In this issue:

SOLVING POLYALPHABETIC CIPHERS BY COMPUTER — D MELIORA has a Pascal program to solve polyalphabetics.

POLLUX REVISITED — BOATTAIL has some enhancements for his Pollux solver.

THE PLAYFAIR CIPHER — An introduction to the Playfair cipher, including a C language program.

AN AID TO MONOALPHABETIC SOLVING — A simple BASIC program to aid in manual monoalphabetic solving.

Plus: News and notes for computerists interested in cryptography, and cryptographers interested in computers.

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]


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

REFERENCE MATERIAL

BASICBUGS - Bugs and errors in GWBASIC (1p). [Also on Issue Disk #11].

BIBLOG — A bibliography of computer magazine articles and books dealing with cryptography (2p). (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].


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 prior to 13 have been exhausted, and are awaiting reprinting. Contact the Editor for current availability.

ISSUE DISKS

$5 per disk; specify issue(s), format and density required. All issues are presently available on two IBM High Density 1.2M disks, archived with PKZIP. For other disk formats, ask. Disk One — Issues 1 - 10; Disk Two — issues 11 to current. Disks contain ONLY 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.UUCP@uunet.uu.net
PO Box 2442 or uunet!anagld!decode!dan
Columbia, Maryland 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 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.
SOLVING POLYALPHABETIC CIPHERS BY COMPUTER

D MELIORA

This article describes a program for solving polyalphabetic cryptograms that use the Vigenère, Beaufort, or Variant encryption schemes. It is based on the material on Kasiski’s method contained in Gaines [1939] and in Kahn [1967]. The program is written in Turbo Pascal, Version 5 and will run on any computer that supports this language.

Programming Kasiski’s Method

In this discussion, I will take the Vigenère as my example, but the method can easily be adapted to the other two types. The first thing we need to know in cracking a Vigenère is the length of the keyword. Kasiski’s analysis helps us find this. It is based on the fact that in a long enough message, some repeated substrings in the plaintext will happen to be encrypted by the same key letters, resulting in repeated substrings in the ciphertext. The spacing between these repetitions will be a multiple of the length of the keyword. Hence we can obtain good guesses at the length of the keyword by looking for repeated substrings, tabulating the intervals between them, factoring the intervals, and seeing which factors come up most frequently. The key length will normally be one of these factors.

When we choose a factor \( f \), we then write the cryptogram as an \( x \)-by-\( f \) matrix where \( x \), the number of rows, depends on the length of the crypt. If \( f \) is the right number, then all the characters in any column have been encrypted by the same key letter. Furthermore, each column in a Vigenère cipher is a Caesar transposition of the corresponding letters in the plaintext. It is then only a matter of some detective work to decide what the key letter for each column is, and having the key we can then decrypt the message.

There are two important shortcuts which obviate a lot of the detective work. First, we can judge whether \( f \) is a good factor or not by computing the index of coincidence (\( \phi \), see Kahn, Chapter 12, on this) for each column. We compute this index for each column and compare it to two reference values, one for random letters (\( \phi_r \)) and one for English letter frequencies (\( \phi_p \); again, see Kahn). A column in which all the letters have been encrypted with the same key letter will tend to have a \( \phi \) that is close to \( \phi_p \); hence if all the columns have \( \phi \)'s close to \( \phi_p \), we have probably guessed the key length correctly.

The second shortcut is this: Since any column of a Vigenère, Beaufort, or Variant cryptogram is either a Caesar cipher or a reversed Caesar cipher, the likely key letter for that column can be found by doing a cyclical cross-correlation between the column’s letter frequencies and the letter frequencies of the English alphabet. This strategy works if there are roughly 15 or more letters in each column, but of course it isn’t always perfect. It’s best to note the three highest cross-correlations and search among the associated key letters for a probable key.

Note that the basic Kasiski method, with these two shortcuts, removes almost all the insight and detective work normally required in solving a polyalphabetic crypt. If you don’t mind doing the dogwork, you don’t need much in the way of brains. This means that we can push the whole task onto the computer, since computers are made for dogwork and procedures that don’t require much in the way of brains are easy to program. That is what I have done here.

The program attacks the cipher with three procedures named Kasiski, Evaluate, and Decrypt. Kasiski scans the ciphertext for repeated substrings, notes the intervals, factors them, and accumulates the factors. It displays the repetitions it found and a list of
factors and their frequencies. Before calling Evaluate, the main program prompts the user for a key length; respond by selecting one of the factors in the list. Evaluate then takes the selected key length, computes the ϕ values for each column, calls a procedure XCor to do the cross-correlations, and displays the results. Decrypt asks the user for a key and uses the key and the known encryption method to reconstruct the plaintext. An additional procedure, Display, shows the ciphertext and the plaintext as x-by-f matrices.

**Running the Program**

The best way to find out about the program is to compile and run it. The program will announce itself and prompt you for a filename. Enter the name of the file containing the cryptogram. For starters, we will use the example given at the end of this article. The program will list the ciphertext as contained in the file, will tell you the character count and the encryption method used. It then looks for repeated substrings and notes the intervals between them. It then displays all the factors of all the intervals, with their frequencies of occurrence, and asks you to choose one. The most probable key lengths are normally 5 to 12 characters, so this is where you should look. In our example, we get the following table of factors and frequencies:

| 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 |
|----------------------|------------------|------------------|------------------|------------------|------------------|
| 5 2 4 3 1 4 3 0 1 1 0 0 2 1 2 1 1 0 1 0 0 0 0 |
| 7 3 8 7 2 11 8 0 3 3 0 0 7 4 8 4 0 0 4 0 0 0 0 |

Since large factors naturally tend to occur less frequently than small ones, I have found, empirically, that the correct key length tends to show up more clearly if the frequencies are weighted by the square roots of the factors; the third line in the table gives these weighted frequencies. In our example, the most frequently occurring factors are 2 (too short), 4, and 7, and the weighted table clearly favors 7, so we guess that the key is 7 letters long.

The program then asks the user to select a factor for the key length and proceeds to evaluate this length by computing ϕ and the most likely key letters for each column. If ϕ is close to the English value, it’s likely that all the characters in the column were indeed enciphered with the same key letter; if the index is close to the random value, they probably weren’t. So the program gets a figure of merit for each column by computing the distance from the random value to the column’s ϕ and comparing that to the separation between the random and English values:

\[ \text{f.m.} = \frac{ϕ - ϕ_r}{ϕ_p - ϕ_r} \]

(Since ϕ_p and ϕ_r are only expected values, ϕ may fall outside this range and give you a figure of merit > 1 or < 0.) The program computes this ratio for each column. In our example, if we type 7, we get the following tabulation:
Col Frequencies (phi_r = 0.0385; phi_p = 0.0667) Likely

<table>
<thead>
<tr>
<th>Col</th>
<th>Frequencies</th>
<th>Likely</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a b c d</td>
<td>w x y z</td>
</tr>
<tr>
<td>1</td>
<td>1 0 1 0</td>
<td>0 0 1 4 0 0.096 2.024 u f y</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 1</td>
<td>2 0 1 0 0.029 -0.322 s d l</td>
</tr>
<tr>
<td>3</td>
<td>1 1 0 2</td>
<td>0 1 1 1 0.029 -0.322 t z p</td>
</tr>
<tr>
<td>4</td>
<td>1 1 2 0</td>
<td>0 0 0 3 0.110 2.456 i v y</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 0</td>
<td>0 0 2 1 0.066 0.981 n r y</td>
</tr>
<tr>
<td>6</td>
<td>1 1 0 0</td>
<td>2 0 0 2 0.066 0.981 o d h</td>
</tr>
<tr>
<td>7</td>
<td>0 0 2 1</td>
<td>1 0 0 0 0.075 1.294 v r u</td>
</tr>
<tr>
<td></td>
<td>Average f. of merit (near 1 =&gt; good): 1.026</td>
<td></td>
</tr>
</tbody>
</table>

The table shows the index of coincidence and the figure of merit for each column, and at the end it displays the average figure of merit. If the key length has been chosen correctly, this average will usually be near 1 and, occasionally, greater. In our example, it is 1.026, and we luck out: the most likely key letters for the seven columns are u-s-t-i-n-o-v, and that is the key.

When the program asks you for the keyword, type Ustinov. (Upper- or lower-case letters are both acceptable.) It then displays the ciphertext and plaintext in parallel matrices and you will be able to read Peter Ustinov’s words of wisdom in the plaintext. (If the matrices are so high that they fill the entire screen, then the display stops when the screen is full; hit <cr> for more.) After displaying the decrypted message, the program offers you a chance to write the solution to a file.

Now that we know the plaintext, we can see where the repetitions came from. Some of them are accidental, but many aren’t. The most conspicuous is the string −sdm− which appears twice. It came from the following coincidence:

...ustinovustinov...
...makemistakes...
....sdm......sdm....

The more repetitions of the keyword, the more likely such coincidences are. It follows that long keywords are better and that extremely long keys generated with the aid of a long-period pseudorandom number generator will give you the greatest security (except in the case of a known-plaintext attack). This, of course, is part of the basis of Vernam encryption.

