :Char;
    begin t:=a;a:=b;b:=t; end;
```

```
procedure bubblesort(n:Integer;var narr1,narr2:intarr);
          {sorts two integer arrays in descending order of the first}
          {n is number of elements in each array}
    var    x    :Integer;
           flag   :Boolean;
    begin
    repeat
        flag:=true;
        for x:=0 to (n-1) do
            if narr1[x]<narr1[x+1] then begin
                intswap(narr1[x],narr1[x+1]);intswap(narr2[x],narr2[x+1]);
                flag:=false;
                end;
    until flag=true;
    end;
procedure move_cursor(wid,lx,fr,lr,sp:Integer; var x,y:Integer);
    var   a :Char;
    begin
    a:=Readkey;
    case a of
    #75: begin    {left arrow}
         Dec(x);if x<1 then begin x:=wid;Dec(y,sp);end;
         if y<fr then y:=lr;
         if (y=lr) and (x>lx) then x:=lx;
         end;
    #77: begin   {right arrow}
         Inc(x); if x>wid then begin x:=1;Inc(y,sp);end;
         if (y=lr) and (x>lx) then x:=lx;
         if y>lr then y:=fr;
         end;
    #72: begin   {up arrow}
         Dec(y,sp); if y<fr then y:=lr;
         if (y=lr) and (x>lx) then x:=lx;
         end;
    #80: begin   {down arrow}
         Inc(y,sp); if (y=lr) and (x<lx) then x:=lx;
         if y>lr then y:=fr;
         end;
    end; {of case statement}
    GotoXY(x,y);
    end;
procedure morsecode(f:Char;var mstr:str6; var pl:Char);
    const
    N = 43; {number of Morse characters in arrays}
    plain :array[0..N] of Char = ’etaonirshldcupfmwybgvkqxjz0123456789.,?:;-/=’;
    morse :array[0..N] of str6 = (’.’,’-’,’.-’,’---’,’-.’,’..’,’.-.’,’...’,
          ’....’,’.-..’,’-..’,’-.-.’,’..-’,’.--.’,’..-.’,’--’,’.--’,’-.--’,
          ’-...’,’--.’,’...-’,’-.-’,’--.-’,’-..-’,’.---’,’--..’,’-----’,’.----’,
          ’..---’,’...--’,’....-’,’.....’,’-....’,’--...’,’---..’,’----.’,
          ’.-.-.-’,’--..--’,’.-.-.-’,’---...’,’-.-.-.’,’-....-’,’-..-.’,
          ’-...-’);
    var  x   :Byte;
    begin
    if f='d' then begin
       x:=0;
       while x<=N do begin
           if mstr=morse[x] then begin pl:=plain[x];exit;end;
           Inc(x);
           end;
       end;
    if f='e' then begin
       x:=0;
       while x<=N do begin
          if pl=plain[x] then begin mstr:=morse[x]; exit;end;
          Inc(x);
          end;
       end;
    end;
END.
```

INTERNET MAILING LIST FOR ACA

Three dedicated ACA members have established the American Cryptogram Association Mailing List for those of the Krewe with Internet access. The mailing list is intended to keep ACA members in touch over the Internet computer network, with a major emphasis on solving cryptograms. The ACA is devoted to helping its members improve their knowledge of cryptography, and to make available materials and publications to assist in the study of cryptography.

DABASAP (Greg Griffin, `vlad@holonet.net`) will serve as the primary editor and will be responsible for screening messages from non-subscribers and for promoting appropriate discussion on the list.

PRIME (Dan Wheeler, `dan.wheeler@uc.edu`) will serve as the primary owner and will be responsible for maintenance of the mailing list and the handling of error messages.

NORTH DECODER (Jerry Metzger, `metzger@rs1.cc.und.nodak.edu`) will serve as Chair of the ACA-L Committee and will be responsible for co-ordination with the ACA Executive Board and with North Dakota State University.

NORTH DECODER also maintains an anonymous `ftp` and `gopher` site on a computer at the University. On a system with an `ftp` client, typing
`ftp rs1.cc.und.nodak.edu`
will connect you. Enter `anonymous` as the user name, and your Internet address as the password. Enter `cd pub/aca` to reach the appropriate files.

On a system with a `gopher` client, typing
`gopher gopher.cc.und.nodak.edu`
will give a menu, one option of which will be American Cryptogram Association, where a few infomative text files have been placed.

The same text files are available from an anonymous ftp site at `plains.nodak.edu` in the directory `pub/aca/info`.

FOUR MINI-REVIEWS

DAEDALUS

Seizing The Enigma; David Kahn; Houghton Mifflin; 1991. $24.95

This is, of course, not a new book, but I was prompted to review it when I found it remaindered by a number of mail-order bookstores and also on the 'reduced' table of my local Barnes and Noble at the ridiculously low price — for a hard cover title of this calibre — of nine dollars and change.

Heros and villains, backroom scientists in a stately home in England, adventure on the high seas, espionage (and even a love interest) are more normally found in novels — but this tale is the real thing! The Poles, French, British (and a little later the Americans) combined resources to crack the German Enigma machine. The effects of their successes on the critical Battle of the Atlantic are presented and evaluated. This volume is a real "can't put it down" attention holder as well as being an invaluable addition to the library of anyone interested in the Enigma machine and Ultra, the cryptanalysts' output. Impeccably written by master historian of the cipher and ACA member David Kahn and well illustrated (including several detailed photographs of an Enigma machine), this volume should be on your bookshelf at any price!

Applied Cryptography: Protocols, Algorithms, and Source Code in C; Bruce Schneier; John Wiley and Sons; 1993. $44.95

This work will act as both an introduction to modern cryptography for programmers and a programming reference for cryptographers whose interests lie in the security of data transmitted between, and stored in, computers. Data encryptation protocols and algorithms are presented in detail. The use (and abuse) of encryptation, digital signatures etc. are described by example as we follow Alice and Bob (and Mallet, Trent, and others) as they explore and exploit cryptology in its most advanced forms. Part Five of this book is an appendix containing the source code, in C, for much of the material presented earlier. For those who want a smoother path, a diskette containing all of the code included in the book - and more - is available from the author for $30. An invaluable reference that is eminently readable!

The Complete Modem Reference, 2nd Ed.; Gilbert Held; John Wiley and Sons; 1994. $34.95

This book may be read as a thorough introduction to the mysteries of data transmission/reception and online communications, or it may serve as an exhaustive reference to both the theoretical and practical aspects of modem use. Chapters on modulation standards and methods sit alongside the practical details of installing a modem in both the PC/XT/AT and PS2 series of IBM machines. The author has updated the earlier (1991) version of his work by including material dealing with v.32bis (14,400 bps) modems and associated error-correction and data-compression protocols and standards. A chapter on fax modems has been added. Some reference to the upcoming standards for 28,800 bps modems is made but the book falls short of "hard" information in this latest, still-developing area.

PGPSHELL v3.2; James Still; Shareware - usual BBS and ftp sources.

This is the latest version of the well-known front end for PGP, fully compatible with PGP v2.6 and all previous versions. Refinements over v3.1 include the ability to configure for the use of an editor other than the one built into the program and also to establish a subdirectory to store plaintext and ciphertext message files. For those who use PGP and find the command line inputs impossible to remember without a crib, this is what you need!

GW-BASIC.C CRACKING

Paul C. Kocher