Notice that you get several requests for confirmation. These permit you to re-try various parts of the program (for example, if you aren’t sure which of several factors to try, you can try them all and see how they look—what the confidence levels are and what the most probable key letters are). At the end of the whole works, it prompts you for another file to decrypt. You can keep going like this indefinitely. To get out of the program, enter a <cr> in response to the input file prompt.

The key may not appear in the first column of probable key letters, but the letters are usually all there in one column or another. You sometimes have to peer at the probable keys to decide whether there is a keyword there, and in very short messages there may be so little statistical information that the method breaks down altogether.

The method is not perfect, but even at its worst it provides enough information for you to make an intelligent decision, and occasionally the probable key letters will show that you are on the right track even when the φ’s are low. For example, No. 114 in Gaines, p. 126, a Beaufort crypt, yields these results:
This doesn’t look at all promising. The numbers suggest a key length of 5, but this yields garbage. When we try a probable key length of 7, we obtain this tabulation:

Col  Frequencies (phi_r = 0.0385; phi_p = 0.0667) Likely

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th></th>
<th></th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>Phi</th>
<th>f.m.</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0.052</td>
<td>0.492</td>
<td>pic</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.052</td>
<td>0.492</td>
<td>uhj</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0.052</td>
<td>0.492</td>
<td>zod</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0.062</td>
<td>0.830</td>
<td>mzq</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.029</td>
<td>-0.352</td>
<td>lhx</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.053</td>
<td>0.501</td>
<td>eti</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0.063</td>
<td>0.874</td>
<td>sol</td>
<td></td>
</tr>
</tbody>
</table>

Average fig. of merit (near 1 => good): 0.476

An average figure of merit of 0.476 is not at all encouraging, but from the likely keys it is pretty obvious that this is the correct length and that the key is puzzles.

The program does not accept a ciphertext from the keyboard. There is just too much room for mistakes that way and there is no easy way to edit the input. When creating a file for decrypting, start by putting the encryption method on a line by itself. Enter a V for Vigenère, a B for Beaufort, or an A for vArian. If you are attacking a crypt in which the method is unknown, enter a question mark. (If you forget to enter the method, Kasisk will ask you for it.) Then start the ciphertext on the next line. You can use either upper or lower case letters and you can group the ciphertext in any way you please. For the best looking output from Kasisk, put no more than about 75 characters on a line (including the blanks separating the groups). End the file with a <cr>; this is how Kasisk recognizes the end of the ciphertext. Remember to put the encryption method on the first line.

The maximum message length is 1,024 characters; input lines must be no more than 80 characters long; factors are limited to a range from 2 to 24; the longest keyword is 24 characters. The program is pretty robust; If you do anything bad, it will generally issue very apologetic diagnostic messages (like “I’ve probably run out of space.”) which should guide you.

**Summary**

It is rather surprising to see that the Vigenère cipher and its relatives can so easily be cracked by a computer with minimal human intervention. This contrasts strikingly with ordinary aristocrats which, although normally considered simpler than polyalphabetic ciphers, are notoriously hard for an unaided computer to crack. More than anything else, this program speaks for the brilliance of Kasiski’s insight and the unexpected weakness of the Vigenère technique.

The index of coincidence is most powerful when it is based on a large body of data. When the key is long, the columns are short and the index is less reliable. Matthews [1986] has proposed an alternative test based on frequency counts that appears to be more sensitive when the key is long. It might be interesting to include this test in the procedure Evaluate.
Appendix

The sample crypt (note the code V in the first line, for Vigenère):

```
V
CFMMY ZDAWG BBFII LPMNZ GGSDM ZWNNS DMFOI XHXZU OKMLA MVBOY
DEQTS INEBA GOFYK TZRHC YOHZF HWYUT CFSNI ENKUQ VLWYC YHCIM
ZPGVV MYHVR WINGM PRA
```

References


KASISK.PAS

Program KASISK;
{
Kasiski analysis of polyalphabetic ciphers (Vigenre, Beaufort, Variant).

Ciphertext is read from file, not keyboard. (If anything goes wrong, it's burdensome to have to reenter text again & again.) See input file format below.

Does Kasiski analysis & lists intervals between repeated substrings & factors of intervals.

Prompts user for factor to try.

For each factor, program shows letter frequencies for each column together with index of coincidence (phi) & (for reference) values of phi for random & monoalphabetic data. Does cross-correlation with English letter frequencies to determine three most likely key letters.

Program prompts user for acceptance & (if accepted) for keyword; displays ciphertext & attempted decryption in matrix form.

References: for Kasiski, Kahn, "The Codebreakers" & Gaines, "Elementary Cryptanalysis"; for index of coincidence, Kahn.

Text is represented as array of char in order to have entire text in a single data structure.

Program organization is essentially sequential:
Reads ciphertext, removes nonalphabets & converts to uniform case.
Does Kasiski analysis & displays factors found. For short messages, repeated substrings of length 2 are accepted; for longer messages, minimum acceptable substring length steadily increases. Evaluates any specified key length, finding letter frequencies for each column. Uses these frequencies to compute phis & probable key letters for each column. Normalizes & averages phis to get confidence level. Decrypts & displays text.

Weaknesses:
There ought to be some kind of editing capability so user can experiment with key interactively; don’t have good way of doing this yet.

Input file format:
First line: encryption method: V for Vigenre, B for Beaufort, A for Variant.
Rest of file: cipher text. Upper or lower case, grouped in any convenient way, max 78 chars/line. Last line must end in <cr>.

T. Parsons June 22, 1991
Revised May 29, 1993 Streamlined logic in a few places, improved tabulation from Evaluate, & added file output proc. Protected user if method omitted in first line of file.

uses crt;
const
max_lgth = 1024; { Max allowable length of cipher text } maxf = 24; { Largest factor considered } lastint = 100; { Number of intervals collected } phi_r = 0.0385; { Kappa for random characters } phi_p = 0.0667; { Plaintext kappa } maxk = maxf; { Longest possible key }

type fstring = string[80];
texttype = array [1..max_lgth] of char;
freqtype = array ['a'..'z'] of integer;

var
ciphr, { Ciphertext } clear: texttype; { Clear text } cc, method: char; infname: fstring; { Name of file containing crypt } fact, msg_lgth: integer; inf: text;

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

function LOCASE(c: char): char; { Force uniform lower case }
begin
  if c in ['A'..'Z'] then
    locase := chr(ord(c) + 32)
  else
    locase := c
end; { LoCase }

procedure READ_NEW; { Inputs new ciphertext }
{ Reads lines from user, converts them }
var { to upper case, forms into screen- }
  i, { sized lines }
  k, { Index into ciphr array }
  col: integer; { Column counter }
  lines: integer;
  cc: char; { Ciphertext character }
  input: string[80];{ Line image }
begin
  for k := 1 to max_lgth do
    begin { Wipe out old text, if any }
      ciphr[k] := ' '; 
      clear[k] := ' '; 
    end;
  k := 0;
  lines := 0;
  readln (inf, input);
  if (length(input) <= 2) and (upcase(input[1]) in ['V', 'B', 'A']) then
    begin
      method := upcase(input[1]);
      lines := 1
    end
  else
    method := '?';
  writeln ('Cipher text:');
  repeat { Until input read in } 
    if lines > 0 then { If method known, or }
    { past first line, } 
    readln (inf, input); { read a line }
    inc (lines);
    if length(input) > 0 then
      begin
        writeln (' ', input); { Display it }
        for i := 1 to length(input) do
          begin
            cc := locase(input[i]);
            if cc in ['a'..'z'] then
              begin
                writeln (' ', locase(input[i]));
              end;
          end;
      end;
  end;
begin { Squeeze out blanks }
  if k < max_lgth then
    inc (k);
    ciphr[k] := cc;
  end { for }
end { if }
until eof(inf);
msg_lgth := k;
close (inf);
if k = 0 then
  writeln ('No characters read.')
else if k >= max_lgth then
  begin
    writeln ('That was ', max_lgth, ' characters long or longer.',
    ' I''ve probably run out of space,');
    writeln (' but I have the first ', max_lgth, ' characters in',
    ' memory and can decrypt them if');
    writeln (' you want.')
  end
else
  writeln(msg_lgth,' characters.'); { Summarize }
write ('Encryption scheme: ');
case method of
  'V': writeln ('Vigenere');
  'B': writeln ('Beaufort');
  'A': writeln ('Variant');
  '?': writeln ('Unknown');
  else begin { This should never happen }
    writeln ('Not recognized.');
    writeln (' You may have omitted',
    ' the encryption code in Line 1.');
    method := '?'
  end
end { case }
end; { Read_New }