```
/* BASCRACK.C

GW-BASIC for MS-DOS appears to encrypt a program using a substitution cipher
with period 143.  There is an 11-byte key and a 13-byte key that are used to
"protect" the program.  Running this program will attempt to crack that protection.
*/
#include <stdio.h>

int main(int argc, char **argv) {
     unsigned char key1[13]={
        0xA9,0x84,0x8D,0xCD,0x75,0x83,0x43,0x63,0x24,0x83,0x19,0xF7,0x9A};
     unsigned char key2[11]={
        0x1E,0x1D,0xC4,0x77,0x26,0x97,0xE0,0x74,0x59,0x88,0x7C};
     int nextbyte, index;
     unsigned char c;
     FILE *infile, *outfile;

     if (argc != 3) {
        printf("Utility to decrypt GWBASIC/BASICA files saved with \",p\"\n\n"
               "Copyright 1992 by Paul C. Kocher.  All rights reserved.\n\n"
               "Usage: BASCRACK encrypted.bas outfile.bas\n");
        exit(1);
     }

     if ((infile=fopen(argv[1],"rb"))==NULL || (outfile=fopen(argv[2],"wb"))==NULL) {
        printf("Error opening file.\n");
        exit(1);
     }

     if (fgetc(infile) == 0xFE) { fputc(0xFF, outfile); }
     else { printf("Not an encrypted BASIC file\n");
        exit(1);
     }