procedure KASISKI; { Kasiski analysis procedure. Finds repeated }
{ substrings in cipher text & intervals be- }
{ tween them. Probable period (key length) }
{ is factor of one or more of these intervals.}
var
  interval: array [1..lastint] of integer;
  factors: array [2..maxf] of integer;
  count, { Length of repeated substring }
  fact, { (Possible) factor }
  i,
  incent, { Indexes interval array }
  ist, jst, { Start of repetitions }
  ic, jc, { Index chars through reps }
  kmin: integer; { Min acceptable length }
begin
writeln ('Kasiski analysis:\');
writeln (' Interval String Interval String');
if msg_lgth > 800 then
  kmin := 5 { Min. acceptable }
else if msg_lgth > 400 then { length depends on }
  kmin := 4 { message length }
else if msg_lgth > 200 then
  kmin := 3
else
  kmin := 2;
for i := 1 to 100 do { Clear arrays }
  interval[i] := 0;
for fact := 2 to maxf do
  factors[fact] := 0;
inct := 0;
for ist := 1 to msg_lgth do
  for jst := ist + 2 to msg_lgth do { Search for repeated }
     begin { substrings (done }
       ic := ist; { by brute force }
       jc := jst;
       count := 0;
       while (jc <= msg_lgth) and (ciph[r][ic] = ciph[r][jc]) do
          begin
            inc (count); { Find length of }
            inc (ic); { substring }
            inc (jc)
          end;
       if count >= kmin then { Log intervals for }
          if inct < lastint then { non-trivial cases }
             begin
               inc(inct);
               interval[inct] := jst - ist;
               write (interval[inct]:6, '':8);
               for i := ist to ist + count - 1 do
                  write(ciph[r][i]);
               if odd(inct) then gotoxy (25, wherey)
                 else
                   writeln
              end
          end; { for ist, jst }
       if odd(inct) then writeln;
     end { for ist, jst } 
for fact := 2 to maxf do { intervals }
  if interval[i] mod fact = 0 then
    inc (factors[fact]);
  writeln ('Factors, frequencies,\ & weighted frequencies:');
  for i := 2 to maxf do
    write (i:3);
  writeln;
  for i := 2 to maxf do
    write (factors[i]:3);
writeln;
for i := 2 to maxf do
    write (round(sqrt(i)*factors[i]):3);
writeln;
end; { Kasiski }

procedure XCOR (var freqs: freqtype; var maxc, maxc2, maxc3: char);
{ Does cross-correlation between column frequencies & English letter frequen- }
{ cies to find most likely key letters. Used by Evaluate, below. }
const natfrqs: freqtype =
    (80, 16, 32, 36,123, 23, 16, 51, 72, 1, 5, 40, 22, { These are English }
    72, 79, 23, 2, 60, 66, 96, { letter frequencies }
    31, 9, 20, 2, 13, 1);
var
    i, j,
    lag: integer; { Current offset }
    lmax: { Offsets with max corr }
    array[1..3] of integer;
    corr: real; { Current correlation }
    cmax: { Three largest corrs. }
    array[1..3] of real;
    c1, c2: char; { Loop indices }
begin
    cmax[3] := 0;
    for lag := 0 to 25 do
        begin
            corr := 0;
            for c1 := 'a' to 'z' do { Compute correlations }
                begin
                    j := ord(c1) + lag - ord('a');
                    if j > 25 then j := j - 26;
                    if method = 'B' then { Special handling }
                        j := 25 - j; { for Beaufort }
                    c2 := chr(j + ord('a'));
                    corr := corr + freqs[c1]*natfrqs[c2];
                end;
            i := 3;
            while i >= 1 do { Update three greatest }
                begin
                    if corr > cmax[i] then
                        begin
                            for j := 1 to i - 1 do
                                begin { Shift others down }
                                    lmax[j] := lmax[j + 1];
                                    cmax[j] := cmax[j + 1];
                                end;
                            cmax[i] := corr;
                            lmax[i] := lag;
                            i := 0; { Quick exit from loop }
                        end;
                    i := i - 1;
                end;
        corr := corr + freqs[c2]*natfrqs[c2];
    end;
end;
end; { if }
dec (i)
end { while }
end; { for lag }
case method of { Find corresponding }
'V': for i := 1 to 3 do { key letters }
begin
lmax[i] := 26 - lmax[i];
if lmax[i] > 25 then lmax[i] := lmax[i] - 26;
end;
'B': for i := 1 to 3 do
begin
lmax[i] := 25 - lmax[i];
if lmax[i] < 0 then lmax[i] := lmax[i] + 26;
end;
'A': begin end { Ok as is }
end; { case }
maxc := chr(ord('a') + lmax[3]);
maxc2 := chr(ord('a') + lmax[2]);
maxc3 := chr(ord('a') + lmax[1]);
end; { XCor }

procedure EVALUATE (period: integer); { Assesses this period length }
{ Finds letter frequencies for each }
{ column together with index of coin- }
{ cidence & most likely key letters. }
var
freqs: freqtype; { Letter frequencies }
col, { Current column }
col_hgt, { Number of chars in col}
i: integer;
delta, { Normalized distance }
{ from phi_r to phi }
phi, { Index of coincidence }
sig, { Figure of merit }
sum: real; { Sum of deltas }
c,
cmax, { Most likely keys }
cmax2, cmax3: char;
begin
writeln; { Write headings }
writeln ('Col Frequencies', '(phi_r =':31, phi_r:7:4, '; phi_p =',
phi_p:7:4, ')', 'Likely':8);
write ('':4);
for c := 'a' to 'z' do
write (c:2);
writeln (' Phi f.m. Keys');
sum := 0;
for col := 1 to period do
begin
write (col:2, ' ');
freqs[c] := 0;
i := col;
col_hgt := 0;
while i <= msg_lgth do { Accumulate } 
begin { letter frequencies }
inc (freqs[ciphr[i]]);
inc (i, period);
inc (col_hgt)
end;
phi := 0; { Compute index }
for c := 'a' to 'z' do { of coincidence }
phi := phi + freqs[c]*(freqs[c] - 1);
phi := phi/(col_hgt*pred(col_hgt));
delta := (phi - phi_r)/(phi_p - phi_r);
sum := sum + delta;
xcor (freqs, cmax, cmax2, cmax3);
for c := 'a' to 'z' do { Show results }
write (freqs[c]:2);
writeln (phi:8:3, delta:7:3, cmax:4, cmax2:2, cmax3:2);
end;
sig := sum/period;
writeln ('Average fig. of merit (near 1 => good):', sig:7:3);
end; { Evaluate }

procedure DECRYPT (period: integer; method: char);
var { Gets keyword from user & } 
i, j, { decrypts message }
ic, ik, ip: integer;
keyst: string[maxk];
begin
repeat { Get & check keyword }
write ('Key: '); 
readln (keyst);
if length(keyst) <> period then
writeln ('Huh???');
until length(keyst) = period;
for i := 1 to period do { Convert to lower case }
keyst[i] := locase(keyst[i]);
for i := 1 to period do { Decrypt }
begin
j := i;
while j <= msg_lgth do
begin
ic := ord(ciphr[j]) - ord('a');
ik := ord(keyst[i]) - ord('a');
case method of
 'V': ip := ic - ik; { These are inverses of } 
 'B': ip := ik - ic; { defining equations } 
 'A': ip := ic + ik; { for encryption } 
end; { case }
end;
if ip < 0 then ip := ip + 26 { Keep within range } 
else if ip > 25 then ip := ip - 26;
clear[j] := chr(ip + ord('a'));
inc (j, period)
end
end; { Decrypt }

procedure DISPLAY (period: integer; { Show ciphertext & cleartext }
var ciphr, clear: texttype); { as matrices }
var
  i, j,
  linecount: integer;
  c: char;
begin
  i := 0;
  linecount := 0;
  while i <= msg_lgth do
    begin { Ciphertext on left }
      for j := i + 1 to i + period do
        if j <= msg_lgth then write (ciphr[j]:2)
        else write (' ':2);
      write ('':6); { Plaintext on right }
      for j := i + 1 to i + period do
        if j <= msg_lgth then write (clear[j]:2)
        else write (' ':2);
      writeln;
      inc (i, period);
      inc (linecount);
      if linecount mod 22 = 0 then
        begin
          write ('<cr> for more');
          readln
        end
    end; { Display }
end;

procedure WRITE_OUT;
var
  i: integer;
  outfname: fstring;
  outf: text;
begin
  if open_out (outf, outfname) then
    begin
      for i := 1 to msg_lgth do
        begin
          write (outf, clear[i]);
          if i mod 65 = 0 then
            writeln (outf)
        end;
    close (outf)
end
end; { Write_Out }

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

begin
  lowvideo;
  writeln;
  writeln ('This is Kasisk [5-21-93].');
  while open_in (inf, inname) do
    begin
      read_new;
      if method = '?' then
        repeat
          write (' Select V[igenere], B[eaufort], or [v]A[riant]: ');
          readln (cc);
          method := upcase(cc);
        until method in ['A', 'B', 'V'];
        if method in ['A', 'B', 'V'] { Need this test in case }
          then { file written incorrectly }
            begin
              kasiski; { Find probable periods }
              repeat
                repeat
                  write ('Select a factor (0 terminates): ');{ $I-}
                  readln (fact);
                  {$I+}
                until ioresult = 0;
                if fact > 0 then
                  begin { Show results for }
                    evaluate (fact); { this factor }
                    if agree ('Ok?') then
                      begin { Get keyword & decrypt }
                        decrypt (fact, method);
                        { Show results }
                        display (fact, ciphr, clear);
                        if agree ('Ok?') then
                          begin
                            fact := 0; { Quick getaway }
                          end
                        if agree ('Write sol to file?') then
                          write_out
                          end
                      end
                    end
                  end
                end
              until fact = 0
            end { if method }
        end; { while }
    writeln ('Done.');
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;