     index = 0;
     nextbyte=fgetc(infile);
     while (c=nextbyte, (nextbyte=fgetc(infile)) != EOF) {
        c -= 11 - (index % 11);
        c ^= key1[ index % 13 ];
        c ^= key2[ index % 11 ];
        c += 13 - (index % 13);
        fputc(c, outfile);
        index = (index+1) % (13*11);
     }
     fputc(c, outfile); /* Don't decrypt the EOF character */
     return 0;
}
```

HIGH PRECISION BASIC

Although BASIC is the language of choice for most Krewe programmers, the popular dialects lack sophisticated number-handling ability. A remedy for this deficit may be found in UBASIC, a BASIC language interpreter written by Professor Yuji Kida of the Department of Mathematics at Rikkyo University, Japan. UBASIC is uniquely powerful in a number of ways:

More than 230 keywords and commands.

Support for a wide range of integers from -65536^{542} to 65536^{542} which is a number in a little more than 2600 figures in decimal expression.

Support for rational and real numbers.

Support for complex numbers, up to 1300 digits in decimal notation, and including complex number versions of the following functions: EXP, LOG, SIN, COS, TAN, ATAN, SINH, COSH, SQRT, BESSELI, BESSELJ.

Includes the following operators: Logical Or, Logical And, Comparisons, Addition, Subtraction, Multiplication, Division, Integer Division, Rational Division, Remainder, Power.

Provides the ability to use "local" variables within subroutines.

Allows polynomials in one variable, including the following calculations: Addition, Subtraction, Multiplication, Division, Remainder, Power, Differential, Value.

Machine language programs can be put in predeclared arrays of Short variables and called by array names.

PRINT statements may be configured to automatically direct output to the screen, printer, and/or a file.

Allows a program to be stopped and continued at a later time. In ordinary BASIC, when you want to do another job while running a program which takes a lot of time, you have no choice but to abort the running program or to give up the new job. There is no satisfactory solution as DOS does not currently support multi-tasking. However, to FREEZE the running program is an effective solution.

1. Stop the running program by ctrl+BREAK or ctrl+c.
2. FREEZE the current status of memory on to a disk.
3. Start and finish another job.
4. MELT the frozen memory.
5. Continue the program by CONT.

The ability to compute a string if it represents a mathematical formula.

UBASIC includes may example programs, including:

PRTEST1 is an implementation of Lenstra's version of the Adleman-Pomerance-Rumely primality test algorithm. It is faster than simple-minded ones for integers of more than 12 to 13 figures. A 70-figure number can be tested in an hour. A present implementation can handle integers of up to 137 figures.

ECM, ECMX — factors integers using the Elliptic Curve Method. In a reasonable time, it can handle integers of more than 200 figures, but with a factor of at most 20 figures. ECMX, which uses a machine language routine, is faster by a few per cent.

MPQSX uses the Multiple Polynomial Quadratic Sieve method to factor integers of up to about 45 figures. If you have 32-bit machines (CPU is 386 or 486) with more than 1 megabyte extended memory and 10 to 100 megabytes of free hard disk space, MPQSHD will decompose up to 80 digits (however, it will take more than 1000 hours).

Version 8.65 is available in the `/public/ubasic` directory on **Decode**, the ACA BBS.

WHAT THE OTHER GUY IS DOING

[Editor's Note: Based on the feedback I get, WOGID is one of the most popular features of the *The Computer Supplement*. I've included some of the initial messages from the (electronic) ACA Mailing List, in the hope these longer messages will trigger some thoughtful responses and provide additional feedback. Be sure to read Larry Loen's excellent observations about computer languages, and GAMESTER's request for your opinions.]

DABASAP (Greg Griffin) continues to collect electronic mail addresses for ACA members. If you have an Internet, Compuserve or other electronic mail address, contact him at `vlad@holonet.net`. He is also instrumental in the set-up and operation of an ACA electronic mailing list. He is using his MacPlus to help solve aristos, patristos and railfence ciphers. He programs in Pascal and APL, and has just completed a C language class, so he'll get up to speed and start coding in C!

George Foot (`georgefoot@oxted.demon.co.uk`) writes:
[The] Internet can be valuable in facilitating correspondence between members both in the case of an exchange of views between two members or as a means of distributing information to all the members who can receive Internet postings.