OPEN_IN.INC

{ File opening procedure. }

Directions for use:
Declare OPEN_IN as a forward reference in calling program:
function OPEN_IN (var inf: file;
    var fname: fstring): boolean; forward;
& follow immediately with an include directive.
This permits declaring inf & fname as any desired type.
Revised   1-5-88   Made a function
1-8-88   Arranged for removal of option switches

function OPEN_IN;
var
    i: integer;
    work: string[64];
    found: boolean;
begin
    repeat
        write ('Input File [or con:]: ');
        readln (fname);
        if (fname = '') or (fname = 'con:') then found := true
        else
            begin
                work := fname;
                i := pos ('/', work);
                if i > 0 then work := copy (work, 1, i - 1);
            {$I-}
            assign (inf, work);
            reset (inf);
function OPEN_OUT;
  var _reply: string[16];
  _accept: boolean;
begin
  repeat
    write ('Output File: ');
    readln (fname);
    if fname = '' then
      _accept := true
    else
      begin
        {$I-}
        assign (outf, fname);
        reset (outf);
        {$I+}
        _accept := ioresult <> 0;
        if not _accept then
        begin
          write ('File already exists. Okay to overwrite? ');
          readln (_reply);
          _accept := (length(_reply) > 0) and (_reply[1] in ['y', 'Y'])
        end;
      end;
    end;
  until _accept;
end;

OPEN_OUT.INC

{ File opening procedure.

Directions for use:
  Declare OPEN_OUT as a forward reference in calling program:
  function OPEN_OUT (var outf: text;
    var fname: fstring): boolean; forward;
  & follow immediately with an include directive.
  This permits declaring outf & fname as any desired type.

Returns FALSE if user gave null string as file name; this
permits no-output option if desired.

}
Pretty Good Privacy version 2.3a has been released. It is currently available from archives in source and executable form. From the documentation:

**Synopsis:** PGP uses public-key encryption to protect E-mail and data files. Communicate securely with people you’ve never met, with no secure channels needed for prior exchange of keys. PGP is well featured and fast, with sophisticated key management, digital signatures, data compression, and good ergonomic design.

**Quick Overview:** Pretty Good™ Privacy (PGP), from Phil’s Pretty Good Software, is a high security cryptographic software application for MSDOS, Unix, VAX/VMS, and other computers. PGP allows people to exchange files or messages with privacy, authentication, and convenience. Privacy means that only those intended to receive a message can read it. Authentication means that messages that appear to be from a particular person can only have originated from that person. Convenience means that privacy and authentication are provided without the hassles of managing keys associated with conventional cryptographic software. No secure channels are needed to exchange keys between users, which makes PGP much easier to use. This is because PGP is based on a powerful new technology called “public key” cryptography.

PGP combines the convenience of the Rivest-Shamir-Adleman (RSA) public key cryptosystem with the speed of conventional cryptography, message digests for digital signatures, data compression before encryption, good ergonomic design, and sophisticated key management. And PGP performs the public-key functions faster than most other software implementations. PGP is public key cryptography for the masses.
NOTES FROM THE KEYBOARD

I am now settled into my new home on the east coast (see the inside front cover for new address and BBS information). Just a few miles from here is Fort Meade, the base for the National Security Agency. Antennas and barbed-wire fences are the order of the day over there.

Cryptography has been in the news several times recently. In April, the Clinton Administration announced a new “standard” for secure telephone communications: the Clipper chip. This chip, while encrypting voices over a telephone line, would also allow law enforcement officials to decrypt any conversation, ostensibly pursuant to a court order. Civil libertarians and those concerned about privacy immediately raised the alarm, especially after the Administration hinted that, if Clipper didn’t become widespread, they would ban other forms of encryption.

Phil Zimmerman, the original author of *Pretty Good Privacy*, was recently served a subpoena to produce all records and documents regarding his program. A Federal Grand Jury in California is apparently interested in finding out how his program made it out of the United States — current US munitions law appears to prohibit the export of cryptographic software. PGP is now wide-spread in the US, Europe, Asia and even in the former Soviet Union.

A large amount of space in this *Computer Supplement* is dedicated to program listings. While few readers will type in all these programs (it would be easier to get the Issue Disk), I hope the listings serve to show methods and approaches to solving problems via computer. One of the best ways to improve programming skills is by example, and we’re trying to provide plenty of examples.

Enjoy the issue, and good solving!

---

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.
POLLUX REVISITED

BOATTAIL

1. Enter Cipher Digits. Enter the cipher digits in strings with no spaces between them. The program will accept a maximum of 600 characters in the input. If your cipher is longer than that, discard the last part.

Each input string is converted immediately into characters in an array. When you have finished the cipher type / as the last character to tell the program that you are done. The program asks Is this correct?. If you type Y the cipher is entered into the working arrays, if you select N, the cipher is discarded.

If you enter test (all lower case) instead of the cipher, the program loads the internal test cipher. This is useful for checking on modifications and for learning how the program functions.

2. Find Separators. The program will try all possible sets of 3 or 4 digits as separators (x’s) and eliminate all that are impossible. When the program asks How many Separators? type in 3 or 4 as the cipher tells you. It is barely possible that there will be two possible sets of separators.

The program displays the possible set(s). If there is only one, that set is automatically processed and the morse text is partially filled in. If there is more than one set, then the sets are displayed and you are asked to choose which one will be filled in.

3. Display Cipher. The ciphertext, morse text, and plaintext are displayed in lines. The maximum display is seven sections of three lines each for a total of 560 characters. If your cipher is over 560, the rest is not displayed.

4. Change Separators. This option clears the morse and plain text arrays and fills in a new set of separator digits. This is rarely used. There is almost always only one possible set.

5. Substitute in Cipher. The cipher, morse, and plain arrays are displayed along with a selection panel at the lower left of the screen. For the test cipher, the panel looks like this:

```
0123456789
 xx x
```

Examine the cipher and select substitutes by trial and error. Use the left and right arrow keys to move the cursor along the bottom row of the panel. If the cursor is under 2 and you type - then the - is entered for all the 2’s in the cipher array. If you type a space, the substitute under the 2’s will be erased.

As you enter substitutes, the program scans the morse array and looks for complete letter sequences. The complete sequences are decoded and the plain letter is filled into the plain array. The plaintext is thus produced.

Hint: Look for xx.x....x.xx. This sequence can only be “the” and you can fill in the dots and dashes to make xx-x....x.xx. A similar attack on short words is the easiest way to solve a Pollux.

To exit the substitution module, hit the Escape key and you are returned to the main menu. When you return, the text arrays are not changed and you can return to them and continue solving.

6. Hardcopy. Selecting this option produces a copy of the solving screen on your printer. The program is written for an Epson FX printer and uses its control codes. If your printer is not Epson-compatible, this module may not work for you. You will have to manually copy the solution off the screen. If you have a Turbo Pascal compiler, you can modify the control codes to suit your printer.
program POLLUX;
{
Written in Borland’s Turbo Pascal 4.0.
Determines separator digits, by trial & error on every combination
from 012 to 789 in procedures ‘compsep3’ for 3 separators and
‘compsep4’ for 4 separators (0123 to 6789).
Puts separators into morse text, displays cipher on screen for solving
plaintext on screen by trial & error.
Assumes a color monitor. If you are using a monochrome display,
delete all TextColor commands.
Printer Commands are for an Epson FX dot matrix printer. If your
printer is not Epson-compatible, change or delete the command
strings in procedure ‘hardcopy’
}
USES Crt,Printer,Ciphlib;
{uses ‘morsedecode’ procedure and others from CIPHLIB.TPU, a library
of standard cipher routines in Turbo Pascal 4.0}
CONST
Maxlen=600; {Maximum Characters in input strings}
Maxdisplay=559; {Maximum characters displayed, 7 lines of 80, 0-559}
test1='345994612789635075921618092784034650796210678921861435349’;
test2='52976018527630853607218186249594357306047289108672510756943’;
test3='832486107529591436813278046905947521361384928765059074623918’;
test4='182476950125679804078362107534652196431827018627050347526191/’;
{Plain = Most widely used word in the world is ok say language ...}
{E-8 NOV-DEC 1991 by RIG R. MORTIS}
TYPE
line = (num,morse,plain);
VAR
exitflag :Boolean;
test :String; {test cipher text}
test1 :String; {test cipher text}
test2 :String; {test cipher text}
test3 :String; {test cipher text}
test4 :String; {test cipher text}
test5 :String; {test cipher text}
test6 :String; {test cipher text}
exitflag :Boolean;
test :String; {test cipher text}
test1 :String; {test cipher text}
test2 :String; {test cipher text}
test3 :String; {test cipher text}
test4 :String; {test cipher text}
test5 :String; {test cipher text}
test6 :String; {test cipher text}
exitflag :Boolean;
test :String; {test cipher text}
test1 :String; {test cipher text}
test2 :String; {test cipher text}
test3 :String; {test cipher text}
test4 :String; {test cipher text}
test5 :String; {test cipher text}
test6 :String; {test cipher text}
}
var x :byte;
begin
for x:=0 to numsep-1 do begin opsep[x]:=sepsol[n,x];ddx[opsep[x]]:='x';end;
end;

procedure morsefill; {fill morse text from dot-dash equivalents}
var x,y :Integer;
begin
FillChar(txt[morse],Sizeof(txt[morse]),' '); {clear morse text}
for x:=0 to 9 do
for y:=0 to clast do txt[morse,y]:=ddx[cnum[y]];
end;

procedure dispall; {display cipher, morse & plain}
var ln :line;
begin
Textmode(CO80);ClrScr;for ln:=num to plain do displayline(ln);
end;

procedure choosesep; {which set of computed separators do you want to try?}
var w :Integer; flg :Boolean;
begin
if solnum=0 then begin
message('Only One Possible Set',Cyan,9,20,false);any_key;end
else begin
digit_in('Which Separator Set? ',Blue,1,20,w);Write(w);
yes_no(itc,White,1,24,flg);
if flg=true then sepfill(w);
end;

procedure compsep3; {deduce three separators by trying all combinations}
var b,i,j,k,n,sc,nsc :Integer;
label 650,660,670;
begin
b:=0;
for i:=0 to 7 do
for j:=i+1 to 8 do
for k :=j+1 to 9 do begin
GotoXY(18,12);Write(i,' ',j,' ',k);sc:=0;nsc:=0;
for n:=0 to clast do begin
if (cnum[n]=i) or (cnum[n]=j) or (cnum[n]=k) then goto 650;
if nsc=4 then goto 670 else begin Inc(nsc);sc:=0;goto 660;
end;
end;
650: if sc=2 then goto 670 else begin Inc(sc);nsc:=0;end;
660: end; {of for n}
sepsol[b,0]:=i;sepsol[b,1]:=j;sepsol[b,2]:=k;Inc(b);
670: end; {of for k}
solnum:=b-1; {number of separator solutions}
end;
procedure compsep4; {deduce four separators by trial}
var  b,h,i,j,k,n,sc,nsc,x :Integer;
label 750,760,770;
begin
b:=0;
for h:=0 to 6 do
  for i:=h+1 to 7 do
    for j:=i+1 to 8 do
      for k:=j+1 to 9 do begin
        GotoXY(18,12);Write(h,' ',i,' ',j,' ',k);
        sc:=0;nsc:=0;
        for n:=0 to clast do begin
          x:=cnum[n];
          if (x=h) or (x=i) or (x=j) or (x=k) then goto 750;
          if nsc=4 then goto 770 else begin
            Inc(nsc);sc:=0;goto 760; end;
        750: if sc=2 then goto 770 else begin Inc(sc);nsc:=0;end;
        760: end; {of for n}
        sepsol[b,0]:=h;sepsol[b,1]:=i;sepsol[b,2]:=j;sepsol[b,3]:=k;
        Inc(b);
      770: end; {of for k}
solnum:=b-1;
end;
procedure displaysep; {display deduced separator set(s) on screen}
var  f,g :Integer;
begin
ClrScr;Textcolor(LightMagenta);GotoXY(1,6);Write('Separators:');
for f:=0 to solnum do begin
  GotoXY(15,6+f);Textcolor(Cyan);Write(f,' ');Textcolor(LightGreen);
  for g:=0 to numsep-1 do Write(sepsol[f,g],' ');end;
end;
procedure decode; {change morse text to plain}
var  x :Integer; strflag,blankflag :Boolean; xstr :Str6;
begin
FillChar(txt[plain],Sizeof(txt[plain]),' ');
strflag:=false;blankflag:=false;xstr:='';
for x:=0 to clast do
  case txt[morse,x] of
    'x' : begin
      if (strflag=true) and (blankflag=false) then begin
        morsecode('d',xstr,txt[plain,x-1]);
        strflag:=false;
      end;
      blankflag:=false;xstr:='';
    end;
    '.' , '-' : begin xstr:=xstr+txt[morse,x];
      strflag:=true;end;
    '' : begin strflag:=false;blankflag:=true;xstr:='';end;
  end;
procedure instruct; {display instructions while using substitution screen}
begin
  GotoXY(1,22);Textcolor(LightGreen);Write('0123456789');
  Textcolor(LightCyan);
  GotoXY(1,24);Writeln('Use right and left arrows to move cursor');
  Write('Insert dot, dash, or blank below digit; Escape for Main Menu');
  Textcolor(Red);
end;

procedure substitute; {interactive, trial & error solving on screen}
var x,indx :Integer; a,b :Char; outflag :Boolean;
begin
 ClrScr;dispall;instruct;indx:=0;outflag:=false;
  repeat
    GotoXY(1,23);TextColor(LightCyan);for x:=0 to 9 do Write(ddx[x]);
    GotoXY(indx+1,23);
    b:=Readkey;
    case b of
      ' ','.','-' : begin ddx[indx]:=b;morsefill;decode;
                    displayline(morse);displayline(plain);end;
      #0 : begin a:=Readkey;
            case a of
              #77 : indx:=(indx+11) mod 10; {left arrow}
              #75 : indx:=(indx+9) mod 10; {right arrow}
            end;
        end;
      #27 : outflag:=true; {escape key}
    end;
    until outflag=true;
end;

procedure hardcopy; {printout cipher, morse, plain, dot-dash}
const dwon=#0#27#87#49#13; {NUL,Esc,W1,CR} {Double Width On}
  dwoff=#27#87#48#13; {Esc,W0,CR} {Double Width Off}
  wid=40; {40 chars. per printed line}
var f,g,h :Integer; out :Boolean; ln :line;
begin
  Write(Lst,dwon); {double width printing on, CR}
  g:=0;h:=wid;out:=false;
  repeat
    if h>clast then begin h:=clast;out:=true;end;
    for ln:=num to plain do begin
      for f:=g to h do Write(Lst,txt[ln,f]);
      Write(Lst,#10#13); {LF,LF,CR}
    end;
    Inc(g,wid);Inc(h,wid);Write(Lst,#10); {blank line between rows}
    until out=true;
  Writeln(Lst,'0123456789');
  for f:=0 to 9 do Write(Lst,ddx[f]);
  Write(Lst,dwoff,#13#12); {double width printing off,CR,Form feed}
end;

begin {main body of program}
test:=test1+test2+test3+test4; {test cipher assembled}
exitflag:=false;
repeat
    Textmode(CD40); {40 column color}
    ClrScr;Textcolor(LightBlue);GotoXY(10,1);flag:=false;
    Writeln('POLLUX CRYPTANALYSIS');Textcolor(Green);
    GotoXY(14,2);Writeln('By BOATTAIL''J);Textcolor(LightCyan);
    Writeln('(1) Enter Cipher Digits''J);
    Writeln('(2) Find Separators''J);
    Writeln('(3) Display Cipher''J);
    Writeln('(4) Change Separators''J);
    Writeln('(5) Substitute in Cipher''J);
    Writeln('(6) Hardcopy''J);
    Writeln('(0) Exit to DOS''J);
    digit_in(sbn,White,10,24,selnum);
    case selnum of
        1 : begin
            cipherin(test,Maxlen,clast,intext);
            yes_no(itc,White,1,24,flag);
            if flag=true then begin
                squeeze(['0'..'9'],clast,intext);
                ciphfill;
            end;
        end;
        2 : begin
            ClrScr;
            number_in('How Many Separators? ',Red,7,12,3,4,numsep);
            if numsep=4 then compsep4 else compsep3;
            sepfill(0);morsefill;displaysep;any_key;
        end;
        3 : begin dispall;any_key;end;
        4 : begin displaysep;choosesep;morsefill;end;
        5 : substitute;
        6 : hardcopy;
        0 : begin ClrScr;yes_no(qtp,Red,7,12,exitflag);
            end;
    end; {of case statement}
until exitflag=true;
end. {of program}
THE PLAYFAIR CIPHER

Charles Shapiro

Introduction
The playfair cipher was invented in 1854 by Charles Wheatstone, the physicist who was also responsible for the measurement tool known as the Wheatstone Bridge. But it bears the name of Lyon Playfair, a British politician who tried to get his foreign office to adopt it. The playfair cipher is a good medium-security cipher system for alphabetic text. You can easily encipher and decipher messages manually with it, and enciphered messages are more secure than they are under a monalphabetic code (such as the Caesar cipher), and easier to encipher and decipher than a book code message.

The playfair cipher works on letter pairs; this makes it less vulnerable to solution by frequency analysis, since the most common letter pairs in English (th and he) comprise a much smaller portion of the possible letter pairs than the most common letters do of the possible letters. It also means that a garbled code group affects only two letters of the message you are trying to send.

Using the playfair cipher
To encipher a message with the playfair cipher, you must first construct the key grid. This grid must be a square; usually, “i” and “j” are equivalent in encoded text so that the key grid is 5 letters on a side. To construct the grid, first think of a key phrase or word with no duplicated letters in it. In our example, we will use the phrase “man bites dog”.