An interest amongst ACA members in cryptographic methods of the type which computers have made possible is to be encouraged and Internet can assist in the study of such methods amongst ACA members by bringing them into touch with the large fraternity of other Internet subscribers who regularly discuss these topics in Internet newsgroups – which include newsgroups created specifically for discussions of cryptography.

NORTH DECODER (Jerry Metzger) is reachable via e-mail at `metzger@rs1.cc.und.nodak.edu`. He writes:
I've always enjoyed puzzles and games... crosswords (particularly cryptic crosswords), chess, go (both of which have remained enigmas to me), and of course, ciphers of all types. I program a bit in a variety of languages, but mostly I prefer pencil and paper solutions to crypts.

Besides being one of the owners of [the ACA mailing list] (with DABASAP and PRIME), my other contribution to the ACA is operating the aca drop box (that stands for the "assorted cryptographic articles drop box") on an [IBM UNIX] AIX system here at UND. This is an area where krewe members can both drop off and retrieve items dealing with the sorts of crypts found in *The Cryptogram*. If you would like details on how to login in to the aca drop box, drop me an e-mail message and i'll send you the login id and the password. (This is not an anonymous site; i'm trying to maintain a little control over access to minimize possible export violations of cryptographic item.)

I recently looked at PGP, found it interesting, and would like to pratice using it a bit. I'm sure that what appears straight forward when read is not so direct when actually put into practice. If anyone would like my public key, please drop me a note.

CORUM (John Zaharychuk) writes:
What I would like to do is 'play' with PGP. That means knowing a group of people with similar interests who don't mind the extra steps involved with using encryption and testing out whether this is a real world option. It also gives me an opportunity to generate/receive encrypted traffic at random.

For example, I am currently typing this on a Mac using MS Word. I'll then use MacPGP to generate my public key and to sign this message. I then need to paste the file into my mail program to send it out. Not the most convenient steps I would say but all I know right now. I am also not sure how easy it is to send out encrypted bulletins or newsletters.

In response to some of the other comments, PGP is available for the Mac and the keyrings are fully portable to at least the DOS and OS/2 environments.

Larry Loen (SHMOO, `lwloen@vnet.IBM.COM`**)** writes:

I have long been interested in computer cryptography – I began a lot of the ACA's work in this area with my participation in convetions back around '78. Use of computers was kind of "in the closet" up 'til then. Eventually, a lot more active members than I came out with things like the Computer Column and the Supplement, but I did a little bit by lobbying for both and participating in convention bull sessions that helped give birth to both (though the late MIKE BARLOW deserves, by far, the bulk of the credit for at least the latter).

I have contributed a decent handful of articles to both the Cryptogram and the Supplement over the years.

I don't do much amateur crypto these days, but I keep my membership up and still participate now and then. When my life slows down, I hope to again take it up in earnest.

I know from earlier work that ACA members are hopelessly diverse when it comes to their computers.

If it was manufactured, somebody has it. It is now probably less diverse than it used to be, but there would probably be a pretty representative split between the IBM/Clone crowd and Macs. Plus an amazing amalgam of die-hards owning C-64s, old Apples, and maybe even a TRS-80 or CP/M machine or two, though this is probably fading fast.

Language-wise, we are doubtless hoplessly diverse, also, but I think Pascal and ordinary Basic has so far dominated at least what is published.

As far as Windows goes, I think that's an elaboration we can probably do without. What most people are interested in is various kinds of aids to solving and a simple command-line interface is adequate. Even if you "must" have Windows, I could imagine Borland's EasyWin (where ordinary "command line" stuff is emulated in a spectacularly simple Window) would be fine for many purposes.

There may be a pathway, however, to have your cake and eat it too, if we go to an object-oriented, collaberative design.