To construct a playfair grid from the key phrase, first write down the key phrase in 5 letter rows, so:

<table>
<thead>
<tr>
<th>m</th>
<th>a</th>
<th>n</th>
<th>b</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>e</td>
<td>s</td>
<td>d</td>
<td>o</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next, fill in the rest of the grid with the letters of the alphabet not found in the phrase:

<table>
<thead>
<tr>
<th>m</th>
<th>a</th>
<th>n</th>
<th>b</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>e</td>
<td>s</td>
<td>d</td>
<td>o</td>
</tr>
<tr>
<td>g</td>
<td>c</td>
<td>f</td>
<td>h</td>
<td>k</td>
</tr>
<tr>
<td>l</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>u</td>
</tr>
<tr>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>

Next, divide your message into letter pairs, inserting a null letter (such as “x”) wherever a letter pair consists of two of the same letter or the message ends without another letter:

i see no tourists only soldiers
is ex en ot ou ri st so nl ys ol di er sx

The second group and the last group in this sample message contain nulls to make encoding it possible.

You can find every letter in the message on the code grid. Each pair of letters on the grid can stand in only three relationships to each other: they can share a grid row, they can share a grid column, or they can share neither a row nor a column. To encipher a pair of letters which share a row, write down the grid letter to the right of each member of the pair. To encipher a pair of letters which share a column, write down the grid letter below each member of the pair. In these cases, the grid wraps around; hence, the letter to the right of “i” in the code grid given above is “m”. If the pair of letters to encipher share neither a row nor a column, their enciphered equivalents are...
the letters which are in the same row as the
ciphered letter, and the same column as its
mate.

Hence, to encipher the letter pair “is”, you will
first find “i” and “s” in the cipher grid, above:

\[
\begin{array}{cccc}
m & a & n & b \\
t & e & s & d \\
\end{array}
\]

The enciphered letter pair is “no”, as follows:

\[
\begin{array}{cccc}
m & a & N & b \\
t & e & s & d \\
\end{array}
\]

The full enciphered text of the message given
above with the key “man bites dog” is as fol-
lows:

\[
is\ ex\ en\ ot\ ou\ ri\ st\ so\ nl\ ys\ ol\ di\ er\ sx
no\ sw\ sa\ te\ kz\ ub\ de\ dt\ mq\ xd\ tu\ ob\ dp\ fn
\]

To decipher an encoded message, simply re-
verse the process explained above: take each
coded pair of letters, find them in your grid,
and look at their relationship. If they share
a row, the plaintext letters are to their left; if
the share a column, the plaintext letters are
above them. If they share neither, the plain-
text letter lies in the same row as the code
letter and the column of its partner.

For added security, you can “transpose” your
playfair key before constructing the code grid.
To transpose a key, write it out and write the
rest of the alphabet beneath it:

\[
\begin{array}{cccc}
m & a & n & b \\
t & e & s & d \\
\end{array} \\
c & f & h & k & l & p & q & r & u & v & w \\
x & y & z
\]

Construct your code grid by reading down this
primary grid, as follows:

\[
\begin{array}{cccc}
m & c & x & a \\
y & n & h & z \\
k & i & t & p \\
e & q & s & r \\
u & o & v & g \\
w
\end{array}
\]

This method eliminates the regular structure
of the grid for letters not in your key phrase.

Once you have spent about 15 minutes to mas-
ter this cipher system, you can encipher and
decipher quickly with pencil and paper.

**Playfair.c**

I have written **PLAYFAIR.C** to automate the
process of enciphering and deciphering with

The playfair cipher. It is limited in the same
ways as the example cipher used above. En-
ciphered messages contain only lowercase let-
ters; numbers, punctuation, and white space
in plaintext are ignored.

**PLAYFAIR.C** takes two arguments: an option
string and a key string. The key string must be
enclosed by quotes; duplicate letters, spaces,
and punctuation marks are automatically re-
moved from it. A - must precede the option
string. At this writing, there are three op-
tion strings: -p prints out a copy of the play-
fair cipher table, -d decodes a message with
the argument key, and -t transposes the arg-
ument key as explained above. Options can
be combined in logical orders – -pt prints a
transposed table, -dt decodes using a trans-
posed table. All I/O happens through stdin
and stdout.

**Conclusion**

I wrote this program for fun and entertain-
ment, so I expect no renumeration for its use.
If you make enhancements to the code, please
let me know about them. Please note that
I will accept no responsibility for the use or
misuse of this encipherment method or this
program. I also make no specific claims about
the security or correctness of the **PLAYFAIR.C**
code or any version of the playfair program.
PLAYFAIR.C

/*
 name: playfair.c

 syntax: playfair [- d | p] [-i] "key string" < infile > outfile

 description: Encipher or decipher text files using the playfair cipher
 "-d" deciphers text; "-p" prints the playfair cipher table
 for the key phrase. "-t" transposes the playfair key table.

 Author: Charles Shapiro 23 November 1988

 This source code is truly public domain; you may use it for fun or profit,
 sell it for whatever the market will bear, or make any modifications to
 it you wish. I ask as a matter of courtesy that you credit me with writing
 it if you plan to distribute it or a modification of it.

 */
#include <stdio.h>
#include <ctype.h>

#define ROW1 0
#define COL1 1
#define ROW2 2
#define COL2 3

char codetbl[5][5];

static char alphabet[] = {"abcdefghijklmnopqrstuvwxyz"};

int print_sw;
int decode_sw;
int transpose_sw;

/*
 This is where we check that we've got valid key and option arguments,
 and set whatever options need to be set.
 */
int args_ok(arc,arv,keyptr)
    int arc;
    char *arv[];
    char **keyptr;
{
    int i;

    if((arc > 4) || (arc < 2))
        return 0;

    *keyptr = 0;
    print_sw = 0;
    decode_sw = 0;
transpose_sw = 0;
for(i=1;i<arc;i++) {
    if( *(arv[i]) == '-' ) {
        switch(tolower(arv[i][1])) {
        case 'd':
            decode_sw = 1;
            if(strlen(arv[i]) > 2) {
                if(tolower(arv[i][2]) == 't')
                    transpose_sw = 1;
                else {
                    fprintf(stderr,\"Bad option argument - %c\", arv[i][2]);
                    exit(1);
                }
            } else {
                *keyptr = arv[i];
            }
        break;
        case 'p':
            print_sw = 1;
            if(strlen(arv[i]) > 2) {
                if(tolower(arv[i][2]) == 't')
                    transpose_sw = 1;
                else {
                    fprintf(stderr,\"Bad option argument - %c\", arv[i][2]);
                    exit(1);
                }
            } else {
                *keyptr = arv[i];
            }
        break;
        case 't':
            transpose_sw = 1;
            break;
        default:
            printf(\"Bad option argument - %c\n\", arv[i][1]);
            return(0);
        }
    } else {
        *keyptr = arv[i];
    }
    return(-1);
}
/*
Remove spaces, duplicate letters from key.
*/
void purify_key(char *inp, char *out)
{
    int i, j, dup;
    int in_len;
    char *out_ptr;
    in_len = strlen(inp);
    }
out_ptr = outbuf;
if(isalpha(*inbuf)) {
    *out_ptr = tolower(*inbuf);
    ++out_ptr;
}

for(i=1;i<=in_len;i++) {
    if(! isalpha(inbuf[i]))
        continue;
    dup = 0;
    for(j=0;j<i;j++)
        if(tolower(inbuf[j]) == tolower(inbuf[i]))
            dup=1;
    if(! dup) {
        *out_ptr = tolower(inbuf[i]);
        if(*out_ptr == 'j')
            *out_ptr='i';
        ++out_ptr;
    }
    *out_ptr=0;
}

/*
Make the playfair key table.
*/
void make_keytbl(char *codekey)
{
    char *tablptr;
    int i;

    strncpy((char *)codetbl,codekey,25);
    tablptr = (char *)codetbl + strlen(codekey);
    for(i=0;i<25;i++) {
        if(! strchr((char *)codetbl,alphabet[i])) {
            *tablptr = alphabet[i];
            ++tablptr;
        }
    }
}

/*
Print out the key table for manual encipher/decipher.
*/
void print_keytbl()
{
    int i;
    for(i=0;i<25;i++) {
        printf("%c ",(char *)codetbl[i]);
    }
if( ! ((i+1) % 5))
  putchar(‘\n’);
}
/*
Get a letter from stdin. Make it lowercase. If it’s a j, make it an i.
*/
int get_letter()
{
  int c;

  c = getchar();

  while((! isalpha(c)) && (c != EOF))
    c = getchar();

  c = tolower(c);
  if(c == 'j')
    c = 'i';
  return(c);
}
/*
Get next pair of letters from stdin, substituting nulls (X’s) where
appropriate.
*/
int get_pair(in_pair)
char *in_pair;
{
  static int in_chars[2] = {0,2};
  int retval=0;

  if(in_chars[0] == EOF)
    goto exit_point;

  if(in_chars[0] != in_chars[1]) {
    in_chars[0] = get_letter();
    if(in_chars[0] == EOF)
      goto exit_point;
    else
      in_chars[1] = get_letter();
  } else
    in_chars[1] = get_letter();

  if(in_chars[1] == EOF) {
    *in_pair = (char)*in_chars;
    in_pair[1] = 0;
    in_chars[0] = EOF;
    retval = 1;
    goto exit_point;
  }
}
if(in_chars[0] == in_chars[1]) {
    in_pair = (char*)in_chars;
    in_pair[1] = 0;
    retval = 1;
    goto exit_point;
}

in_pair[0] = (char)in_chars[0];
in_pair[1] = (char)in_chars[1];
retval = 1;

exit_point:
    if(retval) {
        if(! (in_pair[1])) {
            if(in_pair[0] != 'x')
                in_pair[1] = 'x';
            else
                in_pair[1] = 'y';
        }
    }
    return(retval);
}

/*
Find a letter in the key grid.
*/
void find_letter(letter,row,col)
    char letter;
    int *row;
    int *col;
{
    int i,j;

    *row = EOF;

    for(i=0;i<5;i++) {
        for(j=0;j<5;j++) {
            if(codetbl[i][j] == letter) {
                *row = i;
                *col = j;
            }
        }
        if(*row != EOF)
            break;
    }
}

/*
Encipher a message with the playfair cipher.
*/
void encode_playf(inpair,outpair)
    char *inpair;
    char *outpair;

{  
    int thecase;
    int inpair_coords[4];
    int outpair_coords[4];

    find_letter(inpair[0],&inpair_coords[ROW1],&inpair_coords[COL1]);
    find_letter(inpair[1],&inpair_coords[ROW2],&inpair_coords[COL2]);

    if(inpair_coords[ROW1] == inpair_coords[ROW2]) /* Same row. */
        thecase = 0;
    else
        if(inpair_coords[COL1] == inpair_coords[COL2]) /* Same column */
            thecase = 1;
        else
            thecase = 2; /* same neither */

    switch(thecase) {
    case 0: /* same row. */
        outpair_coords[ROW1] = inpair_coords[ROW1];
        outpair_coords[ROW2] = inpair_coords[ROW2];
        if(inpair_coords[COL1] == 4)
            outpair_coords[COL1] = 0;
        else
            outpair_coords[COL1] = inpair_coords[COL1] + 1;
        if(inpair_coords[COL2] == 4)
            outpair_coords[COL2] = 0;
        else
            outpair_coords[COL2] = inpair_coords[COL2] + 1;
        break;
    case 1:
        outpair_coords[COL1] = inpair_coords[COL1];
        outpair_coords[COL2] = inpair_coords[COL2];
        if(inpair_coords[ROW1] == 4)
            outpair_coords[ROW1] = 0;
        else
            outpair_coords[ROW1] = inpair_coords[ROW1] + 1;
        if(inpair_coords[ROW2] == 4)
            outpair_coords[ROW2] = 0;
        else
            outpair_coords[ROW2] = inpair_coords[ROW2] + 1;
        break;
    case 2:
        outpair_coords[ROW1] = inpair_coords[ROW1];
        outpair_coords[COL1] = inpair_coords[COL2];
        outpair_coords[ROW2] = inpair_coords[ROW2];
        outpair_coords[COL2] = inpair_coords[COL1];
        break;
    }

    outpair[0] = codetbl[outpair_coords[ROW1]][outpair_coords[COL1]];
    outpair[1] = codetbl[outpair_coords[ROW2]][outpair_coords[COL2]];
}
void decode_playf(inpair, outpair)
    char *inpair;
    char *outpair;
{
    int thecase;
    int inpair_coords[4];
    int outpair_coords[4];

    find_letter(inpair[0], &inpair_coords[ROW1], &inpair_coords[COL1]);
    find_letter(inpair[1], &inpair_coords[ROW2], &inpair_coords[COL2]);

    if(inpair_coords[ROW1] == inpair_coords[ROW2]) /* Same row. */
        thecase = 0;
    else
        if(inpair_coords[COL1] == inpair_coords[COL2]) /* Same column */
            thecase = 1;
        else
            thecase = 2; /* Same neither */

    switch(thecase) {
        case 0: /* Same row. */
            outpair_coords[ROW1] = inpair_coords[ROW1];
            outpair_coords[ROW2] = inpair_coords[ROW2];
            if(! inpair_coords[COL1])
                outpair_coords[COL1] = 4;
            else
                outpair_coords[COL1] = inpair_coords[COL1] - 1;
            if(! inpair_coords[COL2])
                outpair_coords[COL2] = 4;
            else
                outpair_coords[COL2] = inpair_coords[COL2] - 1;
            break;
        case 1:
            outpair_coords[COL1] = inpair_coords[COL1];
            outpair_coords[COL2] = inpair_coords[COL2];
            if(! inpair_coords[ROW1])
                outpair_coords[ROW1] = 4;
            else
                outpair_coords[ROW1] = inpair_coords[ROW1] - 1;
            if(! inpair_coords[ROW2])
                outpair_coords[ROW2] = 4;
            else
                outpair_coords[ROW2] = inpair_coords[ROW2] - 1;
            break;
        case 2:
            outpair_coords[ROW1] = inpair_coords[ROW1];
            outpair_coords[COL1] = inpair_coords[COL2];
            outpair_coords[ROW2] = inpair_coords[ROW2];
outpair_coords[COL2] = inpair_coords[COL1];
break;
}
outpair[0] = codetbl[outpair_coords[ROW1]][outpair_coords[COL1]];
outpair[1] = codetbl[outpair_coords[ROW2]][outpair_coords[COL2]];
}
/*
Invert a key.
*/
char *transpose_key(key,inbuf)
char *key; /* the original key. */
char *inbuf; /* The original key with the rest of the alphabet. */
{
int keylen = strlen(key);
char *inbufptr;
char *outbufptr;
int i;
static char transposed[128];
char key_buf[128];

memset(key_buf,0,sizeof(key_buf));
strncpy(key_buf,inbuf,25);
outbufptr = transposed;
for(i=0;i<keylen;i++) {
inbufptr = &(key_buf[i]);
while(*inbufptr) {
*outbufptr = *inbufptr;
++outbufptr;
inbufptr += keylen;
}
}
return(transposed);
}
/*
Main line.
*/
main(argc,argv)
int argc;
char *argv[];
{
char working_key[255];
char *key;
char input_pair[3];
char output_pair[3];
void (*process)();
int output_counter;
if(! args_ok(argc,argv,&key)) {
fprintf(stderr,"\nUsage: playfair [-[d | p]] [-i] "key string"");

exit(1);
}

purify_key(key,working_key);
make_keytbl(working_key);
if(transpose_sw)
    strncpy(codetbl,transpose_key(working_key,(char *)&codetbl),25);
if(print_sw) {
    print_keytbl();
    exit(0);
}
if(decode_sw)
    process = decode_playf;
else
    process = encode_playf;
output_counter = 0;
while(get_pair(input_pair)) {
    (*process)(input_pair,output_pair);
    output_pair[2]=0;
    output_counter += 3;
    if(output_counter > 78) {
        printf("%s\n",output_pair);
        output_counter = 0;
    } else
        printf("%s ",output_pair);
} putchar(’\n’);

ABOUT THIS ISSUE

This issue was produced using Eberhard Mattes’ excellent Em\TeX \version \of Donald Knuth’s \TeX \typesetting \program, \with \help \from \various \style \files, \especially \texttt{multicol.sty} \and \texttt{fullpage.sty}.

The \texttt{.tex} \file \that \produced \this \issue \is \about 117,000 \bytes. \It \was \edited \and \processed \on \an \IBM \PC \clone \using \a \simple \ASCII \text \editor (MKS \Toolkit’s \texttt{vi}). \It \was \printed \on \a \Hewlett-Packard \LaserJet \IIIP.
AN AID TO MONOALPHABETIC SOLVING

This is a simple program to aid in the manual solution of monoalphabetic ciphers. A cipher text file is read in to an integer array (C(,),) and a frequency count is computed.

The program then waits for the user to enter a cipher text letter and the plain text letter that should correspond. The program then updates the map arrays and re-displays the cipher text, line by line, with the plain text appearing below. This repeats until the user presses the ESC (escape) key to exit. In this way the user can manually observe different selections and try different assignments until the cipher text is made plain.

The program makes use of map arrays, which are simply two arrays that keep track of plain text/cipher text correspondence. The MC array maps cipher characters to plain characters, that is, MC(cipher letter) = plain letter. The MP array does the inverse, i.e. MP(plain letter) = cipher letter. In this program, the uppercase letters A, B, etc. are mapped to the numbers 1, 2, etc. For example, MC(1) would give the plain letter that corresponds to the cipher text letter A.