For instance, I have worked out the beginnings of an object-oriented hierarchy that would enable almost any ACA system to be added to a common base. With this object-oriented base, it might be possible to separate the display portion from the cryptography portion without too much performance loss and without adding complexity (in fact, it might simplify things).

With this schema, it might be enough to have a standard Window shell that could adaptively deal with any system, adding systems over time. Any combination of manual and automatic solving methods could be potentially combined.

The downside is that some minimal C++ would be required to pull it off, but even that might be minimized. I personally think C++ will fairly rapidly displace C in any event; certainly, its advantages matter to us. The main issue is how fast ACAers will take up the language.

The upside is, that we can separate the efforts nicely. For example, not only would it be possible to have a command line/DOS text screen/Windows/Mac version all separated from the main-line code that creates and solves, say, Bifids, but it would also be possible for us to bundle up the printing support in such a way that simply adding a cipher to the base, with only a very, very, small amount of added work, could be used to prepare "cons", complete with two page output (one with just the crypt, one with the solution shown) as the section editors prefer.

Best of all, instead of everyone having to write everything, if the design were understood, we could divide and conquer the effort of writing systems. Once I wrote a system and added it, everyone would have my work for the price of a recompile.

That's the theory, anyway.

GAMESTER (Jim Glore) writes:
I have been an ACA member for about 5 years and have enjoyed tackling all areas of the CM. While my degree work was in Chemistry, I have been more involved with computers for the past 15 years, which has exposed me to Fortran, BASIC, assembly languages and now C. Along the way I hav had to cope with various operating systems such as VMS, UNIX, DOS and several extinct dinosaurs.

My computing interest intersected with cryptogram solving when I was running on an Apple II (minus), and I wrote several programs to aid in solving, some of which are for systems which are not currently represented in articles in the computer corner or the supplement (such as gromark). I will be translating the Applesoft programs to run on a PC and would be interested in some feedback from you all before I choose my path.

One choice would be to adapt the programs to run under Windows, using Virtual Basic or Virtual C. How many users are windows lovers/haters? A second choice would be to target DOS users and adapt into BASIC or C. There are pro's and con's for either OS choice as well as either language:

Programs for Windows are generally easier to write and can usually contain functions to pop up windows, for example to see the working version of the keyword solution. However, these are restricted to Windows users and the source code is not very portable to other environments.

DOS programs lack the sex appeal of pop up features and take a bit more effort, but the code can be provided easily and modified by the user to suit his/her tastes more easily. In fact, that is where I got some of my ideas and/or functions.

The question of BASIC vs C probably only makes sense if I decide to ignore Windows. Microsoft BASIC is fairly universal and easy to convert into other versions of BASIC. (Lord knows, if I could duplicate the listings from the Supplement in Applesoft BASIC, it could probably be done for *any* other dialect.) C, on the other hand, is not restricted to the PC environment and probably could be lifted directly into a UNIX machine with only minor problems.

So there you have it... the pro's and con's. Any feedback from the krewe would be most welcome. And the results of my work will eventually find their way into Dan V's BBS as well as the ACA drop box (provided North Decoder and I can figure out how to do that), and just as soon as I can find one of those elusive disks, the 'round tuit'.

DAEDALUS (David Hamer) has returned to the US after a four-year sojourn in Paris. To celebrate this event he bought a new machine, 486DX2/66-based, to assist in his cryptological efforts. His interest in both classical and modern cryptology, with current emphasis on Enigma, continues.
He also asks:
I need help from anyone who has worked through, or is still working on, C. Deavours book, *Breakthrough '32 - The Polish Solution of the Enigma*; Aegean Park Press (#51). I've had the book, and accompanying software for some time, but you know how it is ! Having worked the first three exercises (p.5) without difficulty and arrived at p.12-13, I am asked to run `ENIGMA.BAS`:

```
>...answering Y to the prompt "FAST ROTOR STEPS ONLY (Y OR NO)?"<
```

which I am unable to do because this prompt does not appear, nor is this line included in the BASIC code! Has anyone else had this problem; does anyone have a version of `ENIGMA.BAS` which includes this line ? (the opening screen of *my* version of the program is appended).

```
     GERMAN ARMY ENIGMA (CIRCA 1932)

ROTOR WIRINGS:

 1  : EKMFLGDQVZNTOWYHXUSPAIBRCJ
 2  : AJDKSIRUXBLHWTMCQGZNPYFVOE
 3  : BDFHJLCPRTXVZNYEIWGAKMUSQO

REVERSING ROTOR:  (AY) (BR) (CU) (DH) (EQ) (FS) (GL)
                  (IP) (JX) (KN) (MO) (TZ) (VW)

ENTER PATCH PANEL CONNECTIONS, CR TO TERMINATE (E.G. AU)
?
```

KARL (Waldo Boyd) responds to G4EGG's query from CS #18. QUE books programming series have a few available on QuickBASIC. *Using QuickBASIC 4* by Feldman and Rugg covers 4.5 quite well from a beginner's standpoint, and goes into advanced use around page 500. There's also *Programmer's Toolkit* with a disk by the same authors, and *Advanced Techniques* by Aiken. I devoured all three, and have occasionally found an article in Dr. Dobb's Journal, and other computer magazines on the newsstand. All three are superior, in my opinion, to the two that come with the PC version of Microsoft QuickBASIC 4.5.

NOTES FROM THE KEYBOARD

The September 1994 issue of *Communications of the ACM*, pp. 102–108 contains an article on solving cryptograms via what is basically a dictionary pattern search method. The author goes in to great detail about the theory behind the method, and offers 13 references to back it up. The C program, documentation, and data files are available on the ACA BBS.

A new version of Pretty Good Privacy (PGP) is available from various sources, including the ACA BBS. The new version is the result of collaboration between MIT and the RSA patent holders in an attempt to resolve some of the licensing issues. Reviews are mixed, as there are some that believe the new code has been crippled or compromised in some way. Stay tuned for details, as new versions seem to come about every couple of months, with intermediate "fixes" and "patches." I'm still using version 2.3a.

As noted in the *What the Other Guy is Doing* section, there are several members of the Krewe who would like to "practice" with PGP. Join the Internet mailing list or leave mail on the ACA BBS if you're interested in participating!

SECURING YOUR MS-DOS HARD DRIVE DATA

The ACA BBS now has two programs that allow a user to automatically encrypt some or all of the information on a PC hard drive. For those of you who are looking for a secure and relatively painless way to protect your data, these are two programs you might consider. The following are exerpts from the documentation.

Secure File System (`SFS110.ZIP`)

Ever since Julius Caesar used the cipher which now bears his name to try to hide his military dispatches from prying eyes, people have been working on various means to keep their confidential information private. Over the years, the art of cryptography has progressed from simple pencil-and-paper systems to more sophisticated schemes involving complex electromechanical devices and eventually computers. The means of breaking these schemes has progressed on a similar level. Today, with the ever-increasing amount of information stored on computers, good cryptography is needed more than ever before.

There are two main areas in which privacy protection of data is required:
- Protection of bulk data stored on disk or tape.
- Protection of messages sent to others.
SFS is intended to solve the problem of protecting bulk data stored on disk. The protection of electronic messages is best solved by software packages such as PGP (available on sites the world over) or various implementations of PEM (currently available mainly in the US, although non-US versions are beginning to appear).

SFS has the following features:

- The current implementation runs as a standard DOS device driver, and therefore works with both plain MSDOS or DRDOS as well as other software such as Windows, QEMM, Share, disk cacheing software, Stacker, JAM, and so on.

- Up to five encrypted volumes can be accessed at any one time, chosen from a selection of as many volumes as there is storage for.

- Volumes can be quickly unmounted with a user-defined hotkey, or automatically unmounted after a certain amount of time. They can also be converted back to unencrypted volumes or have their contents destroyed if required.

- The encryption algorithms used have been selected to be free from any patent restrictions, and the software itself is not covered by US export restrictions as it was developed entirely outside the US (although once a copy is sent into the US it can't be re-exported).

- SFS complies with a number of national and international data encryption standards, among them ANSI X3.106, ANSI X9.30 Part 2, Federal Information Processing Standard (FIPS) 180, Australian Standard 2805.5.2, ISO 10116:1991 and ISO 10126-2:1991, and is on nodding terms with several other relevant standards.

- The documentation includes fairly indepth analyses of various security aspects of the software, as well as complete design and programming details necessary to both create SFS-compatible software and to verify the algorithms used in SFS.

- The encryption system provides reasonable performance. One tester has reported a throughput of 250 K/s for the basic version of SFS, and 260 K/s for the 486+ version on his 486 system, when copying a file with the DOS copy command from one location on an SFS volume to another. Throughput on a vanilla 386 system was reported at around 160 K/s.

Secure Drive (`SECDRV13E.ZIP`)

Many people have sensitive or confidential data on their personal computers. Controlling access to this data can be a problem. PC's, and laptops in particular, are highly vulnerable to theft or unauthorized use. Encryption is the most secure means of protection, but is often cumbersome to use. The user must decrypt a file, work with it, encrypt it, and then wipe the plaintext. If encryption were easy, many more people would use it. SecureDrive is a step in this direction. SecureDrive automatically stores sensitive data on your DOS/Windows system in encrypted form.
SecureDrive V1.3 allows you to create up to four encrypted partitions on your hard drive(s). It also allows you to encrypt floppy disks. Encrypted partitions and disks become fully accessible when the TSR is loaded and the proper passphrase entered. The TSR takes only 2.7K of RAM, and can be loaded high. Encryption is performed at the sector level and is completely transparent to the application program. Everything on the disk or partitions except the boot sector is encrypted. Encrypted floppy disks can be freely interchanged with unencrypted ones. Disks and partitions can be decrypted and returned to normal at any time.

SecureDrive uses the IDEA cipher in CFB mode for maximum data security. The MD5 hash function is used to convert the user's passphrase into a 128-bit IDEA key. The disk serial number, and track and sector numbers are used as part of the initialization to make each sector unique.

NOTES TO AUTHORS

The *Computer Supplement* is intended as a forum to publish articles on the cryptographic applications of computers. We are always looking for submissions, but we ask potential authors to bear in mind:

1. Many readers are new to ciphers; please include a brief description of the cipher in question.
2. Many readers are new to computers; explain **why** you are using a computer as well as how.
3. Include the output of a typical run. If possible, build in an example for the reader to check the operation. Indicate how long it took to obtain this result.
4. Include a full description of how the program works, and back it up with comments in the listing.
5. Include a table of variables, either separately or as a part of the listing.
6. If at all possible, please submit *everything* in electronic form, either on a disk (any IBM format) or uploaded to the ACA BBS. This makes it much easier for us to typeset.
7. Send material for publication to Dan Veeneman, PO Box 2442, Columbia, Maryland, 21045–2442, USA.

ACA COMPUTER BULLETIN BOARD UPDATE

The ACA bulletin board system, `Decode`, and is available for both electronic mail (`dan@decode.com`) and file transfer, 24 hours a day at +1 410 730 6734. The following is a sample of some of the new cryptographic files added recently:

```
\public\aca
ultra.zip      16725   21-May-94   Info on Enigma after WW II
wart_c.zip      4786   01-Aug-94   Three analysis utilities (in C) from WART

\public\crypto
cryptbas.zip   16113   27-Jan-94   Basic cryptographic support programs (in C)
doub-des.txt   10323   15-Mar-94   Paper on Double DES encryption
factor.txt     15957   13-May-94   Dr. Ron Rivest on factoring
factor-r.c     10030   03-Mar-94   C program to factor RSA
gifford.cry     1552   29-Mar-94   Pointer to cracking Gifford cryptosystem
knuthrng.zip    4453   06-Mar-94   Code for some of Knuth's RNG algorithms
modrotor.zip   39870   02-Sep-93   Four rotor, 94 element C code
otp-10.zip     83559   24-May-94   One Time Pad
primes.txt      2944   17-Oct-89   Article about selecting primes for RSA
pwdcrypt.zip    9410   18-Apr-92   Discussion and code for DES encryption
random.zip    120522   03-Mar-94   Various pseudo-random number generators
randtest.c      2800   25-May-94   Test randomness of bit strings
rsafaq.zip     51202   13-Feb-94   FAQ on RSA Encryption, in TeX format
snefru.zip    215217   16-Feb-94   SNEFRU One-way hash encryption
vigenere.zip   54014   16-Feb-94   Vigenere system from Applied Cryptography
zipcrkpw.zip   20109   18-Mar-94   Utility to crack PKZIP encryption

\public\utility
idea.zip       32301   31-Mar-94   C implementation of the IDEA cryptosystem
idea22a.zip    14592   11-Aug-94   IDEA v2.2a for DOS
ideafast.txt   10398   11-Jan-94   Comment and code for faster IDEA on 80x86

\public\pgp
pgp26i.zip    259899   24-May-94   Executables for PGP v2.6 (Use at own risk)
pgp26src.zip  612090   23-May-94   Source code for MIT's PGP version 2.6
pgpshe32.zip  111948   07-Jul-94   Menu-driven front end for PGP
pwf20.zip      62766   15-Feb-94   PGP WinFront, Windows front end for PGP

\public\ubasic
malm.zip       37752   03-May-94   Numerical analysis programs for UBASIC
ubas865.zip   451290   03-May-94   UBASIC version 8.65
ubasic.rme      1250   03-May-94   Readme for UBASIC
ubmpqs32.zip   58629   03-May-94   Prime factorization for 32 bit UBASIC
```