MONOHELP.BAS

1000 ' MONOHELP.BAS
1010 ' Assistance for solving monoalphabetic ciphers by trial and error
1020 ' Variables:
1030 ' C(,) Cipher text, in ASCII
1040 ' P(,) Plain text, in ASCII
1050 ' F() Frequency counts for each of the 26 letters
1060 ' MC() Map index of cipher to plain letters
1070 ' MP() Map index of plain to cipher letters
1080 ' CC Cipher character (ASCII)
1090 ' CP Plain character (ASCII)
1100 ' IC Index into MC() array (cipher)
1110 ' IP Index into MP() array (plain)
1120 ' OP Old index pointer into either array (temporary)
1130 DIM C(6,80), P(6,80), F(26), MC(26), MP(26)
1140 ' INITIALIZE
1150 ' Set all letter counts and map indexes to zero
1200 FOR I=1 TO 26
1210 F(I) = 0
1220 MC(I) = 0
1230 MP(I) = 0
1240 NEXT I
1250 FOR I = 1 TO 6 ' Six lines
1260 FOR J = 1 TO 80 ' 80 characters on each line
1270 C(I,J) = 32 ' Set all cipher text to space (ASCII 32)
1280 P(I,J) = 32 ' Set all plain text to space (ASCII 32)
1290 NEXT J
1300 NEXT I
1310 ' LOAD DATA FILE
1320 ' PRINT "Filename";
1330 INPUT F$
1340 OPEN F$ FOR INPUT AS #1
1370 NL = 0
1380 ' Read in each line of text from the file
1400 IF EOF(1) THEN 1660
1420 LINE INPUT #1, A$
1430 NL = NL + 1
1440 NC = 0
1450 ' Examine each character in the line of text
1470 FOR I=1 TO LEN(A$)
1480 ' We can only handle lines with 80 characters or less
1490 IF I > 80 THEN 1640
1500 C=ASC(MID$(A$,I,1))
1520 ' If letter is lower case (ASCII between 97 and 122), make it
1530 ' uppercase by subtracting 32
1540 IF C >= 97 AND C <= 122 THEN C = C - 32
1550 ' If the character is an upper case letter, bump up the frequency count
1560 IF C >= 65 AND C <= 90 THEN F(C-64) = F(C-64) + 1
1570 IF C < 32 THEN C = 32
1580 NC = NC + 1
1590 IF NC < 6 THEN 1410
1600 CLOSE #1
1610 ' Display the frequency chart
1620 FOR I = 1 TO 26
1630 T$=STR$(F(I))
1640 PRINT CHR$(64+I):"":";T$;SPACE$(6-LEN(T$));
1650 IF I = 7 THEN PRINT
1660 IF I = 14 THEN PRINT
1670 IF I = 21 THEN PRINT
1680 NEXT I
1690 ' Display the letter map index
1700 FOR I = 1 TO 26
1710 T$=STR$(F(I))
1720 PRINT CHR$(64+I);"=";
1730 IF I = 7 THEN PRINT
1740 IF I = 14 THEN PRINT
1750 IF I = 21 THEN PRINT
1760 NEXT I
1770 PRINT
1780 ' Display the letter map index
1790 FOR I = 1 TO 26
1800 PRINT CHR$(64+I);"=";
1810 IF MC(I) = 0 THEN 1850
1820 ' Letter is mapped, so display what it maps to
1830 PRINT CHR$(64+MC(I));
1840 GOTO 1870
1850 ' Letter isn't mapped, so leave a space
1860 PRINT " ";
1870 PRINT " ";
1880 IF I = 13 THEN PRINT
1890 NEXT I
1900 PRINT
1910 ' Display the cipher text, and the current plaintext below it
1920 FOR I = 1 TO NL
1930 PRINT
1940 FOR J=1 TO 80
1950 PRINT CHR$(C(I,J));
1960 NEXT J
1970 PRINT
1980 FOR J=1 TO 80
1990 PRINT CHR$(P(I,J));
2000 NEXT J
2010 PRINT
2020 NEXT I
2030 PRINT
2040 PRINT "Replace Cipher Letter: ";
2050 GOSUB 2740
2060 IF CK = 32 THEN 2050
2070 IF CK = 27 THEN 2830
2080 CC = CK
2090 PRINT CHR$(CC);" with Plain Letter: ";
2100 GOSUB 2740
2110 IF CK = 27 THEN 2830
2120 IF CK <> 32 THEN 2220
2130 ' Plain letter selected was a space (ASCII 32), so remove
2140 ' any mapping information for the cipher letter
2150 ' any mapping information for the cipher letter
2160 IC = CC - 64
2170 IP = MC(IC)
2180 MC(IC) = 0
2190 MP(IP) = 0
2200 GOTO 2490
2210 '
2220 CP = CK
2230 PRINT CHR$(CP)
2240 '
2250 IC = CC - 64
2260 IP = CP - 64
2270 '
2280 IF MP(IP) = 0 THEN 2410
2290 ' The plaintext letter is already used
2300 PRINT CHR$(CP);" is already mapped from ";CHR$(64+MP(IP));
2310 PRINT ", move to ";CHR$(CC);" (Y/N)? ";
2320 GOSUB 2740
2330 IF CK = 27 THEN 2830
2340 IF K$ = "N" THEN 2680
2350 IF K$ <> "Y" THEN 2320
Zero indexes for old mappings

IF Cipher letter is already used, clear the old plain text index

Generate plain text from cipher text and mapping array

FOR I = 1 TO NL
FOR J = 1 TO 80
Default to whatever is in the cipher position

select the cipher letter at this location

IF this character isn’t an uppercase letter, skip over it

assume there is no letter mapped, and make the plain letter a blank

convert it to an index into the map array

There is a plain letter mapped to this cipher letter, so

replace the cipher letter with corresponding plain character

MC(IC) = IP
MC(IC) = IC
IF MP(IP) > 0 THEN MP(IP) = 0
MC(IP) = IC
MP(IP) = IC
MP(IP) = 0
OP = MC(IC)
IF OP > 0 THEN MP(OP) = 0
MC(IP) = 0
OP = MC(IC)

IF CC < 65 OR CC > 90 THEN 2650
IC = CC - 64
IF IC = 0 THEN 2650
IP = IC
CP = IP + 64
P(I,J) = CP
FOR I = 1 TO NL
FOR J = 1 TO 80
P(I,J) = C(I,J)
CC = C(I,J)
IF CC < 65 OR CC > 90 THEN 2650
P(I,J) = 32
IC = CC - 64
IF IC = 0 THEN 2650
IP = IC
CP = IP + 64
P(I,J) = CP
NEXT J
NEXT I

K$ = INKEY$
IF K$ = "" THEN 2740
CK = ASC(K$)
IF CK = 27 OR CK = 32 THEN 2810
IF CK >= 97 AND CK <= 122 THEN CK = CK - 32
IF CK < 65 OR CK > 90 THEN 2740
K$ = CHR$(CK)
RETURN
END
WHAT THE OTHER GUY IS DOING

DABASAP (Greg Griffin) is collecting electronic mail addresses for ACA members. If you have an Internet, Compuserve or other electronic mail addresses, contact him at vlad@holonet.net.

NAGUKENU (James Lancaster) is currently working on BASIC implementations of the Haeglin machines. He was an artillery officer in Korea and used the M-209 “Convertor.” He is also reworking some BASIC programs to crack Playfairs. He would like to contact any others of the Krewe that have interest in either Haeglin or Playfair.

BINO (W. H. Edwards) continues to juggle the Chaocipher. Any progress from others?

GRYPHON (Richard Outerbridge) is using a Macintosh and writing in C and 68000 assembler, optimizing implementations of DES and public key for public domain use and distribution. He asks “Anyone who has a pubkey to share why not send it out to the group? Maybe there will come a time when one reason for going to the CON will be exchanging pubkeys face-to-face.”

G4EGG (Wilfred Higginson) interest was peaked by the reference to QuickBASIC in CS#17. He asks if there are any book lists and reviews covering QuickBASIC for the beginner? Perhaps even a definitive article comparing different BASIC dialects?

Bill Corcoran has even kinder things to say about Spectra Publishing’s PowerBasic. He notes the rapid execution speed and the QUAD integer value it supports, giving a range of $2^{63}$-1 as opposed to QuickBasic and TrueBasic’s $2^{31}$-1 and GW-BASIC’s $2^{15}$-1.

SCRYER (Jim Gillogy) is alive and well after the southern California wildfires. Although flames came very close (at one point even on the roof), after a six hour battle firefighters saved Jim’s house. The extent of his loss was some burned and scorched trees — “...not counting the broken front door... it didn’t occur to us to leave it open for the firefighters to access!” Which brings up an interesting question: How are your cryptographic materials protected from loss, whether fire, flood, theft, etc.?

ACA COMPUTER BULLETIN BOARD UPDATE

The ACA bulletin board system has made the move with the Editor out east. The system name is now called decode, and is available for both electronic mail (uunet!anagld!decode!dan) and file transfer, 24 hours a day at +1 410 730 6734.

There are several directories for cryptographic programs:

- /public/aca, containing ACA-related programs and files, including all issue disks and submitted programs;
- /public/crypto, containing general cryptographic programs, files, and documents;
- /public/des, containing code and executables implementing the Data Encryption Standard;
- /public/pgp, with code, executables and documents relating to the public key system Pretty Good Privacy.